

COBOL for AIX



Language Reference

Version 4.1

COBOL for AIX



Language Reference

Version 4.1

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 613.

Second Edition (September 2010)

This edition applies to Version 4.1 of IBM COBOL for AIX (program number 5724-Z87) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

You can order publications online at www.ibm.com/shop/publications/order/, or order by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. Eastern Standard Time (EST). The phone number is (800)879-2755. The fax number is (800)445-9269.

You can also order publications through your IBM representative or the IBM branch office serving your locality.

© Copyright IBM Corporation 1995, 2010.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables xi

Preface xiii

About this information	xiii
How to read the syntax diagrams.	xiii
IBM extensions	xvi
Obsolete language elements	xvi
DBCS notation	xvii
Acknowledgment	xvii
Summary of changes	xviii
Version 4.1 (September 2010)	xviii
How to send your comments.	xviii
Accessibility	xix
Interface information	xix
Keyboard navigation	xix
Accessibility of this information	xix
IBM and accessibility	xx

Part 1. COBOL language structure . . 1

Chapter 1. Characters. 3

Chapter 2. Character sets and code pages. 5

Compile-time code page	5
Runtime code page	5
Character encoding units	6
Single-byte code pages	6
Multibyte code pages	6
Unicode UTF-16	7

Chapter 3. Character-strings 9

COBOL words with single-byte characters	9
User-defined words with multibyte characters.	10
User-defined words.	11
System-names	12
Function-names	12
Reserved words	12
Figurative constants	13
Special registers	16
ADDRESS OF	17
DEBUG-ITEM	17
JNIENVPTR	19
LENGTH OF	19
LINAGE-COUNTER	20
RETURN-CODE	21
SHIFT-OUT and SHIFT-IN	21
SORT-CONTROL	22
SORT-CORE-SIZE	22
SORT-FILE-SIZE	23
SORT-MESSAGE	23
SORT-MODE-SIZE	23
SORT-RETURN	24
TALLY	24

WHEN-COMPILED	24
XML-CODE	25
XML-EVENT	26
XML-NTEXT	27
XML-TEXT	28
Literals	28
Alphanumeric literals	29
Numeric literals	31
DBCS literals	32
National literals	34
PICTURE character-strings	36
Comments.	37

Chapter 4. Separators 39

Rules for separators	39
--------------------------------	----

Chapter 5. Sections and paragraphs . . 43

Sentences, statements, and entries	43
Entries	43
Clauses.	44
Sentences	44
Statements.	44
Phrases.	44

Chapter 6. Reference format 45

Sequence number area.	45
Indicator area.	45
Area A	46
Division headers.	46
Section headers	46
Paragraph headers or paragraph names	47
Level indicators (FD and SD) or level-numbers (01 and 77)	47
DECLARATIVES and END DECLARATIVES	47
End program, end class, and end method markers	47
Area B	48
Entries, sentences, statements, clauses	48
Continuation lines	48
Area A or Area B	50
Level-numbers	50
Comment lines	50
Floating comment indicator (*>)	51
Compiler-directing statements	51
Compiler directives.	51
Debugging lines	52
Pseudo-text	52
Blank lines	52
Source conversion utility (scu)	52

Chapter 7. Scope of names 53

Types of names	53
External and internal resources	55
Resolution of names	56
Names within programs	56

Names within a class definition	57
Chapter 8. Referencing data names, copy libraries, and procedure division names	59
Uniqueness of reference	59
Qualification	59
Identical names	60
References to COPY libraries	60
References to procedure division names	60
References to data division names	61
Condition-name	64
Index-name	65
Index data item	65
Subscripting	65
Reference modification	68
Function-identifier	71
Data attribute specification	72
Chapter 9. Transfer of control	73
Chapter 10. Millennium Language Extensions and date fields	75
Millennium Language Extensions syntax	75
Terms and concepts.	76
Date field	76
Nondate	77
Century window	77
Part 2. COBOL source unit structure	79
Chapter 11. COBOL program structure	81
Nested programs	83
Conventions for program-names	84
Chapter 12. COBOL class definition structure	87
Chapter 13. COBOL method definition structure	91
Part 3. Identification division	93
Chapter 14. Identification division	95
PROGRAM-ID paragraph	98
CLASS-ID paragraph	100
General rules	101
Inheritance	101
FACTORY paragraph.	101
OBJECT paragraph	102
METHOD-ID paragraph.	102
Method signature	102
Method overloading, overriding, and hiding	102
Optional paragraphs	103

Part 4. Environment division	105
Chapter 15. Configuration section	107
SOURCE-COMPUTER paragraph	108
OBJECT-COMPUTER paragraph	108
SPECIAL-NAMES paragraph	110
ALPHABET clause	113
SYMBOLIC CHARACTERS clause	116
CLASS clause	116
CURRENCY SIGN clause	117
DECIMAL-POINT IS COMMA clause	118
REPOSITORY paragraph	118
General rules	120
Identifying and referencing a class	120
Chapter 16. Input-Output section	121
FILE-CONTROL paragraph.	122
SELECT clause	126
ASSIGN clause	126
Assignment name for user-defined words and literals.	128
Assignment name for data-names and environment variables	130
File concatenation	131
RESERVE clause	133
ORGANIZATION clause	133
File organization	133
PADDING CHARACTER clause	136
RECORD DELIMITER clause	136
ACCESS MODE clause	137
File organization and access modes	137
Access modes	138
Relationship between data organizations and access modes	138
RECORD KEY clause.	139
ALTERNATE RECORD KEY clause	140
RELATIVE KEY clause	141
PASSWORD clause	141
FILE STATUS clause	141
I-O-CONTROL paragraph	143
RERUN clause	145
SAME AREA clause	146
SAME RECORD AREA clause.	146
SAME SORT AREA clause	146
SAME SORT-MERGE AREA clause	147
MULTIPLE FILE TAPE clause	147
APPLY WRITE-ONLY clause	147
Part 5. Data division	149
Chapter 17. Data division overview	151
File section	152
Working-storage section.	153
Local-storage section	154
Linkage section.	155
Data units	155
File data	155
Program data	156
Method data	156

Factory data	156
Instance data	156
Data relationships	156
Levels of data	157
Levels of data in a record description entry	157
Special level-numbers	159
Indentation	159
Classes and categories of group items	159
Classes and categories of data	160
Category descriptions	162
Alignment rules	164
Character-string and item size	165
Signed data	166
Operational signs	166
Editing signs	166

Chapter 18. Data division--file description entries 167

File section	172
EXTERNAL clause	172
GLOBAL clause	173
BLOCK CONTAINS clause	173
RECORD clause	174
Format 1	175
Format 2	175
Format 3	175
LABEL RECORDS clause	177
VALUE OF clause	177
DATA RECORDS clause	178
LINAGE clause	178
LINAGE-COUNTER special register	180
RECORDING MODE clause	180
CODE-SET clause	180

Chapter 19. Data division--data description entry 181

Format 1	181
Format 2	182
Format 3	182
Level-numbers	182
BLANK WHEN ZERO clause	184
DATE FORMAT clause	184
Semantics of windowed date fields	185
Restrictions on using date fields	186
EXTERNAL clause	189
GLOBAL clause	189
JUSTIFIED clause	190
GROUP-USAGE clause	191
OCCURS clause	192
Fixed-length tables	193
ASCENDING KEY and DESCENDING KEY phrases	193
INDEXED BY phrase	195
Variable-length tables	195
OCCURS DEPENDING ON clause	196
PICTURE clause	198
Symbols used in the PICTURE clause	199
Character-string representation	203
Data categories and PICTURE rules	204
PICTURE clause editing	211

Simple insertion editing	212
Special insertion editing	212
Fixed insertion editing	213
Floating insertion editing	214
Zero suppression and replacement editing	215
REDEFINES clause	216
REDEFINES clause considerations	218
REDEFINES clause examples	218
Undefined results	219
RENAMES clause	220
SIGN clause	222
SYNCHRONIZED clause	223
Slack bytes	226
Slack bytes within records	226
Slack bytes between records	228
USAGE clause	229
Computational items	231
DISPLAY phrase	233
DISPLAY-1 phrase	234
FUNCTION-POINTER phrase	234
INDEX phrase	234
NATIONAL phrase	235
OBJECT REFERENCE phrase	235
POINTER phrase	236
PROCEDURE-POINTER phrase	237
NATIVE phrase	238
VALUE clause	239
Format 1	239
Format 2	241
Format 3	244

Part 6. Procedure division. 245

Chapter 20. Procedure division structure 249

Requirements for a method procedure division	250
The procedure division header	251
The USING phrase	252
RETURNING phrase	254
References to items in the linkage section	254
Declaratives	255
Procedures	256
Arithmetic expressions	257
Arithmetic operators	258
Arithmetic with date fields	260
Conditional expressions	262
Simple conditions	263
Class condition	263
Condition-name condition	265
Relation conditions	266
General relation conditions	267
Data pointer relation conditions	277
Procedure-pointer and function-pointer relation conditions	278
Object-reference relation conditions	278
Sign condition	279
Switch-status condition	280
Complex conditions	280
Negated simple conditions	281
Combined conditions	282

Abbreviated combined relation conditions	284
Statement categories	286
Imperative statements	286
Conditional statements	288
Delimited scope statements.	289
Explicit scope terminators	290
Implicit scope terminators	290
Compiler-directing statements	290
Statement operations	291
CORRESPONDING phrase	291
GIVING phrase	292
ROUNDED phrase	292
SIZE ERROR phrases	293
Arithmetic statements	294
Arithmetic statement operands	294
Data manipulation statements	296
Input-output statements	296
Common processing facilities	296

Chapter 21. Procedure division statements 303

ACCEPT statement	304
Data transfer	304
System date-related information transfer	305
DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, and TIME.	306
ADD statement.	308
ROUNDED phrase	310
SIZE ERROR phrases	310
CORRESPONDING phrase (format 3)	310
END-ADD phrase	311
ALTER statement	312
Segmentation considerations	312
CALL statement	314
USING phrase	316
BY REFERENCE phrase	317
BY CONTENT phrase	317
BY VALUE phrase.	318
RETURNING phrase	319
ON EXCEPTION phrase.	320
NOT ON EXCEPTION phrase.	320
ON OVERFLOW phrase.	320
END-CALL phrase	321
CANCEL statement	322
CLOSE statement	324
Effect of CLOSE statement on file types	326
COMPUTE statement.	328
ROUNDED phrase	329
SIZE ERROR phrases.	329
END-COMPUTE phrase.	329
CONTINUE statement	330
DELETE statement	331
Sequential access mode	331
Random or dynamic access mode	332
END-DELETE phrase.	332
DISPLAY statement	333
DIVIDE statement.	335
ROUNDED phrase	338
REMAINDER phrase.	338
SIZE ERROR phrases.	338
END-DIVIDE phrase	339

ENTRY statement	340
USING phrase	340
EVALUATE statement	341
END-EVALUATE phrase	342
Determining values	342
Comparing selection subjects and objects	343
Executing the EVALUATE statement	344
EXIT statement	345
EXIT METHOD statement	346
EXIT PROGRAM statement	347
GOBACK statement	348
GO TO statement	349
Unconditional GO TO	349
Conditional GO TO	349
Altered GO TO.	350
MORE-LABELS GO TO	350
IF statement.	351
END-IF phrase	351
Transferring control	352
Nested IF statements	352
INITIALIZE statement	353
REPLACING phrase	354
INITIALIZE statement rules	354
INSPECT statement	356
TALLYING phrase (formats 1 and 3)	359
REPLACING phrase (formats 2 and 3)	360
BEFORE and AFTER phrases (all formats).	361
CONVERTING phrase (format 4).	362
Data flow	363
Comparison cycle	364
Example of the INSPECT statement	364
INVOKE statement	366
USING phrase	368
BY VALUE phrase.	368
RETURNING phrase	369
ON EXCEPTION phrase.	370
NOT ON EXCEPTION phrase.	370
END-INVOKE phrase	371
Interoperable data types for COBOL and Java	371
Miscellaneous argument types for COBOL and Java	373
MERGE statement.	374
ASCENDING/DESCENDING KEY phrase	374
COLLATING SEQUENCE phrase.	376
USING phrase	377
GIVING phrase.	377
OUTPUT PROCEDURE phrase	378
MERGE special registers.	378
Segmentation considerations	379
MOVE statement	380
Elementary moves.	381
Moves involving file record areas.	386
Group moves	386
MULTIPLY statement.	388
ROUNDED phrase	390
SIZE ERROR phrases.	390
END-MULTIPLY phrase.	390
OPEN statement	391
General rules	393
Label records	393
OPEN statement notes	393

PERFORM statement	396	START statement	441
Basic PERFORM statement	396	KEY phrase	441
END-PERFORM	398	INVALID KEY phrases	442
PERFORM with TIMES phrase	398	END-START phrase	442
PERFORM with UNTIL phrase	399	Indexed files	442
PERFORM with VARYING phrase	400	Relative files.	443
Varying identifiers.	401	STOP statement	444
Varying two identifiers	402	STRING statement.	445
Varying three identifiers.	404	ON OVERFLOW phrases	447
Varying more than three identifiers	404	END-STRING phrase.	447
Varying phrase rules	405	Data flow	448
READ statement	406	SUBTRACT statement	450
KEY IS phrase	408	ROUNDED phrase	452
AT END phrases	408	SIZE ERROR phrases.	452
INVALID KEY phrases	408	CORRESPONDING phrase (format 3)	452
END-READ phrase	408	END-SUBTRACT phrase	453
Multiple record processing	408	UNSTRING statement	454
Sequential access mode	409	DELIMITED BY phrase	456
Random access mode.	411	INTO phrase	457
Dynamic access mode	412	POINTER phrase	457
READ statement notes:	412	TALLYING IN phrase	457
RELEASE statement	413	ON OVERFLOW phrases	457
RETURN statement	415	END-UNSTRING phrase	458
AT END phrases	416	Data flow	458
END-RETURN phrase	416	Example of the UNSTRING statement	460
REWRITE statement	417	WRITE statement	462
INVALID KEY phrases	418	ADVANCING phrase.	465
END-REWRITE phrase	418	END-OF-PAGE phrases	466
Reusing a logical record.	418	INVALID KEY phrases	466
Sequential files	418	END-WRITE phrase	467
Indexed files	418	WRITE for sequential files	467
Relative files.	419	WRITE for indexed files.	468
SEARCH statement	420	WRITE for relative files	468
Serial search.	421	XML GENERATE statement	470
Binary search	423	Nested XML GENERATE or XML PARSE	
Search statement considerations	426	statements	475
AT END and WHEN phrases	426	Operation of XML GENERATE	476
NEXT SENTENCE.	426	Format conversion of elementary data	477
END-SEARCH phrase	426	Trimming of generated XML data	478
SET statement	427	XML element name and attribute name	
Format 1: SET for basic table handling	427	formation.	479
Format 2: SET for adjusting indexes	428	XML PARSE statement	480
Format 3: SET for external switches	429	Nested XML GENERATE or XML PARSE	
Format 4: SET for condition-names	430	statements	483
Format 5: SET for USAGE IS POINTER data		Control flow.	484
items	430		
Format 6: SET for procedure-pointer and			
function-pointer data items.	431		
Format 7: SET for USAGE OBJECT REFERENCE			
data items	433		
SORT statement	434		
ASCENDING KEY and DESCENDING KEY			
phrases	435		
DUPLICATES phrase.	436		
COLLATING SEQUENCE phrase.	436		
USING phrase	437		
INPUT PROCEDURE phrase	437		
GIVING phrase.	438		
OUTPUT PROCEDURE phrase	439		
SORT special registers	439		
Segmentation considerations	440		

Part 7. Intrinsic functions 487

Chapter 22. Intrinsic functions 489

Specifying a function.	489
Function definition and evaluation	490
Types of functions.	490
Rules for usage.	491
Arguments	492
Examples.	494
ALL subscripting	494
Function definitions	496
ACOS	500
ANNUITY	500
ASIN	501

ATAN	501
CHAR.	501
COS	502
CURRENT-DATE	502
DATE-OF-INTEGER	504
DATE-TO-YYYYMMDD	504
DATEVAL	505
DAY-OF-INTEGER	506
DAY-TO-YYYYDDD	507
DISPLAY-OF	507
FACTORIAL.	509
INTEGER	509
INTEGER-OF-DATE	510
INTEGER-OF-DAY	510
INTEGER-PART	511
LENGTH.	511
LOG	513
LOG10	513
LOWER-CASE	513
MAX	514
MEAN	515
MEDIAN.	516
MIDRANGE.	516
MIN	517
MOD	518
NATIONAL-OF	518
NUMVAL	519
NUMVAL-C.	520
ORD	522
ORD-MAX	523
ORD-MIN	523
PRESENT-VALUE	524
RANDOM	524
RANGE	525
REM	526
REVERSE.	526
SIN.	527
SQRT	527
STANDARD-DEVIATION	528
SUM	528
TAN	529
UNDATE.	529
UPPER-CASE	530
VARIANCE	530
WHEN-COMPILED	531
YEAR-TO-YYYY	532
YEARWINDOW	533

Part 8. Compiler-directing statements 535

Chapter 23. Compiler-directing statements 537

BASIS statement	537
CBL (PROCESS) statement	538
*CONTROL (*CBL) statement	538
Source code listing	540
Object code listing.	540
Storage map listing	540
COPY statement	540

SUPPRESS phrase	543
REPLACING phrase	543
Replacement and comparison rules	544
DELETE statement	547
EJECT statement	548
ENTER statement	548
INSERT statement	549
READY or RESET TRACE statement	550
REPLACE statement	550
Continuation rules for pseudo-text	552
Comparison operation	552
REPLACE statement notes	553
SERVICE LABEL statement.	554
SERVICE RELOAD statement	554
SKIP statements	554
TITLE statement	555
USE statement	556
EXCEPTION/ERROR declarative.	556
Precedence rules for nested programs	557
LABEL declarative.	558
DEBUGGING declarative	558

Chapter 24. Compiler directives 561

CALLINTERFACE.	561
Syntax and general rules	561

Part 9. Appendixes 563

Appendix A. IBM extensions 565

Appendix B. Compiler limits 577

File input-output limitations	581
---	-----

| Appendix C. Source conversion utility (scu) 583

Appendix D. EBCDIC and ASCII collating sequences 585

EBCDIC collating sequence.	585
US English ASCII code page	588

Appendix E. Source language debugging. 593

Debugging lines	593
Debugging sections	593
DEBUG-ITEM special register	594
Activate compile-time switch	594
Activate object-time switch	594

Appendix F. Reserved words. 595

Appendix G. Locale considerations 609

Compile-time versus runtime locale	609
Code pages	609
Collating sequences	610
Supported locales	611

Notices 613

Programming interface information	614	COBOL for AIX	645
Trademarks	614	Related publications	645
Glossary	617	Index	647
List of resources	645		

Tables

1. Basic COBOL character set	3	35. Abbreviated combined conditions: permissible element sequences	285
2. DEBUG-ITEM subfield contents	18	36. Abbreviated combined conditions: unabbreviated equivalents	286
3. Contents of XML-EVENT and XML-TEXT or XML-NTEXT special registers	26	37. Exponentiation size error conditions	293
4. Separators	39	38. How the composite of operands is determined	295
5. Meanings of environment names.	112	39. File status key values and meanings	297
6. Types of files	122	40. Sequential files and CLOSE statement phrases	326
7. Assignment name for user-defined words	129	41. Indexed and relative file types and CLOSE statement phrases	326
8. Assignment name for literals	129	42. Line-sequential file types and CLOSE statement phrases	326
9. Assignment name for data-names and environment variables	131	43. Meanings of key letters for sequential file types	327
10. Classes and categories of group items	160	44. Treatment of the content of data items	362
11. Class, category, and usage of elementary data items	161	45. Interoperable Java and COBOL data types	371
12. Classes and categories of functions	161	46. Interoperable COBOL and Java array and String data types	372
13. Classes and categories of literals.	162	47. COBOL miscellaneous argument types and corresponding Java types	373
14. Where national group items are processed as groups.	191	48. COBOL literal argument types and corresponding Java types	373
15. PICTURE clause symbol meanings	200	49. Valid and invalid elementary moves	384
16. Numeric types	205	50. Moves involving date fields	386
17. Data categories	211	51. Availability of a file	393
18. SYNCHRONIZE clause effect on other language elements.	224	52. Permissible statements for sequential files	394
19. Relation test references for condition-names	243	53. Permissible statements for indexed and relative files.	394
20. Binary and unary operators	258	54. Permissible statements for line-sequential files	395
21. Valid arithmetic symbol pairs.	259	55. Sending and receiving fields for format-1 SET statement	428
22. Results of using date fields in addition	260	56. Sending and receiving fields for format-5 SET statement	431
23. Results of using date fields in subtraction	261	57. Character positions examined when DELIMITED BY is not specified.	459
24. Storing arithmetic results that involve date fields when ON SIZE ERROR is specified	262	58. Table of functions.	497
25. Valid forms of the class condition for different types of data items	265	59. Execution of debugging declaratives	559
26. Relational operators and their meanings	268	60. IBM extension language elements	565
27. Comparisons involving data items and literals.	270	61. Compiler limits	577
28. Comparisons involving figurative constants	271	62. EBCDIC collating sequence	585
29. Comparisons for index-names and index data items	275	63. ASCII collating sequence	588
30. Comparisons with date fields.	276	64. Reserved words	595
31. Permissible comparisons for USAGE POINTER, NULL, and ADDRESS OF	277		
32. Logical operators and their meanings	280		
33. Combined conditions—permissible element sequences	282		
34. Logical operators and evaluation results of combined conditions.	283		

Preface

About this information

This information describes the COBOL language supported by IBM® COBOL for AIX®, referred to in this information as COBOL for AIX.

Use this information in conjunction with the *COBOL for AIX Programming Guide*.

How to read the syntax diagrams

Use the following description to read the syntax diagrams in this document:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The >>--- symbol indicates the beginning of a syntax diagram.

The ---> symbol indicates that the syntax diagram is continued on the next line.

The >--- symbol indicates that the syntax diagram is continued from the previous line.

The --->< symbol indicates the end of a syntax diagram.

Diagrams of syntactical units other than complete statements start with the >--- symbol and end with the ---> symbol.

- Required items appear on the horizontal line (the main path).

Format

A horizontal line with a double arrow pointing right at the start and a double arrow pointing left at the end. The text "STATEMENT—required item" is placed on the line, with a vertical line segment extending downwards from the word "required" to the word "item".

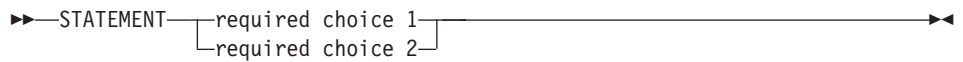
- Optional items appear below the main path.

Format

A horizontal line with a double arrow pointing right at the start and a double arrow pointing left at the end. The text "STATEMENT" is placed on the line, followed by a vertical line segment extending downwards to the text "optional item".

- When you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path.

Format



If choosing one of the items is optional, the entire stack appears below the main path.

Format



- An arrow returning to the left above the main line indicates an item that can be repeated.

Format

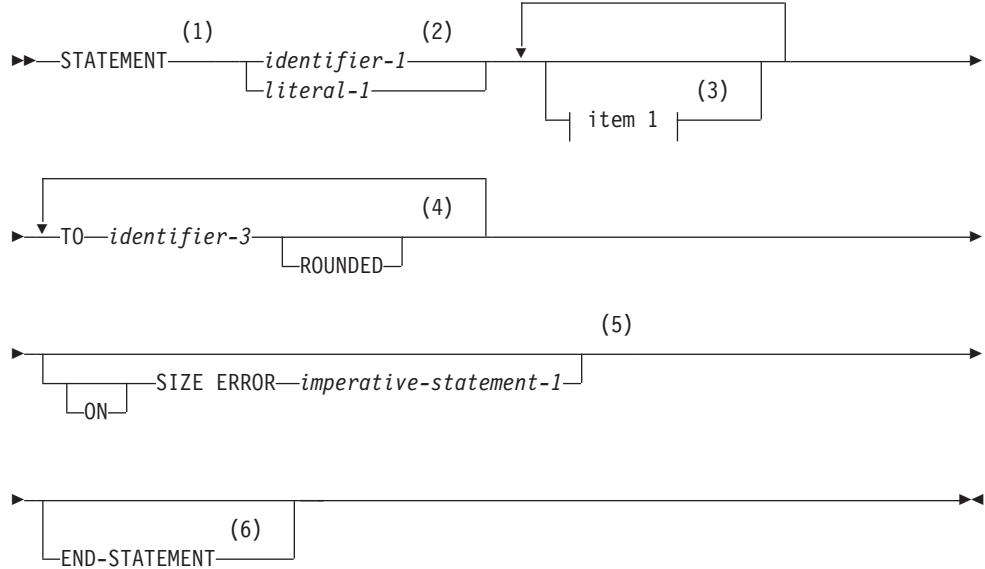


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Variables appear in italic lowercase letters (for example, *parm**x*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, they must be entered as part of the syntax.

The following example shows how the syntax is used.

Format



item 1:



Notes:

- 1 The STATEMENT keyword must be specified and coded as shown.
- 2 This operand is required. Either *identifier-1* or *literal-1* must be coded.
- 3 The item 1 fragment is optional; it can be coded or not, as required by the application. If item 1 is coded, it can be repeated with each entry separated by one or more COBOL separators. Entry selections allowed for this fragment are described at the bottom of the diagram.
- 4 The operand *identifier-3* and associated TO keyword are required and can be repeated with one or more COBOL separators separating each entry. Each entry can be assigned the keyword ROUNDED.
- 5 The ON SIZE ERROR phrase with associated *imperative-statement-1* is optional. If the ON SIZE ERROR phrase is coded, the keyword ON is optional.
- 6 The END-STATEMENT keyword can be coded to end the statement. It is not a required delimiter.

IBM extensions

IBM extensions generally add features, syntax, or rules that are not specified in the ANSI and ISO COBOL standards. In this document, the term Standard COBOL 85 refers to those standards.

Extensions range from minor relaxation of rules to major capabilities, such as XML support, Unicode support, object-oriented COBOL for Java interoperability, and DBCS character handling.

The rest of this document describes the complete language without identifying extensions. You will need to review Appendix A, “IBM extensions,” on page 565 and the *Compiler options* in the *COBOL for AIX Programming Guide* if you want to use only standard language elements.

Obsolete language elements

Obsolete language elements are elements that are categorized as *obsolete* in Standard COBOL 85. Those elements are not part of Standard COBOL 2002.

This does *not* imply that IBM will remove Standard COBOL 85 obsolete elements from a future release of COBOL for AIX.

The following are language elements that Standard COBOL 85 categorized as obsolete:

- ALTER statement
- AUTHOR paragraph
- Comment entry
- DATA RECORDS clause
- DATE-COMPILED paragraph
- DATE-WRITTEN paragraph
- DEBUG-ITEM special register
- Debugging sections
- ENTER statement
- GO TO without a specified procedure-name
- INSTALLATION paragraph
- LABEL RECORDS clause
- MEMORY SIZE clause
- MULTIPLE FILE TAPE clause
- RERUN clause
- REVERSED phrase
- SECURITY paragraph
- Segmentation module
- STOP *literal* format of the STOP statement
- USE FOR DEBUGGING declarative
- VALUE OF clause
- The figurative constant ALL *literal* with a length greater than one, when the figurative constant is associated with a numeric or numeric-edited item

DBCS notation

In this document, DBCS characters are shown in this form: D1D2D3. Latin alphabet characters in DBCS representation are shown in this form: .A.B.C.

Notes

- In EBCDIC DBCS data containing mixed single-byte and double-byte characters, double-byte character strings are delimited by shift-out and shift-in characters. In this document, the shift-out delimiter is represented pictorially by the < character, and the shift-in character is represented pictorially by the > character. The single-byte EBCDIC codes for the shift-out and shift-in delimiters are X'0E' and X'0F', respectively. The <> symbol denotes contiguous shift-out and shift-in characters. The >< symbol denotes contiguous shift-in and shift-out characters.
- In ASCII DBCS data containing mixed single-byte and double-byte characters, double-byte character strings are not delimited by shift-out and shift-in characters.

Acknowledgment

The following extract from Government Printing Office Form Number 1965-0795689 is presented for the information and guidance of the user:

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention COBOL in acknowledgment of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection there with.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of copyrighted material:

- FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC(R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation
- IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM
- FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Note: The Conference on Data Systems Languages (CODASYL), mentioned above, is no longer in existence.

Summary of changes

This section lists the major changes that have been made to this information. The changes have an associated cross-reference for your convenience. The changes with respect to COBOL for AIX Version 3 (both technical changes and editorial changes to terminology) are marked by a vertical bar (|) in the left margin in the PDF version.

Version 4.1 (September 2010)

- The components of the ASSIGN clause have been clarified, and associated examples have been added (“ASSIGN clause” on page 126).
- The information about reference format has been updated to reflect that:
 - Source text can optionally vary in length from 0 to 252 bytes (*extended source format*) or end at column 72 (*fixed source format*) as in earlier versions of IBM COBOL for AIX (Chapter 6, “Reference format,” on page 45).
 - A new floating comment indicator (the character string ‘*>’) can be coded in either fixed or extended source format to indicate that the ensuing text on a line is an inline comment. (“Floating comment indicator (*>)” on page 51).
- The following information about file concatenation has been added:
 - Multiple input files can be concatenated by separating the file identifiers with a colon (:). Concatenation is supported in all of the IBM COBOL for AIX file systems (“File concatenation” on page 131).
 - File status key 33 is a new error condition (“File status key” on page 297).
- Information about the source conversion utility, scu, has been added. scu helps you normalize non-IBM COBOL source and free-format COBOL source so that it can be compiled by IBM COBOL for AIX (Appendix C, “Source conversion utility (scu),” on page 583).
- File input-output limitations have been updated for the new DB2[®] file system (“File input-output limitations” on page 581).
- Several sections have been updated to reflect that the underscore character (_) is supported in user-defined words such as data-names and program-names. Underscores are also supported in the literal form of program-names.

The following terminology change was also made in this release: *VSAM* as a generic reference to either the SdU (SMARTdata Utilities) or Encina SFS (Structured File Server) file system has been replaced with the more precise terms *SdU* or *SFS* where appropriate.

How to send your comments

Your feedback is important in helping us to provide accurate, high-quality information. If you have comments about this document or any other documentation for this product, contact us in one of the following ways:

- Fill out the Readers' Comment Form at the back of this document, and return it by mail or give it to an IBM representative. If there is no form at the back of this document, address your comments to:

IBM Corporation
Reader Comments

DTX/E269
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

- Use the Online Readers' Comments Form at www.ibm.com/software/awdtools/rcf/.
- Send your comments to the following address: comments@us.ibm.com

Be sure to include the name of the document, the publication number of the document, the version of the product, and, if applicable, the specific location (for example, page number or section heading) of the text that you are commenting on.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way that IBM believes appropriate without incurring any obligation to you.

Accessibility

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully. The accessibility features in AIX provide accessibility for COBOL for AIX.

The major accessibility features in AIX are:

- Interfaces that are commonly used by screen readers and screen-magnifier software
- Keyboard-only navigation
- An accessible command-line interface through a direct attached or networked remote Windows or Linux system using AT and terminal emulation software

Interface information

Assistive technology products work with the user interfaces that are found in AIX. For specific guidance information, see the documentation for the assistive technology product that you use to access AIX interfaces.

Keyboard navigation

For information about accessing AIX interfaces, see Accessibility features for AIX in the IBM System p and AIX Information Center at publib16.boulder.ibm.com/pseries/index.htm.

Accessibility of this information

The English-language XHTML format of this information that will be provided in the IBM AIX Compiler Information Center at www.ibm.com/software/awdtools/cobol/aix/library is accessible to visually impaired individuals who use a screen reader.

To enable your screen reader to accurately read syntax diagrams, source code examples, and text that contains the period or comma PICTURE symbols, you must set the screen reader to speak all punctuation.

IBM and accessibility

See the IBM Human Ability and Accessibility Center at www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Part 1. COBOL language structure

Chapter 1. Characters	3	Comments.	37
Chapter 2. Character sets and code pages	5	Chapter 4. Separators	39
Compile-time code page	5	Rules for separators	39
Runtime code page	5	Chapter 5. Sections and paragraphs	43
Character encoding units	6	Sentences, statements, and entries	43
Single-byte code pages	6	Entries	43
Multibyte code pages	6	Clauses	44
USAGE DISPLAY	6	Sentences	44
USAGE DISPLAY-1	7	Statements	44
Unicode UTF-16	7	Phrases	44
Chapter 3. Character-strings	9	Chapter 6. Reference format	45
COBOL words with single-byte characters	9	Sequence number area.	45
User-defined words with multibyte characters	10	Indicator area.	45
User-defined words.	11	Area A	46
System-names	12	Division headers.	46
Function-names	12	Section headers	46
Reserved words	12	Paragraph headers or paragraph names	47
Figurative constants	13	Level indicators (FD and SD) or level-numbers	
Special registers	16	(01 and 77)	47
ADDRESS OF	17	DECLARATIVES and END DECLARATIVES	47
DEBUG-ITEM	17	End program, end class, and end method	
JNIENVPTR	19	markers	47
LENGTH OF	19	Area B	48
LINAGE-COUNTER	20	Entries, sentences, statements, clauses	48
RETURN-CODE	21	Continuation lines	48
SHIFT-OUT and SHIFT-IN	21	Continuation of alphanumeric and national	
SORT-CONTROL	22	literals	49
SORT-CORE-SIZE	22	Area A or Area B	50
SORT-FILE-SIZE	23	Level-numbers	50
SORT-MESSAGE	23	Comment lines	50
SORT-MODE-SIZE	23	Floating comment indicator (*>)	51
SORT-RETURN	24	Compiler-directing statements	51
TALLY	24	Compiler directives.	51
WHEN-COMPILED	24	Debugging lines	52
XML-CODE	25	Pseudo-text	52
XML-EVENT	26	Blank lines	52
XML-NTEXT	27	Source conversion utility (scu)	52
XML-TEXT	28	Chapter 7. Scope of names	53
Literals	28	Types of names	53
Alphanumeric literals	29	External and internal resources	55
Basic alphanumeric literals	29	Resolution of names	56
Hexadecimal notation for alphanumeric		Names within programs	56
literals	30	Names within a class definition	57
Null-terminated alphanumeric literals	31	Chapter 8. Referencing data names, copy	
Numeric literals	31	libraries, and procedure division names	59
Rules for floating-point literal values	32	Uniqueness of reference	59
DBCS literals	32	Qualification	59
DBCS literals with the SOSI compiler option	33	Qualification rules	60
Where DBCS literals can be used	33	Identical names	60
National literals	34	References to COPY libraries	60
Basic national literals	34		
Hexadecimal notation for national literals	35		
Where national literals can be used	36		
PICTURE character-strings	36		

References to procedure division names	60
References to data division names	61
Simple data reference	61
Identifiers	62
Condition-name	64
Index-name	65
Index data item	65
Subscripting	65
Subscripting using data-names	67
Subscripting using index-names (indexing)	67
Relative subscripting	68
Reference modification	68
Evaluation of operands	70
Reference modification examples	71
Function-identifier	71
Data attribute specification	72
 Chapter 9. Transfer of control	 73
 Chapter 10. Millennium Language Extensions	
and date fields	75
Millennium Language Extensions syntax	75
Terms and concepts.	76
Date field	76
Windowed date field	76
Expanded date field	76
Year-last date field	76
Date format	76
Compatible date field	77
Nondate	77
Century window	77

Chapter 1. Characters

The most basic and indivisible unit of the COBOL language is the *character*. The basic character set includes the letters of the Latin alphabet, digits, and special characters. In the COBOL language, individual characters are joined to form *character-strings* and *separators*. Character-strings and separators, then, are used to form the words, literals, phrases, clauses, statements, and sentences that form the language.

The basic characters used in forming character-strings and separators in source code are shown in Table 1.

For certain language elements, the basic character set is extended with the following character sets, depending on the *code page* used at compile time:

- The ASCII Double-Byte Character Set (DBCS). DBCS characters occupy 2 adjacent bytes to represent one character. Characters represented in multiple bytes in source code (including DBCS characters) are referred to in this document as *multibyte characters*. A character-string containing only DBCS characters is also called a *DBCS character-string* or *double-byte character string*.
- UTF-8, an encoding form of the Unicode character set. UTF-8 characters occupy one-to-four bytes per character. UTF-8 characters that occupy 2 or more bytes are referred to in this document as *multibyte characters*.
- Extended UNIX Code (EUC). EUC characters occupy 1 byte to 4 bytes per character (or 1 byte to 3 bytes, depending on the code page). EUC characters that occupy 2 or more bytes are referred to in this document as *multibyte characters*.

Multibyte characters can be used in forming user-defined words.

The content of alphanumeric literals, comment lines, and comment entries can include any of the characters in the computer's compile-time character set, and can include both single-byte and multibyte characters.

Runtime data can include any characters from the runtime character set of the computer. The runtime character set of the computer can include alphanumeric characters, multibyte characters, and national characters. National characters are represented in UTF-16, a 16-bit encoding form of Unicode.

When the NSYMBOL (NATIONAL) compiler option is in effect, literals identified by the opening delimiter N" or N' are national literals and can contain any single-byte or multibyte characters, or both, that are valid for the compile-time code page. Characters contained in national literals are represented as national characters at run time.

For details, see "User-defined words with multibyte characters" on page 10, "DBCS literals" on page 32, and "National literals" on page 34.

Table 1. Basic COBOL character set

Character	Meaning
	Space
+	Plus sign

Table 1. Basic COBOL character set (continued)

Character	Meaning
-	Minus sign or hyphen
*	Asterisk
/	Forward slash or solidus
=	Equal sign
\$	Currency sign
,	Comma
;	Semicolon
.	Decimal point or period
"	Quotation mark
'	Apostrophe
(Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore
A - Z	Alphabet (uppercase)
a - z	Alphabet (lowercase)
0 - 9	Numeric characters

Chapter 2. Character sets and code pages

A *character set* is a set of letters, numbers, special characters, and other elements used to represent information. A character set is independent of a coded representation. A *coded character set* is the coded representation of a set of characters, where each character is assigned a numerical position, called a *code point*, in the encoding scheme. ASCII and EBCDIC are examples of types of coded character sets. Each variation of ASCII or EBCDIC is a specific coded character set.

The term *code page* refers to a coded character set. Each code page that IBM defines is identified by a *code page name*, for example IBM-1252, and a *coded character set identifier* (CCSID), for example 1252.

Compile-time code page

The compile-time code page can be an ASCII single-byte or ASCII double-byte code page, an EUC code page, or UTF-8. The specific code page is indicated by the compile-time locale.

The source program (including user-defined words and the content of alphanumeric, DBCS, and national literals) is encoded in the code page indicated by the locale in effect at compile time.

Runtime code page

The code page used at run time is determined by a combination of a data item's USAGE clause, the compiler options in effect, and the locale (or environment variable value) in effect.

When the CHAR(NATIVE) compiler option is in effect, data items described with USAGE DISPLAY or USAGE DISPLAY-1 are encoded in an ASCII, EUC, or UTF-8 code page as indicated by the runtime locale.

When the CHAR(EBCDIC) compiler option is in effect, data items described with USAGE DISPLAY or USAGE DISPLAY-1 are encoded in an EBCDIC code page, except when the NATIVE phrase is specified in the item's USAGE clause. If the NATIVE phrase is specified, the code page used is the ASCII, EUC, or UTF-8 code page indicated by the runtime locale.

For EBCDIC, the code page is determined from the EBCDIC_CODEPAGE environment variable, if set. If the EBCDIC_CODEPAGE environment variable is not set, the default EBCDIC code page associated with the current runtime locale is used. The default EBCDIC code page associated with each supported locale is identified in *Locales and code pages that are supported* in the *COBOL for AIX Programming Guide*.

The code page for data items described with USAGE NATIONAL and national literals is UTF-16BE (big endian), CCSID 1200. The source text representation of national literals is converted at run time from the compile-time code page to UTF-16BE.

A reference to *UTF-16* in this document is a reference to UTF-16BE.

Character encoding units

A *character encoding unit* (or *encoding unit*) is the unit of data that COBOL treats as a single character at run time. In this document, the terms *character* and *character position* refer to a single encoding unit.

The size of an encoding unit for data items and literals depends on the USAGE clause of the data item or the category of the literal as follows:

- For data items described with USAGE DISPLAY and for alphanumeric literals, an encoding unit is 1 byte, regardless of the code page used and regardless of the number of bytes used to represent a given *graphic character*.
- For data items described with USAGE DISPLAY-1 (DBCS data items) and for DBCS literals, an encoding unit is 2 bytes.
- For data items described with USAGE NATIONAL and for national literals, an encoding unit is 2 bytes.

The relationship between a graphic character and an encoding unit depends on the type of code page used for the data item or literal. The following are the types of runtime code pages:

- Single-byte ASCII or EBCDIC
- Multibyte ASCII-based (ASCII DBCS, EUC, or UTF-8) or multibyte EBCDIC DBCS
- Unicode UTF-16

See the following sections for the details of each type of code page.

Also see the section *Specifying the code page for character data* in the *COBOL for AIX Programming Guide*.

Single-byte code pages

You can use a single-byte code page in data items described with USAGE DISPLAY and in literals of category alphanumeric. An encoding unit is 1 byte and each graphic character is represented in 1 byte. For these data items and literals, you need not be concerned with encoding units.

Multibyte code pages

USAGE DISPLAY

You can use data encoded in a multibyte ASCII-based or EBCDIC code page in data items described with USAGE DISPLAY (category alphanumeric) and in literals of category alphanumeric. An encoding unit is 1 byte and the size of a graphic character varies from 1 byte to 4 bytes, depending on the code page.

When alphanumeric data items or literals contain multibyte data, programmers are responsible for ensuring that operations do not unintentionally separate the multiple encoding units that form a graphic character. Care should be taken with reference modification, and truncation during moves should be avoided. The COBOL runtime system does not check for a split between the encoding units that form a graphic character.

To avoid problems, you can convert alphanumeric literals and data items described with usage DISPLAY to national data (UTF-16) by moving the data items or literals to data items described with usage NATIONAL or by using the NATIONAL-OF intrinsic function. You can then perform operations on the national data with less concern for splitting graphic characters. You can convert the data back to USAGE DISPLAY by using the DISPLAY-OF intrinsic function.

When the source data that you convert is UTF-8, you need to be concerned about splitting graphic characters in the target usage NATIONAL data item because the resulting national data might contain UTF-16 surrogate pairs or combining character sequences as described in “Unicode UTF-16.”

USAGE DISPLAY-1

You can use double-byte characters of a multibyte ASCII DBCS or EBCDIC DBCS code page in data items described with USAGE DISPLAY-1 and in literals of category DBCS. An encoding unit is 2 bytes and each graphic character is represented in a single 2-byte encoding unit. For these data items and literals, you need not be concerned with encoding units.

You cannot use UTF-8 or EUC in data items described with USAGE DISPLAY-1.

Unicode UTF-16

You can use UTF-16 in data items described with USAGE NATIONAL. National literals are stored as UTF-16 characters regardless of the code page used for the source program. An encoding unit for data items of usage NATIONAL and national literals is 2 bytes.

For most of the characters in UTF-16, a graphic character is one encoding unit. Characters converted to UTF-16 from an EBCDIC, ASCII, or EUC code page are represented in one UTF-16 encoding unit. Some of the other graphic characters in UTF-16 are represented by a *surrogate pair* or a *combining character sequence*. A surrogate pair consists of two encoding units (4 bytes). A combining character sequence consists of a base character and one or more *combining marks* or a sequence of one or more combining marks (4 bytes or more, in 2-byte increments). In data items of usage NATIONAL, each 2-byte encoding unit is treated as a character.

When national data contains surrogate pairs or combining character sequences, programmers are responsible for ensuring that operations on national characters do not unintentionally separate the multiple encoding units that form a graphic character. Care should be taken with reference modification, and truncation during moves should be avoided. The COBOL runtime system does not check for a split between the encoding units that form a graphic character.

Chapter 3. Character-strings

A *character-string* is a character or a sequence of contiguous characters that forms a COBOL word, a literal, a PICTURE character-string, or a comment-entry. A character-string is delimited by separators.

A *separator* is a string of contiguous characters used to delimit character strings. Separators are described in detail under Chapter 4, “Separators,” on page 39.

Character strings and certain separators form *text words*. A text word is a character or a sequence of contiguous characters (possibly continued across lines) between character positions 8 and 72 in fixed source format, or positions 8 and 252 in extended source format inclusive in source text, library text, or pseudo-text. For more information about pseudo-text, see “Pseudo-text” on page 52.

Source text, library text, and pseudo-text can be written in single-byte characters and, for some character-strings, multibyte characters of the ASCII, UTF-8, or EUC code page indicated by the compile-time locale.

You can use single-byte and multibyte character-strings to form the following:

- COBOL words
- Literals
- Comment text

You can use only single-byte characters to form PICTURE character-strings.

COBOL words with single-byte characters

A COBOL *word* is a character-string that forms a user-defined word, a system-name, or a reserved word. The maximum size of a COBOL user-defined word is 30 bytes. The number of characters that can be specified depends on the code page indicated by the compile-time locale.

Except for arithmetic operators and relation characters, each character of a COBOL word is selected from the following set:

- Latin uppercase letters A through Z
- Latin lowercase letters a through z
- digits 0 through 9
- - (hyphen)
- _ (underscore)

The hyphen cannot appear as the first or last character in such words. The underscore cannot appear as the first character in such words. Most user-defined words (all except section-names, paragraph-names, priority-numbers, and level-numbers) must contain at least one alphabetic character. Priority numbers and level numbers need not be unique; a given specification of a priority-number or level-number can be identical to any other priority-number or level-number.

In COBOL words (but not in the content of alphanumeric, DBCS, and national literals), each lowercase single-byte alphabetic letter is considered to be equivalent to its corresponding single-byte uppercase alphabetic letter.

The following rules apply for all COBOL words:

- A reserved word cannot be used as a user-defined word or as a system-name.
- The same COBOL word, however, can be used as both a user-defined word and as a system-name. The classification of a specific occurrence of a COBOL word is determined by the context of the clause or phrase in which it occurs.

User-defined words with multibyte characters

When used in the context of user-defined words, the term *multibyte* refers to the following types of words:

- Words formed of DBCS characters, possibly combined with single-byte characters
- Words formed of UTF-8 characters that are composed of one or more bytes
- Words formed of EUC characters that are composed of one or more bytes.

The following are the rules for forming user-defined words with multibyte characters:

Contained characters

A user-defined word can consist of both single-byte and multibyte characters. If a character exists in both single-byte and multibyte forms, its single-byte and multibyte representations are not equivalent.

The single-byte characters in the user-defined word are limited to the following characters:

- Latin letters uppercase A through Z
- Latin letters lowercase a through z
- digits 0 through 9
- - (hyphen)
- _ (underscore)

The single-byte encoded hyphen cannot appear as the first or last character in such words.

The single-byte encoded underscore cannot appear as the first character in such words.

Uppercase and lowercase letters

In COBOL words, each lowercase single-byte encoded character "a" through "z" is considered to be equivalent to its corresponding single-byte encoded uppercase character. Multibyte-encoded uppercase and lowercase letters are not equivalent.

Value range

Valid value ranges for multibyte characters depend on the specific code page being used.

Maximum length

30 bytes. The number of characters that you can specify in 30 bytes varies depending on the source code page and the characters used in the user-defined word.

Continuation

Words formed with multibyte characters cannot be continued across lines.

Use of shift-out and shift-in characters

Applicable only when the dummy shift-out/shift-in (SOSI) compiler option is in effect. See *SOSI* in the *COBOL for AIX Programming Guide* for details of the SOSI compiler option.

User-defined words

The following sets of user-defined words are supported. The second column indicates whether multibyte characters are allowed in words of a given set.

User-defined word	Multibyte characters allowed?
Alphabet-name	Yes
Assignment-name	No
Class-name (of data)	Yes
Condition-name	Yes
Data-name	Yes
File-name	Yes
Index-name	Yes
Level-numbers: 01–49, 66, 77, 88	No
Library-name	No
Mnemonic-name	Yes
Object-oriented class-name	No
Paragraph-name	Yes
Priority-numbers: 00–99	No
Program-name	No
Record-name	Yes
Section-name	Yes
Symbolic-character	Yes
Text-name	No

The maximum length of a user-defined word is 30 bytes, except for level-numbers and priority-numbers. Level-numbers and priority numbers must each be a one-digit or two-digit integer.

A given user-defined word can belong to only one of these sets, except that a given number can be both a priority-number and a level-number. Each user-defined word within a set must be unique, except for priority-numbers and level-numbers and except as specified in Chapter 8, “Referencing data names, copy libraries, and procedure division names,” on page 59.

The following types of user-defined words can be referenced by statements and entries in the program in which the user-defined word is declared:

- Paragraph-name
- Section-name

The following types of user-defined words can be referenced by any COBOL program, provided that the compiling system supports the associated library or other system and that the entities referenced are known to that system:

- Library-name
- Text-name

The following types of names, when they are declared within a configuration section, can be referenced by statements and entries in the program that contains the configuration section or in any program contained within that program:

- Alphabet-name
- Class-name
- Condition-name
- Mnemonic-name
- Symbolic-character

The function of each user-defined word is described in the clause or statement in which it appears.

System-names

A *system-name* is a character string that has a specific meaning to the system. There are three types of system-names:

- Computer-name
- Language-name
- Implementor-name

There are types of implementer-names:

- Environment-name
- External-class-name
- Assignment-name

The meaning of each system-name is described with the format in which it appears.

Multibyte character-strings are allowed for system-names.

Function-names

A *function-name* specifies the mechanism provided to determine the value of an intrinsic function. The same word, in a different context, can appear in a program as a user-defined word or a system-name. For a list of function-names and their definitions, see Table 58 on page 497.

Reserved words

A *reserved word* is a character-string with a predefined meaning in a COBOL source unit. Reserved words are listed in Appendix F, “Reserved words,” on page 595

There are six types of reserved words:

- Keywords

- Optional words
- Figurative constants
- Special character words
- Special object identifiers
- Special registers

Keywords

Keywords are reserved words that are required within a given clause, entry, or statement. Within each format, such words appear in uppercase on the main path.

Optional words

Optional words are reserved words that can be included in the format of a clause, entry, or statement in order to improve readability. They have no effect on the execution of the program.

Figurative constants

See “Figurative constants.”

Special character words

There are two types of *special character words*, which are recognized as special characters only when represented in single-byte characters:

- **Arithmetic operators:** + - / * **
See “Arithmetic expressions” on page 257.
- **Relational operators:** <>=<=>=
See “Conditional expressions” on page 262.

Special object identifiers

COBOL provides two special object identifiers, SELF and SUPER:

SELF A special object identifier that you can use in the procedure division of a method. SELF refers to the object instance used to invoke the currently executing method. You can specify SELF only in places that are explicitly listed in the syntax diagrams.

SUPER

A special object identifier that you can use in the procedure division of a method only as the object identifier in an INVOKE statement. When used in this way, SUPER refers to the object instance used to invoke the currently executing method. The resolution of the method to be invoked ignores any methods declared in the class definition of the currently executing method and methods defined in any class derived from that class. Thus, the method invoked is inherited from an ancestor class.

Special registers

See “Special registers” on page 16.

Figurative constants

Figurative constants are reserved words that name and refer to specific constant values. The reserved words for figurative constants and their meanings are:

ZERO, ZEROS, ZEROES

Represents the numeric value zero (0) or one or more occurrences of the alphanumeric character zero, depending on context.

When the figurative constant ZERO, ZEROS, or ZEROES is used in a context that requires an alphanumeric character, an alphanumeric character zero is used. When the context requires a national character zero, a national character zero is used (value NX'0030'). When the context cannot be determined, an alphanumeric character zero is used.

SPACE, SPACES

Represents one or more blanks or spaces. SPACE is treated as an alphanumeric literal when used in a context that requires an alphanumeric character, as a DBCS literal when used in a context that requires a DBCS character, and as a national literal when used in a context that requires a national character.

HIGH-VALUE, HIGH-VALUES

Represents one or more occurrences of the character that has the highest ordinal position in the collating sequence used.

HIGH-VALUE is treated as an alphanumeric literal in a context that requires an alphanumeric character. For alphanumeric data with the EBCDIC collating sequence, the value is X'FF'. For other alphanumeric data, the value depends on the collating sequence indicated by the locale. For more information about locales, see Appendix G, "Locale considerations," on page 609.

HIGH-VALUE is treated as a national literal when used in a context that requires a national literal. The value is national character NX'FFFF'. HIGH-VALUE can be used in a context that requires a national literal only when the NCOLLSEQ(BIN) compiler option is in effect.

When the context cannot be determined, an alphanumeric context is assumed and the value X'FF' is used.

Usage note: You should not use HIGH-VALUE (or a value assigned from HIGH-VALUE) in a way that results in conversion between one data representation and another. X'FF' does not represent a valid EBCDIC or ASCII character, and NX'FFFF' does not represent a valid national character. Conversion of either the alphanumeric or the national HIGH-VALUE representation to another representation results in a substitution character. For example, conversion of X'FF' to UTF-16 would give a substitution character, not NX'FFFF'.

LOW-VALUE, LOW-VALUES

Represents one or more occurrences of the character that has the lowest ordinal position in the collating sequence used.

LOW-VALUE is treated as an alphanumeric literal in a context that requires an alphanumeric character. For alphanumeric data with the EBCDIC collating sequence, the value is X'00'. For other alphanumeric data, the value depends on the collating sequence indicated by the locale. For more information about locales, see Appendix G, "Locale considerations," on page 609.

LOW-VALUE is treated as a national literal when used in a context that requires a national literal. The value is national character NX'0000'. LOW-VALUE can be used in a context that requires a national literal only when the NCOLLSEQ(BIN) compiler option is in effect.

When the context cannot be determined, an alphanumeric context is assumed and the value X'00' is used.

QUOTE, QUOTES

Represents one or more occurrences of:

- The quotation mark character ("), if the QUOTE compiler option is in effect
- The apostrophe character ('), if the APOST compiler option is in effect

QUOTE or QUOTES represents an alphanumeric character when used in a context that requires an alphanumeric character, and represents a national character when used in a context that requires a national character. The national character value of quotation mark is NX'0022'. The national character value of apostrophe is NX'0027'.

QUOTE and QUOTES cannot be used in place of a quotation mark or an apostrophe to enclose an alphanumeric literal.

ALL *literal*

literal can be an alphanumeric literal, a DBCS literal, a national literal, or a figurative constant other than the ALL literal.

When *literal* is not a figurative constant, ALL *literal* represents one or more occurrences of the string of characters that compose the literal.

When *literal* is a figurative constant, the word ALL has no meaning and is used only for readability.

The figurative constant ALL *literal* must not be used with the CALL, INSPECT, INVOKE, STOP, or STRING statements.

symbolic-character

Represents one or more of the characters specified as a value of the *symbolic-character* in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph.

symbolic-character always represents an alphanumeric character; it can be used in a context that requires a national character only when implicit conversion of alphanumeric to national characters is defined. (It can be used, for example, in a MOVE statement where the receiving item is of class national because implicit conversion is defined when the sending item is alphanumeric and the receiving item is national.)

You cannot specify the SYMBOLIC CHARACTERS clause if a multibyte codepage is indicated by the compile-time locale setting. For more information about locales, see Appendix G, "Locale considerations," on page 609.

NULL, NULLS

Represents a value used to indicate that data items defined with USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, USAGE OBJECT REFERENCE, or the ADDRESS OF special register do not contain a valid address. NULL can be used only where explicitly allowed in the syntax formats. NULL has the value zero.

The singular and plural forms of NULL, ZERO, SPACE, HIGH-VALUE, LOW-VALUE, and QUOTE can be used interchangeably. For example, if DATA-NAME-1 is a five-character data item, each of the following statements moves five spaces to DATA-NAME-1:

```
MOVE SPACE      TO DATA-NAME-1
MOVE SPACES     TO DATA-NAME-1
MOVE ALL SPACES TO DATA-NAME-1
```

When the rules of COBOL permit any one spelling of a figurative constant name, any alternative spelling of that figurative constant name can be specified.

You can use a figurative constant wherever *literal* appears in a syntax diagram, except where explicitly prohibited. When a numeric literal appears in a syntax diagram, only the figurative constant ZERO (or ZEROS or ZEROES) can be used. Figurative constants are not allowed as function arguments except in an arithmetic expression, where the expression is an argument to a function.

The length of a figurative constant depends on the context of its use. The following rules apply:

- When a figurative constant is specified in a VALUE clause or associated with a data item (for example, when it is moved to or compared with another item), the length of the figurative constant character-string is equal to 1 or the number of character positions in the associated data item, whichever is greater.
- When a figurative constant, other than the ALL literal, is not associated with another data item (for example, in a CALL, INVOKE, STOP, STRING, or UNSTRING statement), the length of the character-string is one character.

Special registers

Special registers are reserved words that name storage areas generated by the compiler. Their primary use is to store information produced through specific COBOL features. Each such storage area has a fixed name, and must not be defined within the program.

For programs with the RECURSIVE attribute, for programs compiled with the THREAD option, and for methods, storage for the following special registers is allocated on a per-invocation basis:

- ADDRESS-OF
- RETURN-CODE
- SORT-CONTROL
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MESSAGE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY
- XML-CODE
- XML-EVENT
- XML-NTEXT
- XML-TEXT

For the first call to a program after a cancel of that program, or for a method invocation, the compiler initializes the special register fields to their initial values.

For the following four cases:

- Programs that have the INITIAL clause specified
- Programs that have the RECURSIVE clause specified
- Programs compiled with the THREAD option
- Methods

the following special registers are reset to their initial value on each program or method entry:

- RETURN-CODE
- SORT-CONTROL
- SORT-CORE-SIZE
- SORT-FILE-SIZE
- SORT-MESSAGE
- SORT-MODE-SIZE
- SORT-RETURN
- TALLY
- XML-CODE
- XML-EVENT

Further, in the above four cases, values set in ADDRESS OF special registers persist only for the span of the particular program or method invocation.

In all other cases, the special registers will not be reset; they will be unchanged from the value contained on the previous CALL or INVOKE.

Unless otherwise explicitly restricted, a special register can be used wherever a data-name or identifier that has the same definition as the implicit definition of the special register can be used. Implicit definitions, if applicable, are given in the specification of each special register.

You can specify an alphanumeric special register in a function wherever an alphanumeric argument to a function is allowed, unless specifically prohibited.

If qualification is allowed, special registers can be qualified as necessary to provide uniqueness. (For more information, see “Qualification” on page 59.)

ADDRESS OF

The ADDRESS OF special register references the address of a data item in the linkage section, the local-storage section, or the working-storage section.

For 01 and 77 level items in the linkage section, the ADDRESS OF special register can be used as either a sending item or a receiving item. For all other operands, the ADDRESS OF special register can be used only as a sending item.

The ADDRESS OF special register is implicitly defined as USAGE POINTER.

A function-identifier is not allowed as the operand of the ADDRESS OF special register.

DEBUG-ITEM

The DEBUG-ITEM special register provides information for a debugging declarative procedure about the conditions that cause debugging section execution.

DEBUG-ITEM has the following implicit description:

```
01  DEBUG-ITEM.  
    02  DEBUG-LINE      PICTURE IS X(6).  
    02  FILLER          PICTURE IS X VALUE SPACE.
```

```

02  DEBUG-NAME      PICTURE IS X(30).
02  FILLER          PICTURE IS X  VALUE SPACE.
02  DEBUG-SUB-1     PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
02  FILLER          PICTURE IS X  VALUE SPACE.
02  DEBUG-SUB-2     PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
02  FILLER          PICTURE IS X  VALUE SPACE.
02  DEBUG-SUB-3     PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
02  FILLER          PICTURE IS X  VALUE SPACE.
02  DEBUG-CONTENTS  PICTURE IS X(n).

```

Before each debugging section is executed, DEBUG-ITEM is filled with spaces. The contents of the DEBUG-ITEM subfields are updated according to the rules for the MOVE statement, with one exception: DEBUG-CONTENTS is updated as if the move were an alphanumeric-to-alphanumeric elementary move without conversion of data from one form of internal representation to another.

After updating, the contents of the DEBUG-ITEM subfields are:

DEBUG-LINE

The source-statement sequence number (or the compiler-generated sequence number, depending on the compiler option chosen) that caused execution of the debugging section.

DEBUG-NAME

The first 30 characters of the name that caused execution of the debugging section. Any qualifiers are separated by the word 'OF'.

DEBUG-SUB-1, DEBUG-SUB-2, DEBUG-SUB-3

Always set to spaces. These subfields are documented for compatibility with previous COBOL products.

DEBUG-CONTENTS

Data is moved into DEBUG-CONTENTS, as shown in the following table.

Table 2. DEBUG-ITEM subfield contents

Cause of debugging section execution	Statement referred to in DEBUG-LINE	Contents of DEBUG-NAME	Contents of DEBUG-CONTENTS
<i>procedure-name-1</i> ALTER reference	ALTER statement	<i>procedure-name-1</i>	<i>procedure-name-n</i> in TO PROCEED TO phrase
GO TO <i>procedure-name-n</i>	GO TO statement	<i>procedure-name-n</i>	Spaces
<i>procedure-name-n</i> in SORT or MERGE input/output procedure	SORT or MERGE statement	<i>procedure-name-n</i>	"SORT INPUT", "SORT OUTPUT", or "MERGE OUTPUT" (as applicable)
PERFORM statement transfer of control	This PERFORM statement	<i>procedure-name-n</i>	"PERFORM LOOP"
<i>procedure-name-n</i> in a USE procedure	Statement causing USE procedure execution	<i>procedure-name-n</i>	"USE PROCEDURE"
Implicit transfer from a previous sequential procedure	Previous statement executed in previous sequential procedure ¹	<i>procedure-name-n</i>	"FALL THROUGH"
First execution of first nondeclarative procedure	Line number of first nondeclarative procedure-name	Name of first nondeclarative procedure	"START PROGRAM"
1. If this procedure is preceded by a section header, and control is passed through the section header, the statement number refers to the section header.			

JNIENVPTR

The JNIENVPTR special register references the Java Native Interface (JNI) environment pointer. The JNI environment pointer is used in calling Java callable services.

JNIENVPTR is implicitly defined as USAGE POINTER, and cannot be specified as a receiving data item.

For information about using JNIENVPTR and JNI callable services, see *Accessing JNI services* in the *COBOL for AIX Programming Guide*.

LENGTH OF

The LENGTH OF special register contains the number of bytes used by a data item.

LENGTH OF creates an implicit special register that contains the current byte length of the data item referenced by the identifier.

For data items described with usage DISPLAY-1 (DBCS data items) and data items described with usage NATIONAL, each character occupies 2 bytes of storage.

LENGTH OF can be used in the procedure division anywhere a numeric data item that has the same definition as the implied definition of the LENGTH OF special register can be used.

When the ADDR(32) compiler option is specified the LENGTH OF special register has the implicit definition:

USAGE IS BINARY PICTURE 9(9).

When the ADDR(64) compiler option is specified the LENGTH OF special register has the implicit definition:

USAGE IS BINARY PICTURE 9(18).

If the data item referenced by the identifier contains the GLOBAL clause, the LENGTH OF special register is a global data item.

The LENGTH OF special register can appear within either the starting character position or the length expressions of a reference-modification specification. However, the LENGTH OF special register cannot be applied to any operand that is reference-modified.

The LENGTH OF operand cannot be a function, but the LENGTH OF special register is allowed in a function where an integer argument is allowed.

If the LENGTH OF special register is used as the argument to the LENGTH function, the result is always 4 when the ADDR(32) compiler option is specified or always 8 when the ADDR(64) compiler option is specified, independent of the argument specified for LENGTH OF.

If the ADDRESS OF special register is used as the argument to the LENGTH function, the result is always 4 when the ADDR(32) compiler option is specified or always 8 when the ADDR(64) compiler option is specified, independent of the argument specified for ADDRESS OF.

LENGTH OF cannot be either of the following:

- A receiving data item
- A subscript

When the LENGTH OF special register is used as a parameter on a CALL statement, it must be passed BY CONTENT or BY VALUE.

When a table element is specified, the LENGTH OF special register contains the length in bytes of one occurrence. When referring to a table element, the element name need not be subscripted.

A value is returned for any identifier whose length can be determined, even if the area referenced by the identifier is currently not available to the program.

A separate LENGTH OF special register exists for each identifier referenced with the LENGTH OF phrase. For example:

```
MOVE LENGTH OF A TO B
DISPLAY LENGTH OF A, A
ADD LENGTH OF A TO B
CALL "PROGX" USING BY REFERENCE A BY CONTENT LENGTH OF A
```

The intrinsic function LENGTH can also be used to obtain the length of a data item. For data items of usage NATIONAL, the length returned by the LENGTH function is the number of national character positions, rather than bytes; thus the LENGTH OF special register and the LENGTH intrinsic function have different results for data items of usage NATIONAL. For all other data items, the result is the same.

LINAGE-COUNTER

A separate LINAGE-COUNTER special register is generated for each FD entry that contains a LINAGE clause. When more than one is generated, you must qualify each reference to a LINAGE-COUNTER with its related file-name.

The implicit description of the LINAGE-COUNTER special register is one of the following:

- If the LINAGE clause specifies a data-name, LINAGE-COUNTER has the same PICTURE and USAGE as that data-name.
- If the LINAGE clause specifies an integer, LINAGE-COUNTER is a binary item with the same number of digits as that integer.

For more information, see "LINAGE clause" on page 178.

The value in LINAGE-COUNTER at any given time is the line number at which the device is positioned within the current page. LINAGE-COUNTER can be referred to in procedure division statements; it must not be modified by them.

LINAGE-COUNTER is initialized to 1 when an OPEN statement for its associated file is executed.

LINAGE-COUNTER is automatically modified by any WRITE statement for this file. (See "WRITE statement" on page 462.)

If the file description entry for a sequential file contains the LINAGE clause and the EXTERNAL clause, the LINAGE-COUNTER data item is an external data item.

If the file description entry for a sequential file contains the LINAGE clause and the GLOBAL clause, the LINAGE-COUNTER data item is a global data item.

You can specify the LINAGE-COUNTER special register wherever an integer argument to a function is allowed.

RETURN-CODE

The RETURN-CODE special register can be used to pass a return code to the calling program or operating system when the current COBOL program ends. When a COBOL program ends:

- If control returns to the operating system, the value of the RETURN-CODE special register is passed to the operating system as a user return code. The supported user return code values are determined by the operating system, and might not include the full range of RETURN-CODE special register values. For information about user return code values under AIX, see *Getting feedback from date and time callable services* in the *COBOL for AIX Programming Guide*.
- If control returns to a calling program, the value of the RETURN-CODE special register is passed to the calling program. If the calling program is a COBOL program, the RETURN-CODE special register in the calling program is set to the value of the RETURN-CODE special register in the called program.

The RETURN-CODE special register has the implicit definition:

```
01 RETURN-CODE GLOBAL PICTURE S9(4) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the GLOBAL clause in the outermost program.

The following are examples of how to set the RETURN-CODE special register:

- COMPUTE RETURN-CODE = 8.
- MOVE 8 to RETURN-CODE.

The RETURN-CODE special register does not return a value from an invoked method or from a program that uses CALL ... RETURNING. For more information, see "INVOKE statement" on page 366 or "CALL statement" on page 314.

You can specify the RETURN-CODE special register in a function wherever an integer argument is allowed.

The RETURN-CODE special register does not return information from a date/time callable service. For more information, see *Manipulating dates and time* in the *COBOL for AIX Programming Guide*.

SHIFT-OUT and SHIFT-IN

The SHIFT-OUT and SHIFT-IN special registers are supported only when compiling with the CHAR(EBCDIC) compiler option. However, their values are not recognized as delimiters for double-byte characters in the code pages supported for COBOL for AIX.

The SHIFT-OUT and SHIFT-IN special registers are implicitly defined as alphanumeric data items of the format:

```
01 SHIFT-OUT GLOBAL PICTURE X(1) USAGE DISPLAY VALUE X"0E".  
01 SHIFT-IN GLOBAL PICTURE X(1) USAGE DISPLAY VALUE X"0F".
```

When used in nested programs, these special registers are implicitly defined with the global attribute in the outermost program.

These special registers represent EBCDIC shift-out and shift-in control characters, which are unprintable characters.

You can specify the SHIFT-OUT and SHIFT-IN special registers in a function wherever an alphanumeric argument is allowed.

These special registers cannot be receiving items. SHIFT-OUT and SHIFT-IN cannot be used in place of the keyboard control characters when you are defining multibyte user-defined words or specifying EBCDIC DBCS literals.

SORT-CONTROL

The SORT-CONTROL special register is the name of an alphanumeric data item that is implicitly defined as:

```
01 SORT-CONTROL GLOBAL PICTURE X(160) VALUE "file name".
```

where "file name" is the file name used by the sort as the source for additional sort/merge options.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

You can specify the SORT-CONTROL special register in a function wherever an alphanumeric argument is allowed.

The SORT-CONTROL special register is not necessary for a successful sorting or merging operation.

The sort control file takes precedence over the SORT special registers.

SORT-CORE-SIZE

The SORT-CORE-SIZE special register is the name of a binary data item that you can use to specify the number of bytes of storage available to the sort utility. It has the implicit definition:

```
01 SORT-CORE-SIZE GLOBAL PICTURE S9(8) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The amount of storage indicated in the SORT-CORE-SIZE special register does not include memory areas required by COBOL library functions not related to the SORT or MERGE function. It also does not include the fixed amount of memory areas (modules, control blocks, fixed-size work areas) required for the sort and merge implementation.

You can specify the SORT-CORE-SIZE special register in a function wherever an integer argument is allowed.

SORT-FILE-SIZE

The SORT-FILE-SIZE special register is the name of a binary data item that you can use to specify the estimated number of records in the sort input file, *file-name-1*. It has the implicit definition:

```
01 SORT-FILE-SIZE GLOBAL PICTURE S9(8) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

References to the SORT-FILE-SIZE special register are resolved by the compiler; however, the value in the special register has no effect on the execution of a SORT or MERGE statement.

You can specify the SORT-FILE-SIZE special register in a function wherever an integer argument is allowed.

SORT-MESSAGE

The SORT-MESSAGE special register is the name of an alphanumeric data item that is available to both sort and merge programs.

References to the SORT-MESSAGE special register are resolved by the compiler; however, the value in the special register has no effect on the execution of a SORT or MERGE statement.

The SORT-MESSAGE special register has the implicit definition:

```
01 SORT-MESSAGE GLOBAL PICTURE X(8) USAGE DISPLAY VALUE "SYSOUT".
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

You can specify the SORT-MESSAGE special register in a function wherever an alphanumeric argument is allowed.

SORT-MODE-SIZE

The SORT-MODE-SIZE special register is the name of a binary data item that you can use to specify the length of variable-length records that occur most frequently. It has the implicit definition:

```
01 SORT-MODE-SIZE GLOBAL PICTURE S9(5) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

References to the SORT-MODE-SIZE special register are resolved by the compiler; however, the value in the special register has no effect on the execution of a SORT or MERGE statement.

You can specify the SORT-MODE-SIZE special register in a function wherever an integer argument is allowed.

SORT-RETURN

The SORT-RETURN special register is the name of a binary data item and is available to both sort and merge programs.

The SORT-RETURN special register has the implicit definition:

```
01 SORT-RETURN GLOBAL PICTURE S9(4) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The SORT-RETURN special register contains a return code of 0 (successful) or 16 (unsuccessful) at the completion of a sort or merge operation. If the sort or merge is unsuccessful and there is no reference to this special register anywhere in the program, a message is displayed on the terminal.

You can set the SORT-RETURN special register to 16 in an error declarative or input/output procedure to terminate a sort or merge operation before all records are processed. The operation is terminated on the next input or output function for the sort or merge operation.

You can specify the SORT-RETURN special register in a function wherever an integer argument is allowed.

TALLY

The TALLY special register is the name of a binary data item that has the following definition:

```
01 TALLY GLOBAL PICTURE 9(5) USAGE BINARY VALUE ZERO.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

You can refer to or modify the contents of TALLY.

You can specify the TALLY special register in a function wherever an integer argument is allowed.

WHEN-COMPILED

The WHEN-COMPILED special register contains the date at the start of the compilation. WHEN-COMPILED is an alphanumeric data item that has the implicit definition:

```
01 WHEN-COMPILED GLOBAL PICTURE X(16) USAGE DISPLAY.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The WHEN-COMPILED special register has the format:

```
MM/DD/YYhh.mm.ss (MONTH/DAY/YEARhour.minute.second)
```

For example, if compilation began at 2:04 PM on 15 October 2007, WHEN-COMPILED would contain the value 10/15/0714.04.00.

WHEN-COMPILED can be used only as the sending field in a MOVE statement.

WHEN-COMPILED special register data cannot be reference-modified.

The compilation date and time can also be accessed with the intrinsic function WHEN-COMPILED (see “WHEN-COMPILED” on page 531). That function supports four-digit year values and provides additional information.

XML-CODE

The XML-CODE special register is used for the following purposes:

- To communicate status between the XML parser and the processing procedure that was identified in an XML PARSE statement
- To indicate either that an XML GENERATE statement executed successfully or that an exception occurred during XML generation

The XML-CODE special register has the implicit definition:

```
01 XML-CODE PICTURE S9(9) USAGE BINARY VALUE 0.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

When the XML parser encounters an XML event, it sets XML-CODE and then passes control to the processing procedure. For all events except an EXCEPTION event, XML-CODE contains zero when the processing procedure receives control.

For an EXCEPTION event, the parser sets XML-CODE to an exception code that indicates the nature of the exception. XML PARSE exception codes are discussed in *Handling XML PARSE exceptions* in the *COBOL for AIX Programming Guide*.

You can set XML-CODE before returning to the parser, as follows:

- To -1, after a normal event, to indicate that the parser is to terminate immediately without processing any remaining XML document text, and without causing an EXCEPTION event.
- To 0, after an EXCEPTION event for which continuation is allowed, to indicate that the parser is to continue processing. The parser attempts to continue processing the XML document, but results are undefined.
- You can set XML-CODE to a code page identifier after an encoding conflict exception in some cases. See *Handling XML PARSE exceptions* in the *COBOL for AIX Programming Guide* for details.

If you set XML-CODE to any other value before returning to the parser, results are undefined.

When the parser returns control to the XML PARSE statement, XML-CODE contains the most recent value set by the processing procedure or the parser. In some cases, the parser overrides the value set by the processing procedure.

At termination of an XML GENERATE statement, XML-CODE contains either zero, indicating successful completion of XML generation, or a nonzero error code, indicating that an exception occurred during XML generation. XML GENERATE exception codes are detailed in *XML GENERATE exceptions* in the *COBOL for AIX Programming Guide*.

XML-EVENT

The XML-EVENT special register is used to communicate event information from the XML parser to the processing procedure that was identified in the XML PARSE statement. Prior to passing control to the processing procedure, the XML parser sets the XML-EVENT special register to the name of the XML event, as described in Table 3.

XML-EVENT has the implicit definition:

```
01 XML-EVENT USAGE DISPLAY PICTURE X(30) VALUE SPACE.
```

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The content of XML-EVENT is encoded in the EBCDIC or ASCII code page in effect at run time, depending on the setting of the CHAR compiler option (EBCDIC, NATIVE, or S390).

XML-EVENT cannot be used as a receiving data item.

Table 3. Contents of XML-EVENT and XML-TEXT or XML-NTEXT special registers

XML event (content of XML-EVENT)	Content of XML-TEXT or XML-NTEXT
ATTRIBUTE-CHARACTER	The single character that corresponds with the predefined entity reference in the attribute value
ATTRIBUTE-CHARACTERS	The value within quotation marks or apostrophes. This can be a substring of the attribute value if the value includes an entity reference.
ATTRIBUTE-NAME	The attribute name; the string to the left of the equal sign
ATTRIBUTE-NATIONAL-CHARACTER	Regardless of the type of the XML document specified by <i>identifier-1</i> in the XML PARSE statement, XML-TEXT is empty with length zero and XML-NTEXT contains the single national character that corresponds with the numeric character reference.
COMMENT	The text of the comment between the opening character sequence "<!--" and the closing character sequence "-->"
CONTENT-CHARACTER	The single character that corresponds with the predefined entity reference in the element content
CONTENT-CHARACTERS	The character content of the element between start and end tags. This can be a substring of the character content if the content includes an entity reference or another element.
CONTENT-NATIONAL-CHARACTER	Regardless of the type of the XML document specified by <i>identifier-1</i> in the XML PARSE statement, XML-TEXT is empty with length zero and XML-NTEXT contains the single national character that corresponds with the numeric character reference. ¹
DOCUMENT-TYPE-DECLARATION	The entire document type declaration, including the opening and closing character sequences "<!DOCTYPE" and ">"
ENCODING-DECLARATION	The value, between quotes or apostrophes, of the encoding declaration in the XML declaration
END-OF-CDATA-SECTION	The string "]]>"
END-OF-DOCUMENT	Empty with length zero
END-OF-ELEMENT	The name of the end element tag or empty element tag

Table 3. Contents of XML-EVENT and XML-TEXT or XML-NTEXT special registers (continued)

XML event (content of XML-EVENT)	Content of XML-TEXT or XML-NTEXT
EXCEPTION	The part of the document that was successfully scanned, up to and including the point at which the exception was detected. ² Special register XML-CODE contains the unique error code that identifies the exception.
PROCESSING-INSTRUCTION-DATA	The rest of the processing instruction (after the target name), not including the closing sequence, ">", but including trailing, and not leading, white space characters
PROCESSING-INSTRUCTION-TARGET	The processing instruction target name, which occurs immediately after the processing instruction opening sequence, "<?"
STANDALONE-DECLARATION	The value, between quotation marks or apostrophes ("yes" or "no"), of the stand-alone declaration in the XML declaration
START-OF-CDATA-SECTION	The string "<![CDATA["
START-OF-DOCUMENT	The entire document
START-OF-ELEMENT	The name of the start element tag or empty element tag, also known as the element type
UNKNOWN-REFERENCE-IN-CONTENT	The entity reference name, not including the "&" and ";" delimiters
UNKNOWN-REFERENCE-IN-ATTRIBUTE	The entity reference name, not including the "&" and ";" delimiters
VERSION-INFORMATION	The value, between quotation marks or apostrophes, of the version information in the XML declaration
<ol style="list-style-type: none"> 1. National characters with scalar values greater than 65,535 (NX"FFFF") are represented using two encoding units (a "surrogate pair"). Programmers are responsible for ensuring that operations on the content of XML-NTEXT do not split the pair of encoding units that together form a graphic character, thereby forming invalid data. 2. Exceptions for encoding conflicts are signaled before parsing begins. For these exceptions, XML-TEXT or XML-NTEXT is either zero length or contains only the encoding declaration value from the document. See <i>XML GENERATE Exceptions</i> in the <i>COBOL for AIX Programming Guide</i> for information about XML exception codes. 	

XML-NTEXT

The XML-NTEXT special register is defined during XML parsing to contain document fragments that are represented in usage NATIONAL.

XML-NTEXT is an elementary data item of category national of the length of the contained XML document fragment. The length of XML-NTEXT can vary from 0 through 1,073,741,823 *national character positions*. The maximum byte length is 2,147,483,646.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The parser sets XML-NTEXT to the document fragment associated with an event before transferring control to the processing procedure in these cases:

- When the operand of the XML PARSE statement is a data item of category national
- For the ATTRIBUTE-NATIONAL-CHARACTER event
- For the CONTENT-NATIONAL-CHARACTER event

When XML-NTEXT is set, the XML-TEXT special register has a length of zero. At any given time, only one of the two special registers XML-NTEXT and XML-TEXT has a nonzero length.

Use the LENGTH function to determine the number of national characters that XML-NTEXT contains. Use the LENGTH OF special register to determine the number of bytes, rather than the number of national characters, that XML-NTEXT contains.

XML-NTEXT cannot be used as a receiving item.

XML-TEXT

The XML-TEXT special register is defined during XML parsing to contain document fragments that are represented in usage DISPLAY.

XML-TEXT is an elementary data item of category alphanumeric of the length of the contained XML document fragment. The length of XML-TEXT can vary from 0 through 2,147,483,646 bytes.

There is no equivalent COBOL data description entry.

When used in nested programs, this special register is implicitly defined with the global attribute in the outermost program.

The content of XML-TEXT has the encoding of the source XML document: ASCII or UTF-8 if the CHAR(NATIVE) compiler option is in effect; EBCDIC if the CHAR(EBCDIC) compiler option is in effect.

The parser sets XML-TEXT to the document fragment associated with an event before transferring control to the processing procedure when the operand of the XML PARSE statement is an alphanumeric data item, except for the ATTRIBUTE-NATIONAL-CHARACTER event and the CONTENT-NATIONAL-CHARACTER event.

When XML-TEXT is set, the XML-NTEXT special register has a length of zero. At any given time, only one of the two special registers XML-NTEXT and XML-TEXT has a nonzero length.

Use the LENGTH function or the LENGTH OF special register for XML-TEXT to determine the number of bytes that XML-TEXT contains.

XML-TEXT cannot be used as a receiving item.

Literals

A *literal* is a character-string whose value is specified either by the characters of which it is composed or by the use of a figurative constant. (See “Figurative constants” on page 13.) For descriptions of the different types of literals, see:

- “Alphanumeric literals” on page 29
- “DBCS literals” on page 32
- “National literals” on page 34
- “Numeric literals” on page 31

Alphanumeric literals

COBOL for AIX provides several formats of alphanumeric literals:

- Format 1: “Basic alphanumeric literals”
- Format 2: “Hexadecimal notation for alphanumeric literals” on page 30
- Format 3: “Null-terminated alphanumeric literals” on page 31

Basic alphanumeric literals

Basic alphanumeric literals can contain only single-byte or multibyte characters.

The following is the format for a basic alphanumeric literal:

Format 1: Basic alphanumeric literals
<i>"single-byte or multibyte characters"</i> <i>'single-byte or multibyte characters'</i>

The enclosing quotation marks or apostrophes are excluded from the literal when the program is compiled.

An embedded quotation mark or apostrophe must be represented by a pair of quotation marks (") or a pair of apostrophes ('), respectively, when it is the character used as the opening delimiter. For example:

```
"THIS ISN""T WRONG"  
'THIS ISN' 'T WRONG'
```

The delimiter character used as the opening delimiter for a literal must be used as the closing delimiter for that literal. For example:

```
'THIS IS RIGHT'  
"THIS IS RIGHT"  
'THIS IS WRONG"
```

You can use apostrophes or quotation marks as the literal delimiters independent of the APOST/QUOTE compiler option.

Any punctuation characters included within an alphanumeric literal are part of the value of the literal.

"single-byte or multibyte characters" can be any characters represented in the code page indicated by the locale in effect at compile time.

The maximum length of an alphanumeric literal is 160 bytes. The minimum length is 1 byte.

Alphanumeric literals are in the alphanumeric data class and category. (Data classes and categories are described in “Classes and categories of data” on page 160.)

See *SOSI* in the *COBOL for AIX Programming Guide* which describes special treatment of values X'1E' and X'1F' in alphanumeric literals, and additional programming restrictions for alphanumeric literals containing these characters when the *SOSI* compiler option is in effect.

Usage note: Use hexadecimal notation to express control characters X'00' through X'1F' within an alphanumeric literal. Results are unpredictable if you specify these control characters in a basic alphanumeric literal.

Hexadecimal notation for alphanumeric literals

Hexadecimal notation can be used for alphanumeric literals. Hexadecimal notation has the following format:

Format 2: Hexadecimal notation for alphanumeric literals
X"hexadecimal-digits" X'hexadecimal-digits'

- X" or X'** The opening delimiter for the hexadecimal notation of an alphanumeric literal.
- " or '** The closing delimiter for the hexadecimal notation of an alphanumeric literal. If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

Hexadecimal digits are characters in the range '0' to '9', 'a' to 'f', and 'A' to 'F', inclusive. Two hexadecimal digits represent one character in a single-byte character set (EBCDIC or ASCII). Four hexadecimal digits represent one character in a DBCS character set. For UTF-8 or EUC, the number of hexadecimal digits that represent a character depends on the byte length of the character. An even number of hexadecimal digits must be specified. The maximum length of a hexadecimal literal is 320 hexadecimal digits.

The continuation rules are the same as those for any alphanumeric literal. The opening delimiter (X" or X') cannot be split across lines.

An alphanumeric literal in hexadecimal notation has data class and category alphanumeric. The compiler converts the hexadecimal notation into the normal characters of an alphanumeric literal. Hexadecimal notation for alphanumeric literals can be used anywhere alphanumeric literals can be used.

However, alphanumeric literals in hexadecimal notation are interpreted as EBCDIC if the CHAR(EBCDIC) compiler option is in effect, whereas basic alphanumeric literals are always interpreted in the (ASCII-based) code page of the current locale. See *CHAR* in the *COBOL for AIX Programming Guide* for more information on the CHAR(EBCDIC) compiler option.

Usage note: Use hexadecimal notation to express control characters X'00' through X'1F' within an alphanumeric literal. Results are unpredictable if you specify these control characters in a basic alphanumeric literal.

See also “Hexadecimal notation for national literals” on page 35.

Null-terminated alphanumeric literals

Alphanumeric literals can be null-terminated, with the following format:

Format 3: Null-terminated alphanumeric literals
<i>Z</i> " <i>mixed-characters</i> " <i>Z</i> ' <i>mixed-characters</i> '

Z" or Z'

The opening delimiter for a null-terminated alphanumeric literal. Both characters of the opening delimiter (Z" or Z') must be on the same source line.

" or ' The closing delimiter for a null-terminated alphanumeric literal.

If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

mixed-characters

Can be any of the following:

- Solely single-byte characters
- Mixed single-byte and multibyte characters
- Solely multibyte characters
- UTF-8 characters
- EUC characters
- DBCS characters

However, you cannot specify the single-byte character with the value X'00'. X'00' is the null character automatically appended to the end of the literal. The content of the literal is otherwise subject to the same rules and restrictions as an alphanumeric literal with multibyte characters (format 1).

The length of the string of characters in the literal content can be 0 to 159 bytes. The actual length of the literal includes the terminating null character, and is a maximum of 160 bytes.

A null-terminated alphanumeric literal has data class and category alphanumeric. It can be used anywhere an alphanumeric literal can be used except that null-terminated literals are not supported in ALL *literal* figurative constants.

The LENGTH intrinsic function, when applied to a null-terminated literal, returns the number of bytes in the literal prior to but not including the terminating null. (The LENGTH special register does not support literal operands.)

Numeric literals

A *numeric literal* is a character-string whose characters are selected from the digits 0 through 9, a sign character (+ or -), and the decimal point. If the literal contains no decimal point, it is an integer. (In this documentation, the word *integer* appearing in a format represents a numeric literal of nonzero value that contains no sign and no decimal point, except when other rules are included with the description of the format.) The following rules apply:

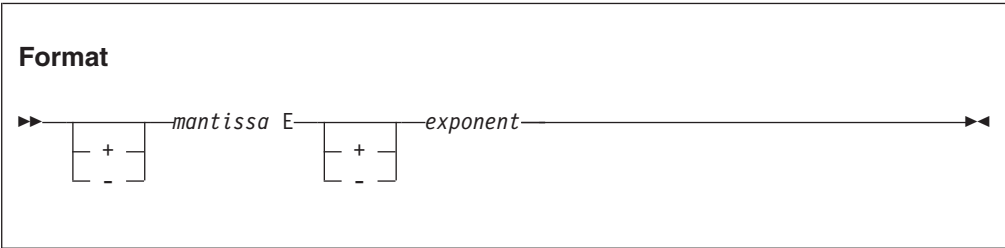
- If the ARITH(COMPAT) compiler option is in effect, one through 18 digits are allowed. If the ARITH(EXTEND) compiler option is in effect, one through 31 digits are allowed.
- Only one sign character is allowed. If included, it must be the leftmost character of the literal. If the literal is unsigned, it is a positive value.
- Only one decimal point is allowed. If a decimal point is included, it is treated as an assumed decimal point (that is, as not taking up a character position in the literal). The decimal point can appear anywhere within the literal except as the rightmost character.

The value of a numeric literal is the algebraic quantity expressed by the characters in the literal. The size of a numeric literal is equal to the number of digits specified by the user.

Numeric literals can be fixed-point or floating-point numbers.

Rules for floating-point literal values

The format and rules for floating-point literals are listed below.



- The sign is optional before the mantissa and the exponent; if you omit the sign, the compiler assumes a positive number.
- The mantissa can contain between one and 16 digits. A decimal point must be included in the mantissa.
- The exponent is represented by an E followed by an optional sign and one or two digits.
- The magnitude of a floating-point literal value must fall between:
 - 32-bit representation: 1.175(10⁻³⁸) to 3.403(10³⁸)
 - 64-bit representation: 2.225(10⁻³⁰⁸) to 1.798(10³⁰⁸)

Numeric literals are in the numeric data class and category. (Data classes and categories are described under “Classes and categories of data” on page 160.)

DBCS literals

The formats and rules for DBCS literals are listed below.

Format for DBCS literals
G'DBCS-characters"
G'DBCS-characters'
N'DBCS-characters"
N'DBCS-characters'

G", G', N", or N'

Opening delimiters.

N" and N' identify a DBCS literal when the NSYMBOL(DBCS) compiler option is in effect. They identify a national literal when the NSYMBOL(NATIONAL) compiler option is in effect, and the rules specified in "National literals" on page 34 apply.

" or ' The closing delimiter. If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

DBCS-characters

Any DBCS character.

Maximum length

The maximum length is limited by the available space on one source line.

Continuation rules

Cannot be continued across lines

DBCS literals with the SOSI compiler option

When the SOSI compiler option is in effect, workstation shift-out (SO) and shift-in (SI) control characters delimit DBCS characters in source text. The following is the syntax for DBCS literals with shift-out and shift-in delimiters:

Format for DBCS literals
G"<DBCS-characters>"
G'<DBCS-characters>'
N"<DBCS-characters>"
N'<DBCS-characters>'

< Represents the shift-out control character (X'1E')

> Represents the shift-in control character (X'1F')

Rules for DBCS characters, literal delimiters, maximum length, and continuation are the same as for DBCS literals without the SOSI compiler option. See *SOSI* in the *COBOL for AIX Programming Guide* for details of the SOSI compiler option.

Where DBCS literals can be used

DBCS literals can be used in the following places:

- Data division
 - In the VALUE clause of data description entries that define a data item of class DBCS.
 - In the VALUE OF clause of file description entries.
- Procedure division
 - In a relation condition when the comparand is a DBCS data item, an elementary data item of class national, a national group item, or an alphanumeric group item
 - As an argument passed BY CONTENT in a CALL statement
 - In the DISPLAY and EVALUATE statements
 - In the following statements:
 - INITIALIZE; for details, see "INITIALIZE statement" on page 353.

- INSPECT; for details, see “INSPECT statement” on page 356.
- MOVE; for details, see “MOVE statement” on page 380.
- STRING; for details, see “STRING statement” on page 445.
- UNSTRING, for details, see “UNSTRING statement” on page 454.
- In figurative constant ALL
- As an argument to the NATIONAL-OF intrinsic function
- Compiler-directing statements COPY, REPLACE, and TITLE

National literals

COBOL for AIX provides the following national literal formats:

- “Basic national literals”
- “Hexadecimal notation for national literals” on page 35

Basic national literals

The following are the format and rules for basic national literals.

Format 1: Basic national literals
<code>N"character-data"</code> <code>N'character-data'</code>

When the NSYMBOL(NATIONAL) compiler option is in effect, the opening delimiter `N"` or `N'` identifies a national literal. A national literal is of the class and category national.

When the NSYMBOL(DBCS) compiler option is in effect, the opening delimiter `N"` or `N'` identifies a DBCS literal, and the rules specified in “DBCS literals” on page 32 apply.

N" or N'

Opening delimiters. The opening delimiter must be coded as single-byte characters. It cannot be split across lines.

- " or '** The closing delimiter. The closing delimiter must be coded as a single-byte character. If a quotation mark is used in the opening delimiter, it must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, it must be used as the closing delimiter.

To include the quotation mark or apostrophe used in the opening delimiter in the content of the literal, specify a pair of quotation marks or apostrophes, respectively. Examples:

```
N'This literal's content includes an apostrophe'
N'This literal includes ", which is not used in the opening delimiter'
N'This literal includes "", which is used in the opening delimiter"
```

character-data

The source text representation of the content of the national literal. *character-data* can include any combination of single-byte and multibyte characters represented in the code page in effect for the source code.

DBCS characters in the content of the literal can be delimited by workstation shift-out and shift-in control characters as described for the SOSI compiler option. See SOSI in the *COBOL for AIX Programming Guide* for details of the SOSI compiler option.

Maximum length

The maximum length of a national literal is 80 character positions, excluding the opening and closing delimiters. If the source content of the literal contains one or more multibyte characters, the maximum length is limited by the available space in Area B of a single source line.

The literal must contain at least one character. Each single-byte character in the literal counts as one character position and each multibyte character in the literal counts as one character position. Workstation shift-in and shift-out delimiters for DBCS characters are not counted.

Continuation rules

When the content of the literal includes multibyte characters, the literal cannot be continued. When the content of the literal does not include multibyte characters, normal continuation rules apply.

The source text representation of *character-data* is automatically converted to UTF-16 for use at run time (for example, when the literal is moved to or compared with a data item of category national).

Hexadecimal notation for national literals

The following are the format and rules for the hexadecimal notation format of national literals.

Format 2: Hexadecimal notation for national literals
<code>NX"hexadecimal-digits"</code> <code>NX'hexadecimal-digits'</code>

The hexadecimal notation format of national literals is not affected by the NSYMBOL compiler option.

NX" or NX'

Opening delimiters. The opening delimiter must be represented in single-byte characters. It must not be split across lines.

" or ' The closing delimiter. The closing delimiter must be represented as a single-byte character.

If a quotation mark is used in the opening delimiter, a quotation mark must be used as the closing delimiter. Similarly, if an apostrophe is used in the opening delimiter, an apostrophe must be used as the closing delimiter.

hexadecimal-digits

Hexadecimal digits in the range '0' to '9', 'a' - 'f', and 'A' to 'F', inclusive. Each group of four hexadecimal digits represents a single national character and must represent a valid code point in UTF-16. The number of hexadecimal digits must be a multiple of four.

Maximum length

The length of a national literal in hexadecimal notation must be from four to 320 hexadecimal digits, excluding the opening and closing delimiters. The length must be a multiple of four.

Continuation rules

Normal continuation rules apply.

The content of a national literal in hexadecimal notation is stored as national characters. The resulting content has the same meaning as a basic national literal that specifies the same national characters.

A national literal in hexadecimal notation has data class and category national and can be used anywhere that a basic national literal can be used.

Where national literals can be used

National literals can be used:

- In a VALUE clause associated with a data item of class national or a VALUE clause associated with a condition-name for a conditional variable that is defined with usage NATIONAL
- In figurative constant ALL
- In a relation condition
- In the WHEN phrase of a format-2 SEARCH statement (binary search)
- In the ALL, LEADING, or FIRST phrase of an INSPECT statement
- In the BEFORE or AFTER phrase of an INSPECT statement
- In the DELIMITED BY phrase of a STRING statement
- In the DELIMITED BY phrase of an UNSTRING statement
- As the method-name in a METHOD-ID paragraph, an END METHOD marker, and an INVOKE statement
- As an argument passed BY CONTENT in the CALL statement
- As an argument passed BY VALUE in an INVOKE or CALL statement
- In the DISPLAY and EVALUATE statements
- As a sending item in the following procedural statements:
 - INITIALIZE
 - INSPECT
 - MOVE
 - STRING
 - UNSTRING
- In the argument list to the following intrinsic functions:
DISPLAY-OF, LENGTH, LOWER-CASE, MAX, MIN, ORD-MAX, ORD-MIN, REVERSE, and UPPER-CASE
- In the compiler-directing statements COPY, REPLACE, and TITLE

A national literal can be used only as specified in the detailed rules in this document.

PICTURE character-strings

A *PICTURE character-string* is composed of the currency symbol and certain combinations of characters in the COBOL character set. PICTURE character-strings are delimited only by the separator space, separator comma, separator semicolon, or separator period.

A chart of PICTURE clause symbols appears in Table 15 on page 200.

Comments

A *comment* is a character-string that can contain any combination of characters from the character set of the computer. It has no effect on the execution of the program. There are two forms of comments:

Comment entry (identification division)

This form is described under “Optional paragraphs” on page 103.

Comment line (any division)

This form is described under “Comment lines” on page 50.

Character-strings that form comments can contain any single-byte or multibyte character from the code page in effect for compilation.

Multiple comment lines that contain multibyte strings are allowed. The embedding of multibyte characters in a comment line must be done on a line-by-line basis. Words containing those characters cannot be continued to a following line. No syntax checking for valid strings is provided in comment lines.

Chapter 4. Separators

A *separator* is a character or a string of two or more contiguous characters that delimits character-strings. The separators are shown in the following table.

Table 4. Separators

Separator	Meaning
<i>b</i> ¹	Space
, <i>b</i> ¹	Comma
. <i>b</i> ¹	Period
; <i>b</i> ¹	Semicolon
(Left parenthesis
)	Right parenthesis
:	Colon
" <i>b</i> ¹	Quotation mark
' <i>b</i> ¹	Apostrophe
X"	Opening delimiter for a hexadecimal format alphanumeric literal
X'	Opening delimiter for a hexadecimal format alphanumeric literal
Z"	Opening delimiter for a null-terminated alphanumeric literal
Z'	Opening delimiter for a null-terminated alphanumeric literal
N"	Opening delimiter for a national literal ²
N'	Opening delimiter for a national literal ²
NX"	Opening delimiter for a hexadecimal format national literal
NX'	Opening delimiter for a hexadecimal format national literal
G"	Opening delimiter for a DBCS literal
G'	Opening delimiter for a DBCS literal
==	Pseudo-text delimiter
<p>1. <i>b</i> represents a blank.</p> <p>2. N" and N' are the opening delimiter for a DBCS literal when the NSYMBOL(DBCS) compiler option is in effect.</p>	

Rules for separators

In the following description, {} (curly braces) enclose each separator, and *b* represents a space. Anywhere a space is used as a separator or as part of a separator, more than one space can be used.

Space {*b*}

A space can immediately precede or follow any separator except:

- The opening pseudo-text delimiter, where the preceding space is required.
- Within quotation marks. Spaces between quotation marks are considered part of the alphanumeric literal; they are not considered separators.

Period {*b*}, Comma {*b*}, Semicolon {*b*}

A separator comma is composed of a comma followed by a space. A separator period is composed of a period followed by a space. A separator semicolon is composed of a semicolon followed by a space.

The separator period must be used only to indicate the end of a sentence, or as shown in formats. The separator comma and separator semicolon can be used anywhere the separator space is used.

- In the identification division, each paragraph must end with a separator period.
- In the environment division, the SOURCE-COMPUTER, OBJECT-COMPUTER, SPECIAL-NAMES, and I-O-CONTROL paragraphs must each end with a separator period. In the FILE-CONTROL paragraph, each file-control entry must end with a separator period.
- In the data division, file (FD), sort/merge file (SD), and data description entries must each end with a separator period.
- In the procedure division, separator commas or separator semicolons can separate statements within a sentence and operands within a statement. Each sentence and each procedure must end with a separator period.

Parentheses { () ... { } }

Except in pseudo-text, parentheses can appear only in balanced pairs of left and right parentheses. They delimit subscripts, a list of function arguments, reference-modifiers, arithmetic expressions, or conditions.

Colon { : }

The colon is a separator and is required when shown in general formats.

Quotation marks { " } ... { " }

An opening quotation mark must be immediately preceded by a space or a left parenthesis. A closing quotation mark must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. Quotation marks must appear as balanced pairs. They delimit alphanumeric literals, except when the literal is continued (see "Continuation lines" on page 48).

Apostrophes { ' } ... { ' }

An opening apostrophe must be immediately preceded by a space or a left parenthesis. A closing apostrophe must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. Apostrophes must appear as balanced pairs. They delimit alphanumeric literals, except when the literal is continued (see "Continuation lines" on page 48).

Null-terminated literal delimiters { Z" } ... { " }, { Z' } ... { ' }

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter.

DBCS literal delimiters { G" } ... { " }, { G' } ... { ' }, { N" } ... { " }, { N' } ... { ' }

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. N" and N' are DBCS literal delimiters when the NSYMBOL(DBCS) compiler option is in effect.

National literal delimiters {N"} ... {"}, {N'} ... {'}, {NX"} ... {"}, {NX'} ... {'}

The opening delimiter must be immediately preceded by a space or a left parenthesis. The closing delimiter must be immediately followed by a separator space, comma, semicolon, period, right parenthesis, or pseudo-text delimiter. N" and N' are DBCS literal delimiters when the NSYMBOL(DBCS) compiler option is in effect.

Pseudo-text delimiters {b==} ... {==b}

An opening pseudo-text delimiter must be immediately preceded by a space. A closing pseudo-text delimiter must be immediately followed by a separator space, comma, semicolon, or period. Pseudo-text delimiters must appear as balanced pairs. They delimit pseudo-text. (See "COPY statement" on page 540.)

Any punctuation character included in a PICTURE character-string, a comment character-string, or an alphanumeric literal is not considered a punctuation character, but is part of the character-string or literal.

Chapter 5. Sections and paragraphs

Sections and paragraphs define a program. Sections and paragraphs are subdivided into sentences, statements, and entries (see “Sentences, statements, and entries”). Sentences are subdivided into statements (see “Statements” on page 44), and statements are subdivided into phrases (see “Phrases” on page 44). Entries are subdivided into clauses (see “Clauses” on page 44) and phrases.

For more information about sections, paragraphs, and statements, see “Procedures” on page 256.

Sentences, statements, and entries

Unless the associated rules explicitly state otherwise, each required clause or statement must be written in the sequence shown in its format. If optional clauses or statements are used, they must be written in the sequence shown in their formats. These rules are true even for clauses and statements treated as comments.

The syntactical hierarchy follows this form:

- Identification division
 - Paragraphs
 - Entries
 - Clauses
- Environment division
 - Sections
 - Paragraphs
 - Entries
 - Clauses
 - Phrases
- Data division
 - Sections
 - Entries
 - Clauses
 - Phrases
- Procedure division
 - Sections
 - Paragraphs
 - Sentences
 - Statements
 - Phrases

Entries

An *entry* is a series of clauses that ends with a separator period. Entries are constructed in the identification, environment, and data divisions.

Clauses

A *clause* is an ordered set of consecutive COBOL character-strings that specifies an attribute of an entry. Clauses are constructed in the identification, environment, and data divisions.

Sentences

A *sentence* is a sequence of one or more statements that ends with a separator period. Sentences are constructed in the procedure division.

Statements

A *statement* specifies an action to be taken by the program. Statements are constructed in the procedure division. For descriptions of the different types of statements, see:

- “Imperative statements” on page 286
- “Conditional statements” on page 288
- Chapter 7, “Scope of names,” on page 53
- Chapter 23, “Compiler-directing statements,” on page 537

Phrases

Each clause or statement in a program can be subdivided into smaller units called *phrases*.

Chapter 6. Reference format

COBOL source text must be written in COBOL *reference format* that can be either *fixed source format*, or *extended source format*. Fixed source format consists of the following areas in a 72-character line. Extended source format consists of the following areas in a 252-character line.

Sequence number area

Columns 1 through 6

Indicator area

Column 7

Area A

Columns 8 through 11

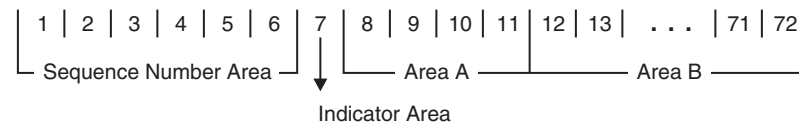
Area B

Columns 12 through 72 in fixed source format

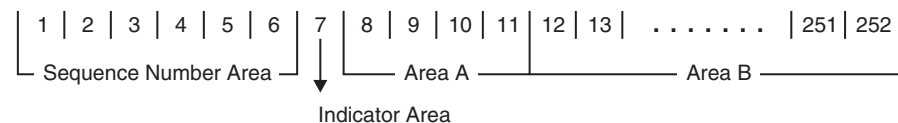
Columns 12 through 252 in extended source format

Remember: In fixed source format and extended source format, lines shorter than the maximum length are extended with spaces to the maximum length.

This figure illustrates fixed source format for a COBOL source line.



This figure illustrates extended source format for a COBOL source line.



For information about how to indicate fixed source format or extended source format, see *SRCFORMAT* in the *COBOL for AIX Programming Guide*.

Sequence number area

The sequence number area can be used to label a source statement line. The content of this area can consist of any character in the character set of the computer.

Indicator area

Use the indicator area to specify:

- The continuation of words or alphanumeric literals from the previous line onto the current line
- The treatment of text as documentation
- Debugging lines

See “Continuation lines” on page 48, “Comment lines” on page 50, and “Debugging lines” on page 52.

The indicator area can be used for source listing formatting. A slash (/) placed in the indicator column causes the compiler to start a new page for the source listing, and the corresponding source record to be treated as a comment. The effect can be dependent on the LINECOUNT compiler option. For information about the LINECOUNT compiler option, see *LINECOUNT* in the *COBOL for AIX Programming Guide*.

Area A

The following items must begin in Area A:

- Division headers
- Section headers
- Paragraph headers or paragraph names
- Level indicators or level-numbers (01 and 77)
- DECLARATIVES and END DECLARATIVES
- End program, end class, and end method markers

Division headers

A division header is a combination of words, followed by a separator period, that indicates the beginning of a division:

- IDENTIFICATION DIVISION.
- ENVIRONMENT DIVISION.
- DATA DIVISION.
- PROCEDURE DIVISION.

A division header (except when a USING phrase is specified with a procedure division header) must be immediately followed by a separator period. Except for the USING phrase, no text can appear on the same line.

Section headers

In the environment and procedure divisions, a section header indicates the beginning of a series of paragraphs. For example:

INPUT-OUTPUT SECTION.

In the data division, a section header indicates the beginning of an entry; for example:

FILE SECTION.

LINKAGE SECTION.

LOCAL-STORAGE SECTION.

WORKING-STORAGE SECTION.

A section header must be immediately followed by a separator period.

Paragraph headers or paragraph names

A paragraph header or paragraph name indicates the beginning of a paragraph.

In the environment division, a paragraph consists of a paragraph header followed by one or more entries. For example:

OBJECT-COMPUTER. *computer-name*.

In the procedure division, a paragraph consists of a paragraph-name followed by one or more sentences.

Level indicators (FD and SD) or level-numbers (01 and 77)

A level indicator can be either FD or SD. It must begin in Area A and be followed by a space. (See “File section” on page 172.) A level-number that must begin in Area A is a one- or two-digit integer with a value of 01 or 77. It must be followed by a space or separator period.

DECLARATIVES and END DECLARATIVES

DECLARATIVES and END DECLARATIVES are keywords that begin and end the declaratives part of the source unit.

In the procedure division, each of the keywords DECLARATIVES and END DECLARATIVES must begin in Area A and be followed immediately by a separator period; no other text can appear on the same line. After the keywords END DECLARATIVES, no text can appear before the following section header. (See “Declaratives” on page 255.)

End program, end class, and end method markers

The end markers are a combination of words followed by a separator period that indicates the end of a COBOL program, method, class, factory, or object definition. For example:

```
END PROGRAM program-name.  
END CLASS class-name.  
END METHOD "method-name".  
END OBJECT.  
END FACTORY.
```

For programs

program-name must be identical to the *program-name* of the corresponding PROGRAM-ID paragraph. Every COBOL program, except an outermost program that contains no nested programs and is not followed by another batch program, must end with an END PROGRAM marker.

For classes

class-name must be identical to the *class-name* in the corresponding CLASS-ID paragraph.

For methods

method-name must be identical to the *method-name* in the corresponding METHOD-ID paragraph.

For object paragraphs

There is no name in an object paragraph header or in its end marker. The syntax is simply END OBJECT.

For factory paragraphs

There is no name in a factory paragraph header or in its end marker. The syntax is simply END FACTORY.

Area B

The following items must begin in Area B:

- Entries, sentences, statements, and clauses
- Continuation lines

Entries, sentences, statements, clauses

The first entry, sentence, statement, or clause begins on either the same line as the header or paragraph-name that it follows, or in Area B of the next nonblank line that is not a comment line. Successive sentences or entries either begin in Area B of the same line as the preceding sentence or entry, or in Area B of the next nonblank line that is not a comment line.

Within an entry or sentence, successive lines in Area B can have the same format or can be indented to clarify program logic. The output listing is indented only if the input statements are indented. Indentation does not affect the meaning of the program. The programmer can choose the amount of indentation, subject only to the restrictions on the width of Area B. See also Chapter 5, "Sections and paragraphs," on page 43.

Continuation lines

Any sentence, entry, clause, or phrase that requires more than one line can be continued in Area B of the next line that is neither a comment line nor a blank line. The line being continued is a *continued line*; the succeeding lines are *continuation lines*. Area A of a continuation line must be blank.

If there is no hyphen (-) in the indicator area (column 7) of a line, the last character of the preceding line is assumed to be followed by a space.

The following cannot be continued:

- Multibyte user-defined words
- DBCS literals
- Alphanumeric literals containing multibyte characters
- National literals containing multibyte characters

However, alphanumeric literals and national literals in hexadecimal notation can be continued regardless of the kind of characters expressed in hexadecimal notation.

All characters that make up an opening literal delimiter must be on the same line. For example, Z", G", N", NX", or X".

Both characters that make up the pseudo-text delimiter separator "==" must be on the same line.

If there is a hyphen in the indicator area of a line, the first nonblank character of the continuation line immediately follows the last nonblank character of the continued line without an intervening space.

Continuation of alphanumeric and national literals

Alphanumeric and national literals can be continued only when there are no multibyte characters in the content of the literal.

The following rules apply to alphanumeric and national literals that do not contain multibyte characters:

- If the continued line contains an alphanumeric or national literal without a closing quotation mark, all spaces at the end of the continued line (through column 72 in fixed source format, or through column 252 in extended source format) are considered to be part of the literal. The continuation line must contain a hyphen in the indicator area, and the first nonblank character must be a quotation mark. The continuation of the literal begins with the character immediately following the quotation mark.
- If an alphanumeric or national literal that is to be continued on the next line has as its last character a quotation mark in column 72 in fixed source format, or in column 252 in extended source format, the continuation line must start with two consecutive quotation marks. This will result in a single quotation mark as part of the value of the literal.

If the last character on the continued line of an alphanumeric or national literal is a single quotation mark in Area B, the continuation line can start with a single quotation mark. This will result in two consecutive literals instead of one continued literal.

The rules are the same when an apostrophe is used instead of a quotation mark in delimiters.

If you want to continue a literal such that the continued lines and the continuation lines are part of one literal:

- Code a hyphen in the indicator area of each continuation line.
- Code the literal value using all columns of each continued line, up to and including column 72 in fixed source format, or column 252 in extended source format. (Do not terminate the continued lines with a single quotation mark followed by a space.)
- Code a quotation mark before the first character of the literal on each continuation line.
- Terminate the last continuation line with a single quotation mark followed by a space.

In the following fixed source format examples, the number and size of literals created are indicated below the example:

```
|...+*..1....+....2....+....3....+....4....+....5....+....6....+....7..
000001      "AAAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDEEEEEEEEE
-           "GGGGGGGGGHHHHHHHHHHIIIIIIIIJJJJJJJJKKKKKKKKK
-           "LLLLLLLLLLMMMMMMMMM"
```

- Literal 000001 is interpreted as one alphanumeric literal that is 120 bytes long. Each character between the starting quotation mark and up to and including column 72 of continued lines is counted as part of the literal.

```
|...+*..1....+....2....+....3....+....4....+....5....+....6....+....7..
000003      N"AAAAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDDEEEEEEEEE
-           "GGGGGGGGG"
```

- Literal 000003 is interpreted as one national literal that is 60 national character positions in length (120 bytes). Each character between the starting quotation

mark and the ending quotation mark on the continued line is counted as part of the literal. Although single-byte characters are entered, the value of the literals is stored as national characters.

```
|...+.*..1....+....2....+....3....+....4....+....5....+....6....+....7..
000005 "AAAAAAAAAABBBBBBBBBBCCCCCCCCDDDDDDDDDEEEEEEEEE
- "GGGGGGGGGGHHHHHHHHHHIIIIIIIIJJJJJJJJKKKKKKKKK
- "LLLLLLLLLLMMMMMMMMM"
```

- Literal 000005 is interpreted as one literal that is 140 bytes long. The blanks at the end of each continued line are counted as part of the literal because the continued lines do not end with a quotation mark.

```
|...+.*..1....+....2....+....3....+....4....+....5....+....6....+....7..
000010 "AAAAAAAAAABBBBBBBBBBCCCCCCCCDDDDDDDDDEEEEEEEEE"
- "GGGGGGGGGGHHHHHHHHHHIIIIIIIIJJJJJJJJKKKKKKKKK"
- "LLLLLLLLLLMMMMMMMMM"
```

- Literal 000010 is interpreted as three separate literals that have lengths of 50, 50, and 20, respectively. The quotation mark with the following space terminates the continued line. Only the characters within the quotation marks are counted as part of the literals. Literal 000010 is not valid as a VALUE clause literal for non-level-88 data items.

To code a continued literal where the length of each continued part of the literal is less than the length of Area B, adjust the starting column such that the last character of the continued part is in column 72 in fixed source format, or in column 252 in extended source format.

Area A or Area B

The following items can begin in either Area A or Area B:

- Level-numbers
- Comment lines
- Floating comment indicator (*>)
- Compiler-directing statements
- Compiler directives
- Debugging lines
- Pseudo-text
- Blank lines

Level-numbers

A level-number that can begin in Area A or B is a one- or two-digit integer with a value of 02 through 49, 66, or 88. A level-number that must begin in Area A is a one- or two-digit integer with a value of 01 or 77. A level-number must be followed by a space or a separator period. For more information, see “Level-numbers” on page 182.

Comment lines

A *comment line* is any line with an asterisk (*) or slash (/) in the indicator area (column 7) of the line, or with a floating comment indicator (*>) as the first character-string in the program text area (Area A plus Area B). The comment can be written anywhere in the program text area of that line, and can consist of any combination of characters from the character set of the computer.

Comment lines can be placed anywhere in a program, method, or class definition. Comment lines placed before the identification division header must follow any control cards (for example, PROCESS or CBL).

Important: Comments intermixed with control cards could nullify some of the control cards and cause them to be diagnosed as errors.

Multiple comment lines are allowed. Each must begin with an asterisk (*) or a slash (/) in the indicator area, or with a floating comment indicator (*>).

For more information about floating comment indicators, see “Floating comment indicator (*>).”

An asterisk (*) comment line is printed on the next available line in the output listing. The effect can be dependent on the LINECOUNT compiler option. For information about the LINECOUNT compiler option, see *LINECOUNT* in the *COBOL for AIX Programming Guide*. A slash (/) comment line is printed on the first line of the next page, and the current page of the output listing is ejected.

The compiler treats a comment line as documentation, and does not check it syntactically.

Floating comment indicator (*>)

In addition to the fixed indicators that can be used in the indicator area of the source reference format, a floating comment indicator (*>) can be used in either fixed source format or extended source format.

A floating comment indicator indicates a comment line if it is the first character string in the program text area (Area A plus Area B), or indicates an inline comment if it is after one or more character strings in the program text area.

These are the rules for floating comment indicators:

- Both characters (*) and (>) that form the multiple-character floating indicator must be contiguous and on the same line.
- A space is implied immediately after a floating comment indicator.
- The floating comment indicator for an inline comment must be preceded by a separator space, and can be specified wherever a separator space can be specified.
- An end-of-line terminates a floating comment.

Compiler-directing statements

Most compiler-directing statements, including COPY and REPLACE, can start in either Area A or Area B.

BASIS, CBL (PROCESS), *CBL (*CONTROL), DELETE, EJECT, INSERT, SKIP1, SKIP2, SKIP3, and TITLE statements can also start in Area A or Area B.

Compiler directives

Compiler directives must start in Area B. Currently, the only compiler directive is CALLINTERFACE. For more information, see Chapter 24, “Compiler directives,” on page 561.

Debugging lines

A *debugging line* is any line with a D (or d) in the indicator area of the line. Debugging lines can be written in the environment division (after the OBJECT-COMPUTER paragraph), the data division, and the procedure division. If a debugging line contains only spaces in Area A and Area B, it is considered a blank line.

See "WITH DEBUGGING MODE" in "SOURCE-COMPUTER paragraph" on page 108.

Pseudo-text

The character-strings and separators that comprise *pseudo-text* can start in either Area A or Area B. If, however, there is a hyphen in the indicator area (column 7) of a line that follows the opening pseudo-text delimiter, Area A of the line must be blank, and the rules for continuation lines apply to the formation of text words. See "Continuation lines" on page 48 for details.

Blank lines

A *blank line* contains nothing but spaces in column 7 through column 72 in fixed source format, or in column 7 through column 252 in extended source format. A blank line can be anywhere in a program.

Source conversion utility (scu)

The source conversion utility (scu) is a stand-alone AIX program that assists in the conversion of COBOL source programs from non-IBM or free-format source formats to a format that can be compiled by COBOL for AIX.

For more information about scu, see Appendix C, "Source conversion utility (scu)," on page 583.

Chapter 7. Scope of names

A user-defined word names a data resource or a COBOL programming element. Examples of named data resources are a file, a data item, or a record. Examples of named programming elements are a program, a paragraph, a method, or a class definition. The sections below define the types of names in COBOL and explain where the names can be referenced:

- “Types of names”
- “External and internal resources” on page 55
- “Resolution of names” on page 56

Types of names

In addition to identifying a resource, a name can have global or local attributes. Some names are always global, some names are always local, and some names are either local or global depending on specifications in the program in which the names are declared.

For programs

A *global name* can be used to refer to the resource with which it is associated both:

- From within the program in which the global name is declared
- From within any other program that is contained in the program that declares the global name

Use the GLOBAL clause in the data description entry to indicate that a name is global. For more information about using the GLOBAL clause, see “GLOBAL clause” on page 173.

A *local name* can be used only to refer to the resource with which it is associated from within the program in which the local name is declared.

By default, if a data-name, a file-name, a record-name, or a condition-name declaration in a data description entry does not include the GLOBAL clause, the name is local.

For methods

All names declared in methods are implicitly local.

For classes

Names declared in a class definition are global to all the methods contained in that class definition.

For object paragraphs

Names declared in the data division of an object paragraph are global to the methods contained in that object paragraph.

For factory paragraphs

Names declared in the data division of a factory paragraph are global to the methods contained in that factory paragraph.

Restriction: Specific rules sometimes prohibit specifying the GLOBAL clause for certain data description, file description, or record description entries.

The following list indicates the names that you can use and whether the name can be local or global:

data-name

data-name assigns a name to a data item.

A data-name is global if the GLOBAL clause is specified either in the data description entry that declares the data-name or in another entry to which that data description entry is subordinate.

file-name

file-name assigns a name to a file connector.

A file-name is global if the GLOBAL clause is specified in the file description entry for that file-name.

record-name

record-name assigns a name to a record.

A record-name is global if the GLOBAL clause is specified in the record description that declares the record-name, or in the case of record description entries in the file section, if the GLOBAL clause is specified in the file description entry for the file name associated with the record description entry.

condition-name

condition-name associates a value with a conditional variable.

A condition-name that is declared in a data description entry is global if that entry is subordinate to another entry that specifies the GLOBAL clause.

A condition-name that is declared within the configuration section is always global.

program-name

program-name assigns a name to an external or internal (nested) program. For more information, see “Conventions for program-names” on page 84.

A program-name is neither local nor global. For more information, see “Conventions for program-names” on page 84.

method-name

method-name assigns a name to a method. *method-name* must be specified as the content of an alphanumeric literal or a national literal.

section-name

section-name assigns a name to a section in the procedure division.

A section-name is always local.

paragraph-name

paragraph-name assigns a name to a paragraph in the procedure division.

A paragraph-name is always local.

basis-name

basis-name specifies the name of source text that is to be included by the compiler into the source unit. For details, see “BASIS statement” on page 537.

library-name

library-name specifies the COBOL library that the compiler uses for including COPY text. For details, see “COPY statement” on page 540.

text-name

text-name specifies the name of COPY text to be included by the compiler into the source unit. For details, see “COPY statement” on page 540.

alphabet-name

alphabet-name assigns a name to a specific character set or collating sequence, or both, in the SPECIAL-NAMES paragraph of the environment division.

An alphabet-name is always global.

class-name (of data)

class-name assigns a name to the proposition in the SPECIAL-NAMES paragraph of the environment division for which a truth value can be defined.

A class-name is always global.

class-name (object-oriented)

class-name assigns a name to an object-oriented class or subclass.

mnemonic-name

mnemonic-name assigns a user-defined word to an implementer-name.

A mnemonic-name is always global.

symbolic-character

symbolic-character specifies a user-defined figurative constant.

A symbolic-character is always global.

index-name

index-name assigns a name to an index associated with a specific table.

If a data item that possesses the global attribute includes a table accessed with an index, that index also possesses the global attribute. In addition, the scope of that index-name is identical to the scope of the data-name that includes the table.

External and internal resources

The storage associated with a data item or a file connector can be *external* or *internal* to the program or method in which the resource is declared.

A data item or file connector is external if the storage associated with that resource is associated with the run unit rather than with any particular program or method within the run unit. An external resource can be referenced by any program or method in the run unit that describes the resource. References to an external resource from different programs or methods using separate descriptions of the resource are always to the same resource. In a run unit, there is only one representation of an external resource.

A resource is internal if the storage associated with that resource is associated only with the program or method that describes the resource.

External and internal resources can have either global or local names.

A data record described in the working-storage section is given the external attribute by the presence of the EXTERNAL clause in its data description entry. Any data item described by a data description entry subordinate to an entry that describes an external record also attains the external attribute. If a record or data

item does not have the external attribute, it is part of the internal data of the program or method in which it is described.

Two programs or methods in a run unit can reference the same file connector in the following circumstances:

- An external file connector can be referenced from any program or method that describes that file connector.
- If a program is contained within another program, both programs can refer to a global file connector by referring to an associated global file-name either in the containing program or in any program that directly or indirectly contains the containing program.

Two programs or methods in a run unit can reference common data in the following circumstances:

- The data content of an external data record can be referenced from any program or method provided that program or method has described that data record.
- If a program is contained within another program, both programs can refer to data that possesses the global attribute either in the program or in any program that directly or indirectly contains the containing program.

The data records described as subordinate to a file description entry that does not contain the EXTERNAL clause or to a sort-merge file description entry, as well as any data items described subordinate to the data description entries for such records, are always internal to the program or method that describes the file-name. If the EXTERNAL clause is included in the file description entry, the data records and the data items attain the external attribute.

Resolution of names

The rules for resolution of names depend on whether the names are specified in a program or in a class definition.

Names within programs

When a program, program B, is directly contained within another program, program A, both programs can define a condition-name, a data-name, a file-name, or a record-name using the same user-defined word. When such a duplicated name is referenced in program B, the following steps determine the referenced resource (these rules also apply to classes and contained methods):

1. The referenced resource is identified from the set of all names that are defined in program B and all global names defined in program A and in any programs that directly or indirectly contain program A. The normal rules for qualification and any other rules for uniqueness of reference are applied to this set of names until one or more resources is identified.
2. If only one resource is identified, it is the referenced resource.
3. If more than one resource is identified, no more than one resource can have a name local to program B. If zero or one of the resources has a name local to program B, the following rules apply:
 - If the name is declared in program B, the resource in program B is the referenced resource.
 - If the name is not declared in program B, the referenced resource is:
 - The resource in program A if the name is declared in program A

- The resource in the containing program if the name is declared in the program that contains program A

This rule is applied to further containing programs until a valid resource is found.

Names within a class definition

Within a class definition, resources can be defined within the following units:

- The factory data division
- The object data division
- A method data division

If a resource is defined with a given name in the data division of an object definition, and there is no resource defined with the same name in an instance method of that object definition, a reference to that name from an instance method is a reference to the resource in the object data division.

If a resource is defined with a given name in the data division of a factory definition, and there is no resource defined with the same name in a factory method of that factory definition, a reference to that name from a factory method is a reference to the resource in the factory data division.

If a resource is defined within a method, any reference within the method to that resource name is always a reference to the resource in the method.

The normal rules for qualification and uniqueness of reference apply when the same name is associated with more than one resource within a given method data division, object data division, or factory data division.

Chapter 8. Referencing data names, copy libraries, and procedure division names

References can be made to external and internal resources. References to data and procedures can be either explicit or implicit. The following sections:

- “Uniqueness of reference”
- “Data attribute specification” on page 72

contain the rules for qualification and for explicit and implicit data references.

Uniqueness of reference

Every user-defined name in a COBOL program is assigned by the user to name a resource for solving a data processing problem. To use a resource, a statement in a COBOL program must contain a reference that uniquely identifies that resource.

To ensure uniqueness of reference, a user-defined name can be qualified. A subscript is required for unique reference to a table element, except as specified in “Subscripting” on page 65. A data-name or function-name, any subscripts, and the specified reference-modifier uniquely reference a data item defined by reference modification.

When the same name has been assigned in separate programs to two or more occurrences of a resource of a given type, and when qualification by itself does not allow the references in one of those programs to differentiate between the identically named resources, then certain conventions that limit the scope of names apply. The conventions ensure that the resource identified is that described in the program containing the reference. For more information about resolving program-names, see “Resolution of names” on page 56.

Unless otherwise specified by the rules for a statement, any subscripts and reference modification are evaluated only once as the first step in executing that statement.

Qualification

A name that exists within a hierarchy of names can be made unique by specifying one or more higher-level names in the hierarchy. The higher-level names are called *qualifiers*, and the process by which such names are made unique is called *qualification*.

Qualification is specified by placing one or more phrases after a user-specified name, with each phrase made up of the word IN or OF followed by a qualifier. (IN and OF are logically equivalent.)

In any hierarchy, the data-name associated with the highest level must be unique if it is referenced, and cannot be qualified.

You must specify enough qualification to make the name unique; however, it is not always necessary to specify all the levels of the hierarchy. For example, if there is more than one file whose records contain the field EMPLOYEE-NO, but only one of the files has a record named MASTER-RECORD:

- EMPLOYEE-NO OF MASTER-RECORD sufficiently qualifies EMPLOYEE-NO.
- EMPLOYEE-NO OF MASTER-RECORD OF MASTER-FILE is valid but unnecessary.

Qualification rules

The rules for qualifying a name are:

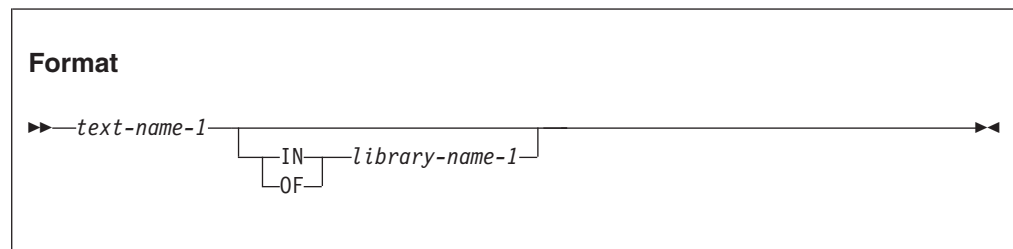
- A name can be qualified even though it does not need qualification except in a REDEFINES clause, in which case it must not be qualified.
- Each qualifier must be of a higher level than the name it qualifies and must be within the same hierarchy.
- If there is more than one combination of qualifiers that ensures uniqueness, any of those combinations can be used.

Identical names

When programs are directly or indirectly contained within other programs, each program can use identical user-defined words to name resources. A program references the resources that that program describes rather than the same-named resources described in another program, even if the names are different types of user-defined words.

These same rules apply to classes and their contained methods.

References to COPY libraries

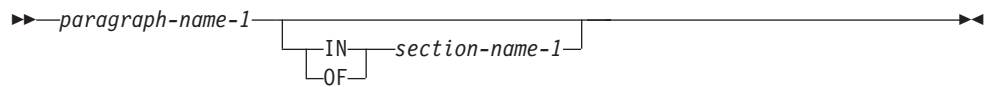


If *library-name-1* is not specified, SYSLIB is assumed as the library name.

For rules on referencing COPY libraries, see “COPY statement” on page 540.

References to procedure division names

Format 1



Format 2



Procedure division names that are explicitly referenced in a program must be unique within a section. A section-name is the highest and only qualifier available for a paragraph-name and must be unique if referenced. (Section-names are described under “Procedures” on page 256.)

If explicitly referenced, a paragraph-name must not be duplicated within a section. When a paragraph-name is qualified by a section-name, the word SECTION must not appear. A paragraph-name need not be qualified when referred to within the section in which it appears. A paragraph-name or section-name that appears in a program cannot be referenced from any other program.

References to data division names

This section discusses the following types of references:

- “Simple data reference”
- “Identifiers” on page 62

Simple data reference

The most basic method of referencing data items in a COBOL program is *simple data reference*, which is *data-name-1* without qualification, subscripting, or reference modification. Simple data reference is used to reference a single elementary or group item.

Format



data-name-1

Can be any data description entry.

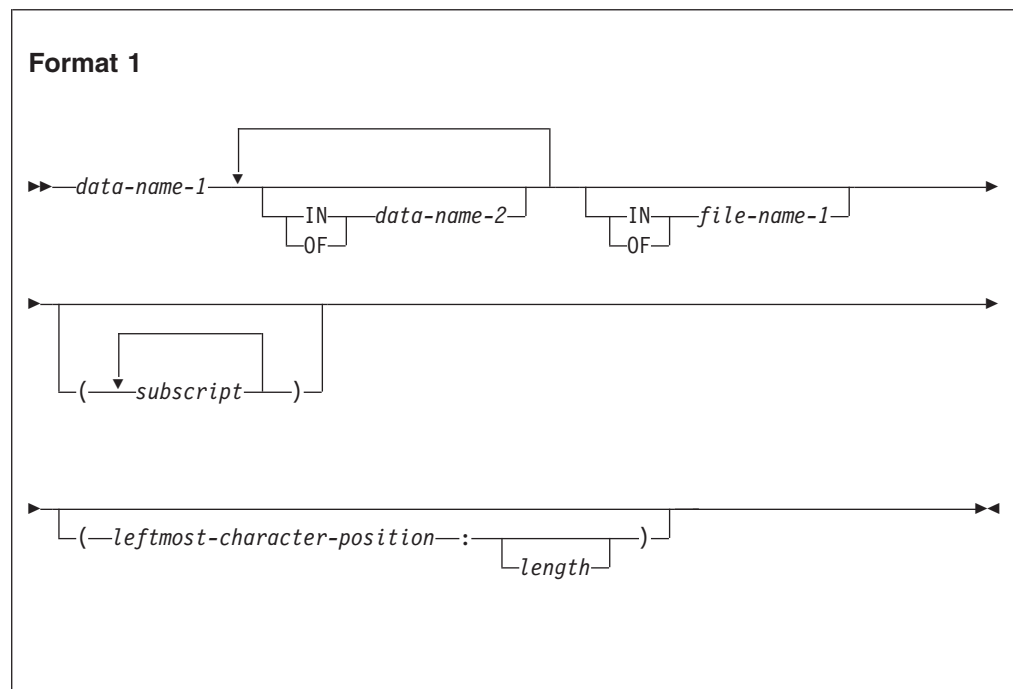
data-name-1 must be unique in a program.

Identifiers

When used in a syntax diagram in this information, the term *identifier* refers to a valid combination of a data-name or function-identifier with its qualifiers, subscripts, and reference-modifiers as required for uniqueness of reference. Rules for identifiers associated with a format can however specifically prohibit qualification, subscripting, or reference modification.

The term *data-name* refers to a name that must not be qualified, subscripted, or reference modified unless specifically permitted by the rules for the format.

- For a description of qualification, see “Qualification” on page 59.
- For a description of subscripting, see “Subscripting” on page 65.
- For a description of reference modification, see “Reference modification” on page 68.



data-name-1, data-name-2

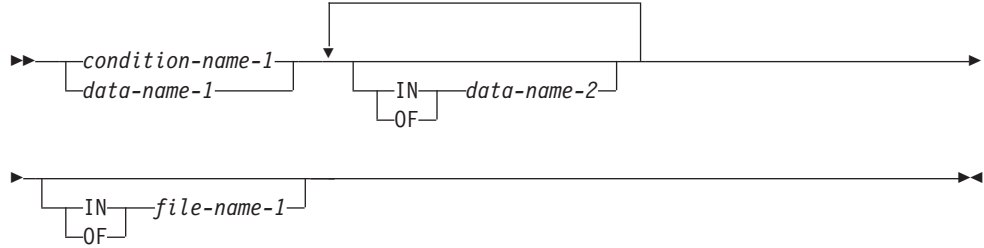
Can be a record-name.

file-name-1

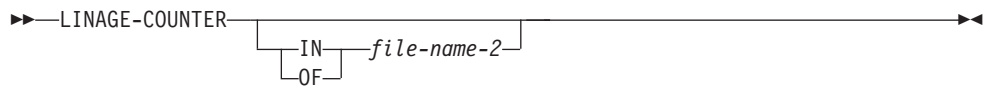
Must be identified by an FD or SD entry in the data division.

file-name-1 must be unique within this program.

Format 2



Format 3



data-name-1, data-name-2

Can be a record-name.

condition-name-1

Can be referenced by statements and entries either in the program that contains the configuration section or in a program contained within that program.

file-name-1

Must be identified by an FD or SD entry in the data division.

Must be unique within this program.

LINAGE-COUNTER

Must be qualified each time it is referenced if more than one file description entry that contains a LINAGE clause has been specified in the source unit.

file-name-2

Must be identified by the FD or SD entry in the data division. *file-name-2* must be unique within this program.

Duplication of data-names must not occur in those places where the data-names cannot be made unique by qualification.

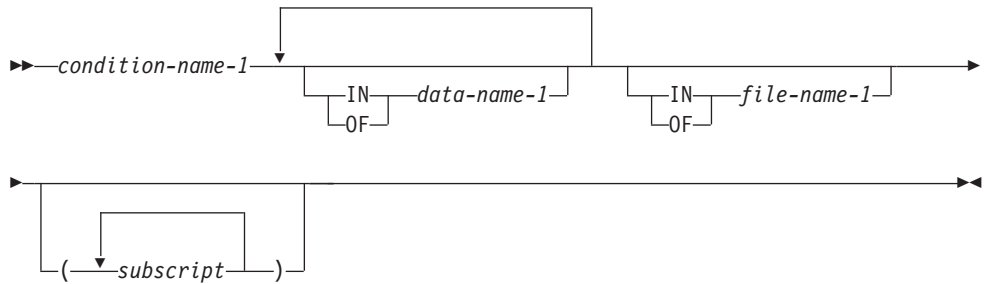
In the same program, the data-name specified as the subject of the entry whose level-number is 01 that includes the EXTERNAL clause must not be the same data-name specified for any other data description entry that includes the EXTERNAL clause.

In the same data division, the data description entries for any two data items for which the same data-name is specified must not include the GLOBAL clause.

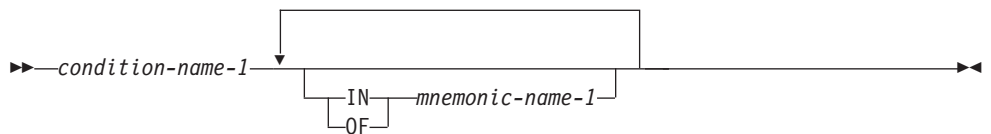
Data division names that are explicitly referenced must either be uniquely defined or made unique through qualification. Unreferenced data items need not be uniquely defined. The highest level in a data hierarchy (a data item associated with a level indicator (FD or SD in the file section) or with level-number 01) must be uniquely named if referenced. Data items associated with level-numbers 02 through 49 are successively lower levels of the hierarchy.

Condition-name

Format 1: condition-name in data division



Format 2: condition-name in SPECIAL-NAMES paragraph



condition-name-1

Can be referenced by statements and entries either in the program that contains the definition of *condition-name-1*, or in a program contained within that program.

If explicitly referenced, a condition-name must be unique or be made unique through qualification or subscripting (or both) except when the scope of names by itself ensures uniqueness of reference.

If qualification is used to make a condition-name unique, the associated conditional variable can be used as the first qualifier. If qualification is used, the hierarchy of names associated with the conditional variable itself must be used to make the condition-name unique.

If references to a conditional variable require subscripting, reference to any of its condition-names also requires the same combination of subscripting.

In this information, *condition-name* refers to a condition-name qualified or subscripted, as necessary.

data-name-1

Can be a record-name.

file-name-1

Must be identified by an FD or SD entry in the data division.

file-name-1 must be unique within this program.

mnemonic-name-1

For information about acceptable values for *mnemonic-name-1*, see “SPECIAL-NAMES paragraph” on page 110.

Index-name

An index-name identifies an index. An index can be regarded as a private special register that the compiler generates for working with a table. You name an index by specifying the INDEXED BY phrase in the OCCURS clause that defines a table.

You can use an index-name in only the following language elements:

- SET statements
- PERFORM statements
- SEARCH statements
- Subscripts
- Relation conditions

An index-name is not the same as the name of an index data item, and an index-name cannot be used like a data-name.

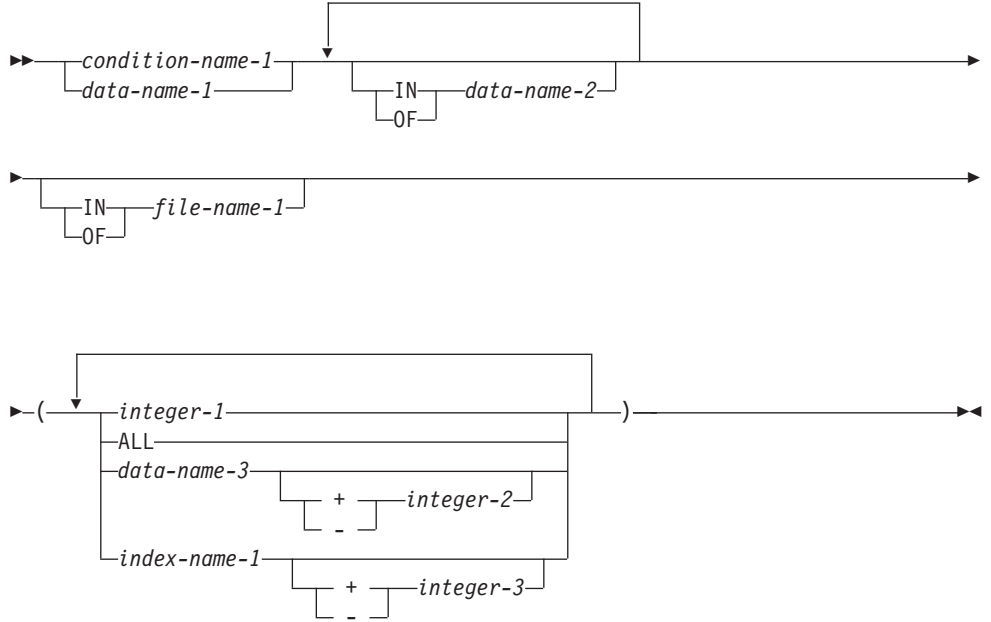
Index data item

An index data item is a data item that can hold the value of an index. You define an index data item by specifying the USAGE IS INDEX clause in a data description entry. The name of an index data item is a data-name. An index data item can be used anywhere a data-name or identifier can be used, unless stated otherwise in the rules of a particular statement. You can use the SET statement to save the value of an index (referenced by index-name) in an index data item.

Subscripting

Subscripting is a method of providing table references through the use of subscripts. A *subscript* is a positive integer whose value specifies the occurrence number of a table element.

Format



condition-name-1

The conditional variable for *condition-name-1* must contain an OCCURS clause or must be subordinate to a data description entry that contains an OCCURS clause.

data-name-1

Must contain an OCCURS clause or must be subordinate to a data description entry that contains an OCCURS clause.

data-name-2, file-name-1

Must name data items or records that contain *data-name-1*.

integer-1

Can be signed. If signed, it must be positive.

data-name-3

Must be a numeric elementary item representing an integer.

data-name-3 can be qualified. *data-name-3* cannot be a windowed date field.

index-name-1

Corresponds to a data description entry in the hierarchy of the table being referenced that contains an INDEXED BY phrase that specifies that name.

integer-2, integer-3

Cannot be signed.

The subscripts, enclosed in parentheses, are written immediately following any qualification for the name of the table element. The number of subscripts in such a reference must equal the number of dimensions in the table whose element is being referenced. That is, there must be a subscript for each OCCURS clause in the hierarchy that contains the data-name including the data-name itself.

When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the data organization. If a multidimensional table is thought of as a series of nested tables and the most inclusive or outermost table in the nest is considered to be the major table with the innermost or least inclusive table being the minor table, the subscripts are written from left to right in the order major, intermediate, and minor.

For example, if TABLE-THREE is defined as:

```
01 TABLE-THREE.  
   05 ELEMENT-ONE OCCURS 3 TIMES.  
     10 ELEMENT-TWO OCCURS 3 TIMES.  
       15 ELEMENT-THREE OCCURS 2 TIMES    PIC X(8).
```

a valid subscripted reference to TABLE-THREE is:

```
ELEMENT-THREE (2 2 1)
```

Subscripted references can also be reference modified. See the third example under “Reference modification examples” on page 71. A reference to an item must not be subscripted unless the item is a table element or an item or condition-name associated with a table element.

Each table element reference must be subscripted except when such reference appears:

- In a USE FOR DEBUGGING statement
- As the subject of a SEARCH statement
- In a REDEFINES clause
- In the KEY is phrase of an OCCURS clause

The lowest permissible occurrence number represented by a subscript is 1. The highest permissible occurrence number in any particular case is the maximum number of occurrences of the item as specified in the OCCURS clause.

Subscripting using data-names

When a data-name is used to represent a subscript, it can be used to reference items within different tables. These tables need not have elements of the same size. The same data-name can appear as the only subscript with one item and as one of two or more subscripts with another item. A data-name subscript can be qualified; it cannot be subscripted or indexed. For example, valid subscripted references to TABLE-THREE, assuming that SUB1, SUB2, and SUB3 are all items subordinate to SUBSCRIPT-ITEM, include:

```
ELEMENT-THREE (SUB1 SUB2 SUB3)
```

```
ELEMENT-THREE IN TABLE-THREE (SUB1 OF SUBSCRIPT-ITEM,  
                                SUB2 OF SUBSCRIPT-ITEM, SUB3 OF SUBSCRIPT-ITEM)
```

Subscripting using index-names (indexing)

Indexing allows such operations as table searching and manipulating specific items. To use indexing, you associate one or more index-names with an item whose data description entry contains an OCCURS clause. An index associated with an index-name acts as a subscript, and its value corresponds to an occurrence number for the item to which the index-name is associated.

The INDEXED BY phrase, by which the index-name is identified and associated with its table, is an optional part of the OCCURS clause. There is no separate entry

to describe the index associated with index-name. At run time, the contents of the index corresponds to an occurrence number for that specific dimension of the table with which the index is associated.

The initial value of an index at run time is undefined, and the index must be initialized before it is used as a subscript. An initial value is assigned to an index with one of the following:

- The PERFORM statement with the VARYING phrase
- The SEARCH statement with the ALL phrase
- The SET statement

The use of an integer or data-name as a subscript that references a table element or an item within a table element does not cause the alteration of any index associated with that table.

An index-name can be used to reference any table. However, the element length of the table being referenced and of the table that the index-name is associated with should match. Otherwise, the reference will not be to the same table element in each table, and you might get runtime errors.

Data that is arranged in the form of a table is often searched. The SEARCH statement provides facilities for producing serial and nonserial searches. It is used to search for a table element that satisfies a specific condition and to adjust the value of the associated index to indicate that table element.

To be valid during execution, an index value must correspond to a table element occurrence of neither less than one, nor greater than the highest permissible occurrence number.

For more information about index-names, see “Index-name” on page 65 and “INDEXED BY phrase” on page 195.

Relative subscripting

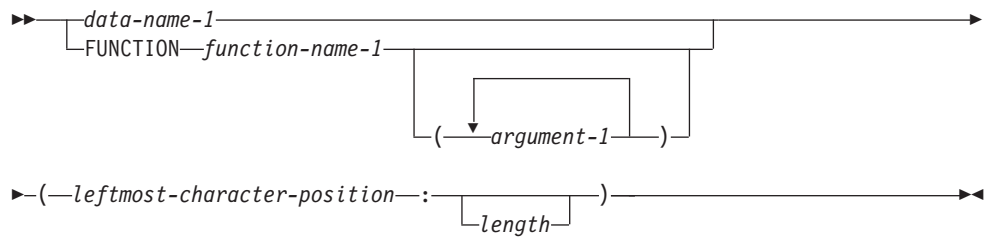
In *relative subscripting*, the name of a table element is followed by a subscript of the form data-name or index-name followed by the operator + or -, and a positive or unsigned integer literal.

The operators + and - must be preceded and followed by a space. The value of the subscript used is the same as if the index-name or data-name had been set up or down by the value of the integer. The use of relative indexing does not cause the program to alter the value of the index.

Reference modification

Reference modification defines a data item by specifying a leftmost character and optional length for the data item.

Format: reference modification



data-name-1

Must reference a data item described explicitly or implicitly with usage DISPLAY, DISPLAY-1, or NATIONAL. A national group item is processed as an elementary data item of category national.

data-name-1 can be qualified or subscripted. *data-name-1* cannot be a windowed date field.

function-name-1

Must reference an alphanumeric or national function.

leftmost-character-position

Must be an arithmetic expression. The evaluation of *leftmost-character-position* must result in a positive nonzero integer that is less than or equal to the number of characters in the data item referenced by *data-name-1*.

The evaluation of *leftmost-character-position* must not result in a windowed date field.

length

Must be an arithmetic expression.

The evaluation of *length* must result in a positive nonzero integer.

The evaluation of *length* must not result in a windowed date field.

The sum of *leftmost-character-position* and *length* minus the value 1 must be less than or equal to the number of character positions in *data-name-1*. If *length* is omitted, the length used will be equal to the number of character positions in *data-name-1* plus 1, minus *leftmost-character-position*.

For usages DISPLAY-1 and NATIONAL, each character position occupies 2 bytes. Reference modification operates on whole character positions and not on the individual bytes of the characters in usages DISPLAY-1 and NATIONAL. For usage DISPLAY, reference modification operates as though each character were a single-byte character.

Unless otherwise specified, reference modification is allowed anywhere an identifier or function-identifier that references a data item or function with the same usage as the reference-modified data item is permitted.

Each character position referenced by *data-name-1* or *function-name-1* is assigned an ordinal number incrementing by one from the leftmost position to the rightmost position. The leftmost position is assigned the ordinal number one. If the data description entry for *data-name-1* contains a SIGN IS SEPARATE clause, the sign position is assigned an ordinal number within that data item.

If *data-name-1* is described with usage DISPLAY and category numeric, numeric-edited, alphabetic, alphanumeric-edited, or external floating-point, *data-name-1* is operated upon for purposes of reference modification as if it were redefined as a data item of category alphanumeric with the same size as the data item referenced by *data-name-1*.

If *data-name-1* is described with usage NATIONAL and category numeric, numeric-edited, national-edited, or external floating-point, *data-name-1* is operated upon for purposes of reference modification as if it were redefined as a data item of category national with the same size as the data item referenced by *data-name-1*.

If *data-name-1* is a national group item, *data-name-1* is processed as an elementary data item of category national.

If *data-name-1* is an expanded date field, then the result of reference modification is a nondate.

Reference modification creates a unique data item that is a subset of *data-name-1* or a subset of the value referenced by *function-name-1* and its arguments, if any. This unique data item is considered an elementary data item without the JUSTIFIED clause.

When a function is reference-modified, the unique data item has class, category, and usage national if the type of the function is national; otherwise, it has class and category alphanumeric and usage display.

When *data-name-1* is reference-modified, the unique data item has the same class, category, and usage as that defined for the data item referenced by *data-name-1* except that:

- If *data-name-1* has category national-edited, the unique data item has category national.
- If *data-name-1* has usage NATIONAL and category numeric-edited, numeric, or external floating-point, the unique data item has category national.
- If *data-name-1* has usage DISPLAY, and category numeric-edited, alphanumeric-edited, numeric, or external floating-point, the unique data item has category alphanumeric.
- If *data-name-1* references an alphanumeric group item, the unique data item is considered to have usage DISPLAY and category alphanumeric.
- If *data-name-1* references a national group item, the unique data item has usage NATIONAL and category national.

If *length* is not specified, the unique data item created extends from and includes the character position identified by *leftmost-character-position* up to and including the rightmost character position of the data item referenced by *data-name-1*.

Evaluation of operands

Reference modification for an operand is evaluated as follows:

- If subscripting is specified for the operand, the reference modification is evaluated immediately after evaluation of the subscript.
- If subscripting is not specified for the operand, the reference modification is evaluated at the time subscripting would be evaluated if subscripts had been specified.

Reference modification examples

The following statements transfer the first 10 characters of the data-item referenced by WHOLE-NAME to the data-item referenced by FIRST-NAME.

```
77 WHOLE-NAME PIC X(25).  
77 FIRST-NAME PIC X(10).  
77 START-P PIC 9(9) BINARY VALUE 1.  
77 LENGTH-STR PIC 9(9) BINARY VALUE 10.  
...  
MOVE WHOLE-NAME(1:10) TO FIRST-NAME.  
MOVE WHOLE-NAME(START-P:END-P) TO FIRST-NAME.
```

The following statement transfers the last 15 characters of the data-item referenced by WHOLE-NAME to the data-item referenced by LAST-NAME.

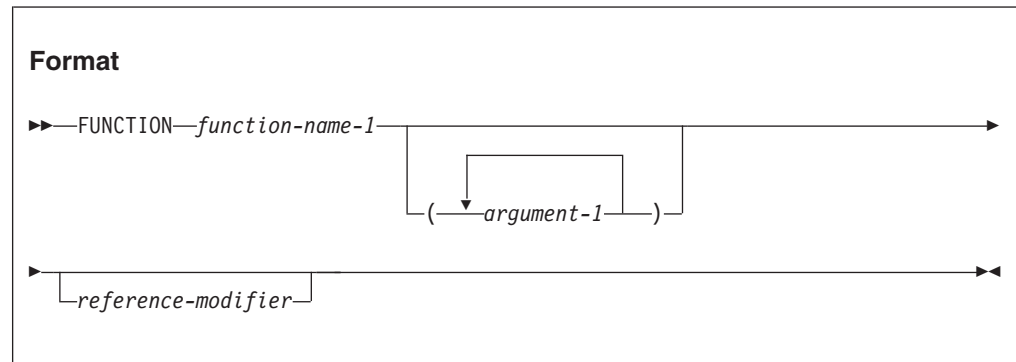
```
77 WHOLE-NAME PIC X(25).  
77 LAST-NAME PIC X(15).  
...  
MOVE WHOLE-NAME(11:) TO LAST-NAME.
```

The following statement transfers the fourth and fifth characters of the third occurrence of TAB to the variable SUFFIX.

```
01 TABLE-1.  
02 TAB OCCURS 10 TIMES PICTURE X(5).  
77 SUFFIX PICTURE X(2).  
...  
MOVE TAB OF TABLE-1 (3) (4:2) TO SUFFIX.
```

Function-identifier

A *function-identifier* is a sequence of character strings and separators that uniquely references the data item that results from the evaluation of a function.



argument-1

Must be an identifier, literal (other than a figurative constant), or arithmetic expression.

For more information, see Chapter 22, “Intrinsic functions,” on page 489.

function-name-1

function-name-1 must be one of the intrinsic function names.

reference-modifier

Can be specified only for functions of the type alphanumeric or national.

A function-identifier that makes reference to an alphanumeric or national function can be specified anywhere that a data item of category alphanumeric or category national, respectively, can be referenced and where references to functions are not specifically prohibited, except as follows:

- As a receiving operand of any statement
- Where a data item is required to have particular characteristics (such as class and category, size, sign, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have these characteristics

A function-identifier that makes reference to an integer or numeric function can be used wherever an arithmetic expression can be used.

Data attribute specification

Explicit data attributes are data attributes that you specify in COBOL coding.

Implicit data attributes are default values. If you do not explicitly code a data attribute, the compiler assumes a default value.

For example, you need not specify the USAGE of a data item. If USAGE is omitted and the symbol N is not specified in the PICTURE clause, the default is USAGE DISPLAY, which is the implicit data attribute. When PICTURE symbol N is used, USAGE DISPLAY-1 is the default when the NSYMBOL(DBCS) compiler option is in effect; USAGE NATIONAL is the default when the NSYMBOL(NATIONAL) compiler option is in effect. These are implicit data attributes.

Chapter 9. Transfer of control

In the procedure division, unless there is an *explicit* control transfer or there is no next executable statement, program flow transfers control from statement to statement in the order in which the statements are written. This normal program flow is an *implicit* transfer of control.

In addition to the implicit transfers of control between consecutive statements, implicit transfer of control also occurs when the normal flow is altered without the execution of a procedure branching statement. The following examples show *implicit* transfers of control, overriding statement-to-statement transfer of control:

- After execution of the last statement of a procedure that is executed under control of another COBOL statement, control implicitly transfers. (COBOL statements that control procedure execution are, for example, MERGE, PERFORM, SORT, and USE.) Further, if a paragraph is being executed under the control of a PERFORM statement that causes iterative execution, and that paragraph is the first paragraph in the range of that PERFORM statement, an implicit transfer of control occurs between the control mechanism associated with that PERFORM statement and the first statement in that paragraph for each iterative execution of the paragraph.
- During SORT or MERGE statement execution, control is implicitly transferred to an input or output procedure.
- During XML PARSE statement execution, control is implicitly transferred to a processing procedure.
- During execution of any COBOL statement that causes execution of a declarative procedure, control is implicitly transferred to that procedure.
- At the end of execution of any declarative procedure, control is implicitly transferred back to the control mechanism associated with the statement that caused its execution.

COBOL also provides *explicit* control transfers through the execution of any procedure branching, program call, or conditional statement. (Lists of procedure branching and conditional statements are contained in “Statement categories” on page 286.)

Definition: The term *next executable statement* refers to the next COBOL statement to which control is transferred, according to the rules given above. There is no next executable statement under the following circumstances:

- When the program contains no procedure division
- Following the last statement in a declarative section when the paragraph in which it appears is not being executed under the control of some other COBOL statement
- Following the last statement in a program or method when the paragraph in which it appears is not being executed under the control of some other COBOL statement in that program
- Following the last statement in a declarative section when the statement is in the range of an active PERFORM statement executed in a different section and this last statement of the declarative section is not also the last statement of the procedure that is the exit of the active PERFORM statement
- Following a STOP RUN statement or EXIT PROGRAM statement that transfers control outside the COBOL program

- Following a GOBACK statement that transfers control outside the COBOL program
- Following an EXIT METHOD statement that transfers control outside the COBOL method
- The end program or end method marker

When there is no next executable statement and control is not transferred outside the COBOL program, the program flow of control is undefined unless the program execution is in the nondeclarative procedures portion of a program under control of a CALL statement, in which case an implicit EXIT PROGRAM statement is executed.

Similarly, if control reaches the end of the procedure division of a method and there is no next executable statement, an implicit EXIT METHOD statement is executed.

Chapter 10. Millennium Language Extensions and date fields

Many applications use two digits rather than four digits to represent the year in date fields, and assume that these values represent years from 1900 to 1999. This compact date format works well for the 1900s, but it does not work for the year 2000 and beyond because these applications interpret "00" as 1900 rather than 2000, producing incorrect results.

The millennium language extensions are designed to allow applications that use two-digit years to continue performing correctly in the year 2000 and beyond, with minimal modification to existing code. This is achieved using a technique known as *windowing*, which removes the assumption that all two-digit year fields represent years from 1900 to 1999. Instead, windowing enables two-digit year fields to represent years within a 100-year range known as a *century window*.

For example, if a two-digit year field contains the value 15, many applications would interpret the year as 1915. However, with a century window of 1960–2059, the year would be interpreted as 2015.

The millennium language extensions provide support for the most common operations on date fields: comparisons, moving and storing, and incrementing and decrementing. This support is limited to date fields of certain formats; for details, see "DATE FORMAT clause" on page 184.

For information about supported operations and restrictions when using date fields, see "Restrictions on using date fields" on page 186.

Millennium Language Extensions syntax

The millennium language extensions introduce the following language elements:

- The DATE FORMAT clause in data description entries, which defines data items as date fields.
- The following intrinsic functions:

DATEVAL

Converts a nondate to a date field.

UNDATE

Converts a date field to a nondate.

YEARWINDOW

Returns the first year of the century window specified by the YEARWINDOW compiler option.

For details on using the millennium language extensions in applications, see *Millennium language extensions (MLE)* in the *COBOL for AIX Programming Guide*.

The millennium language extensions have no effect unless your program is compiled using the DATEPROC compiler option and the century window is specified by the YEARWINDOW compiler option.

Terms and concepts

This document uses the following terms when referring to the millennium language extensions:

- “Date field”
- “Nondate” on page 77
- “Century window” on page 77

Date field

A *date field* can be any of the following:

- A data item whose data description entry includes a DATE FORMAT clause
- A value returned by one of the following intrinsic functions:
 - DATE-OF-INTEGER
 - DATE-TO-YYYYMMDD
 - DATEVAL
 - DAY-OF-INTEGER
 - DAY-TO-YYYYDDD
 - YEAR-TO-YYYY
 - YEARWINDOW
- The conceptual data items DATE, DATE YYYYMMDD, DAY, or DAY YYYYDDD of the ACCEPT statement
- The result of certain arithmetic operations (for details, see “Arithmetic with date fields” on page 260)

The term *date field* refers to both *expanded date fields* and *windowed date fields*.

Windowed date field

A *windowed date field* is a date field that contains a windowed year. A *windowed year* consists of two digits, representing a year within the century window.

Expanded date field

An *expanded date field* is a date field that contains an expanded year. An *expanded year* consists of four digits.

The main use of expanded date fields is to provide correct results when these are used in combination with windowed date fields; for example, where migration to four-digit year dates is not complete. If all the dates in an application use four-digit years, there is no need to use the millennium language extensions.

Year-last date field

A *year-last date field* is a date field whose DATE FORMAT clause specifies one or more Xs preceding the YY or YYYY. Year-last date fields are supported in a limited number of operations, typically involving another date with the same (year-last) date format, or a nondate.

Date format

Date format refers to the date pattern of a date field, specified either:

- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function *argument-2*
- Implicitly, by statements and intrinsic functions that return date fields.

Compatible date field

The meaning of the term *compatible*, when applied to date fields, depends on the COBOL division in which the date field occurs:

Data division

Two date fields are compatible if they have identical USAGE and meet at least one of the following conditions:

- They have the same date format.
- Both are windowed date fields, where one consists only of a windowed year, DATE FORMAT YY.
- Both are expanded date fields, where one consists only of an expanded year, DATE FORMAT YYYY.
- One has DATE FORMAT YYXXXX, the other, YYYY.
- One has DATE FORMAT YYYYXXXX, the other, YYYYXX.

A windowed date field can be subordinate to an expanded date group data item. The two date fields are compatible if the subordinate date field has USAGE DISPLAY, starts 2 bytes after the start of the group expanded date field, and the two fields meet at least one of the following conditions:

- The subordinate date field has a DATE FORMAT pattern with the same number of Xs as the DATE FORMAT pattern of the group date field.
- The subordinate date field has DATE FORMAT YY.
- The group date field has DATE FORMAT YYYYXXXX and the subordinate date field has DATE FORMAT YYYY.

Procedure division

Two date fields are compatible if they have the same date format except for the year part, which can be windowed or expanded. For example, a windowed date field with DATE FORMAT YYXXX is compatible with:

- Another windowed date field with DATE FORMAT YYXXX
- An expanded date field with DATE FORMAT YYYYXXX

Nondate

A *nondate* can be any of the following:

- A data item whose date description entry does not include the DATE FORMAT clause
- A date field that has been converted using the UNDATE function
- A literal
- A reference-modified date field
- The result of certain arithmetic operations that can include date field operands; for example, the difference between two compatible date fields

Century window

A *century window* is a 100-year interval within which any two-digit year is unique. There are several ways to specify a century window in a COBOL program:

- For windowed date fields, a century window is specified by the YEARWINDOW compiler option.
- For windowing intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY, a century window is specified by *argument-2*.

Part 2. COBOL source unit structure

Chapter 11. COBOL program structure	81
Nested programs	83
Conventions for program-names	84
Rules for program-names.	84
Chapter 12. COBOL class definition structure	87
Chapter 13. COBOL method definition structure	91

Chapter 11. COBOL program structure

A COBOL source program is a syntactically correct set of COBOL statements.

Nested programs

A *nested program* is a program that is contained in another program. Contained programs can reference some of the resources of the programs that contain them. If program B is contained in program A, it is *directly* contained if there is no program contained in program A that also contains program B. Program B is *indirectly* contained in program A if there exists a program contained in program A that also contains program B. For more information about nested programs, see *Nested programs* in the *COBOL for AIX Programming Guide*.

Object program

An *object program* is a set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. An object program is generally the machine language result of the operation of a COBOL compiler on a source program. The term object program also refers to the methods that result from compiling a class definition.

Run unit

A *run unit* is one or more object programs that interact with one another and that function at run time as an entity to provide problem solutions.

Sibling program

Sibling programs are programs that are directly contained in the same program.

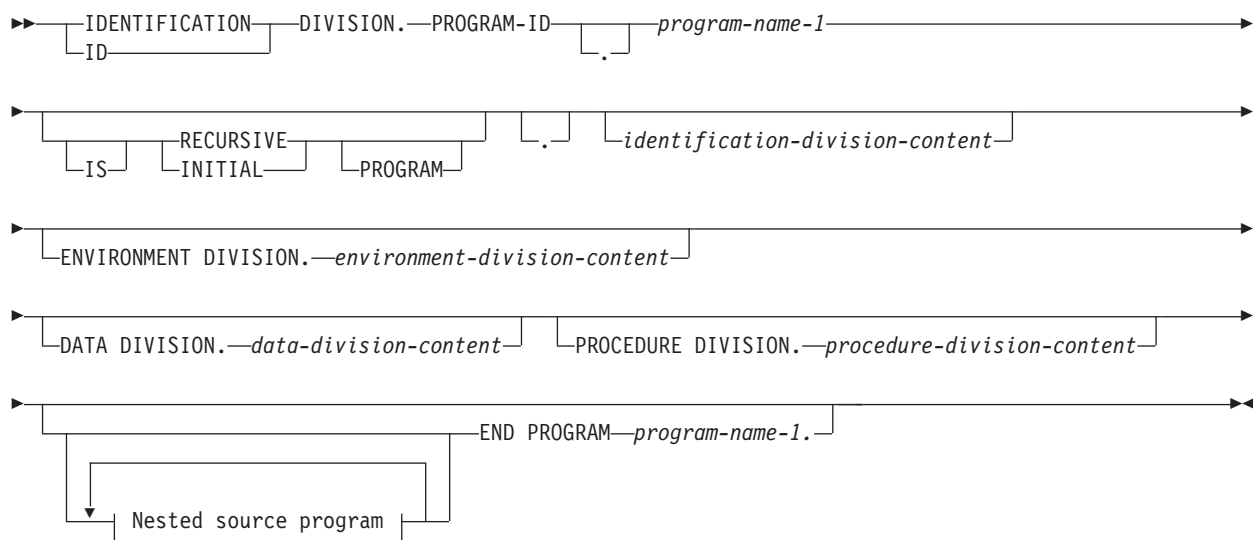
With the exception of the COPY and REPLACE statements and the end program marker, the statements, entries, paragraphs, and sections of a COBOL source program are grouped into the following four divisions:

- Identification division
- Environment division
- Data division
- Procedure division

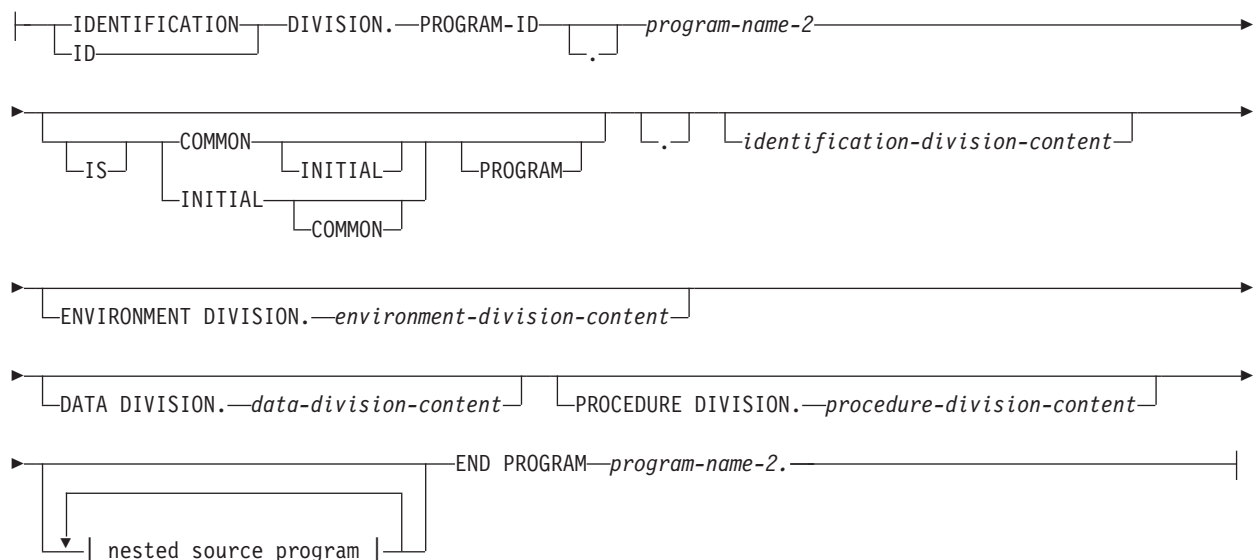
The end of a COBOL source program is indicated by the END PROGRAM marker. If there are no nested programs, the absence of additional source program lines also indicates the end of a COBOL program.

Following is the format for the entries and statements that constitute a separately compiled COBOL source program.

Format: COBOL source program



nested source program:



A sequence of separate COBOL programs can also be input to the compiler. The following is the format for the entries and statements that constitute a sequence of source programs (batch compile).

Format: sequence of COBOL source programs



END PROGRAM *program-name*

An end program marker separates each program in the sequence of programs. *program-name* must be identical to a program-name declared in a preceding program-ID paragraph.

program-name can be specified either as a user-defined word or in an alphanumeric literal. Either way, *program-name* must follow the rules for forming a user-defined word. *program-name* cannot be a figurative constant. Any lowercase letters in the literal are folded to uppercase.

An end program marker is optional for the last program in the sequence only if that program does not contain any nested source programs.

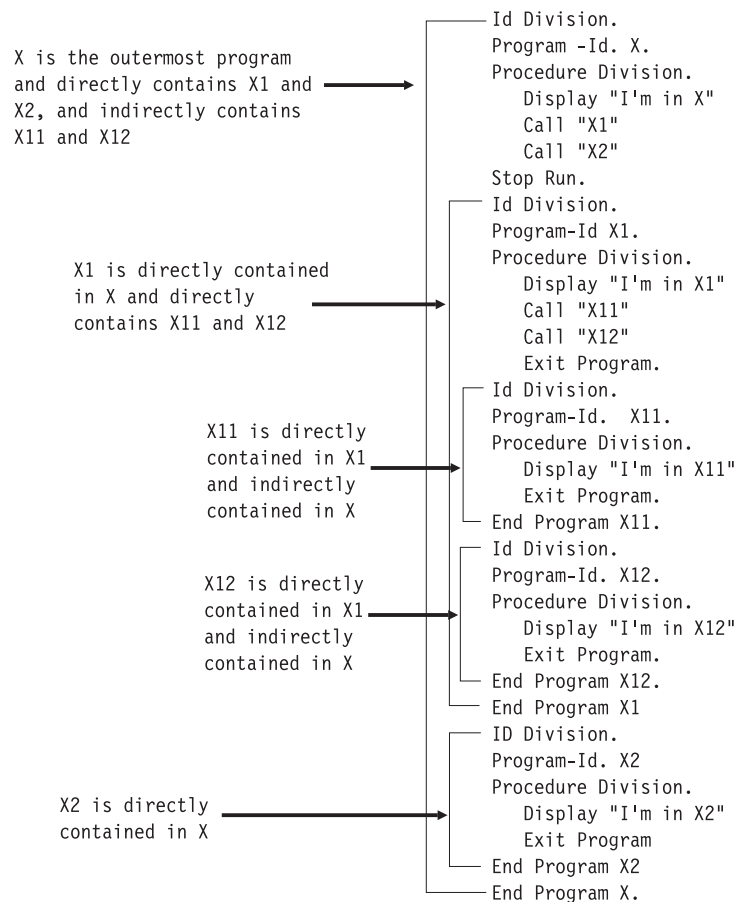
Nested programs

A COBOL program can contain other COBOL programs, which in turn can contain still other COBOL programs. These contained programs are called *nested programs*. Nested programs can be *directly* or *indirectly* contained in the containing program.

In the following code fragment, program Outer-program *directly* contains program Inner-1. Program Inner-1 *directly* contains program Inner-1a, and Outer-program *indirectly* contains Inner-1a:

```
Id division.
Program-id. Outer-program.
  Procedure division.
    Call "Inner-1".
    Stop run.
Id division.
Program-id. Inner-1
...
  Call Inner-1a.
  Stop run.
Id division.
Program-id. Inner-1a.
...
End Inner-1a.
End Inner-1.
End Outer-program.
```

The following figure describes a more complex nested program structure with directly and indirectly contained programs.



Conventions for program-names

The program-name of a program is specified in the PROGRAM-ID paragraph of the program's identification division. A program-name can be referenced only by the CALL statement, the CANCEL statement, the SET statement, or the END PROGRAM marker. Names of programs that constitute a run unit are not necessarily unique, but when two programs in a run unit are identically named, at least one of the programs must be directly or indirectly contained within another separately compiled program that does not contain the other of those two programs.

A separately compiled program and all of its directly and indirectly contained programs must have unique program-names within that separately compiled program.

Rules for program-names

The following rules define the scope of a program-name:

- If the program-name is that of a program that does not possess the COMMON attribute and that program is directly contained within another program, that program-name can be referenced only by statements included in that containing program.
- If the program-name is that of a program that does possess the COMMON attribute and that program is directly contained within another program, that program-name can be referenced only by statements included in the containing

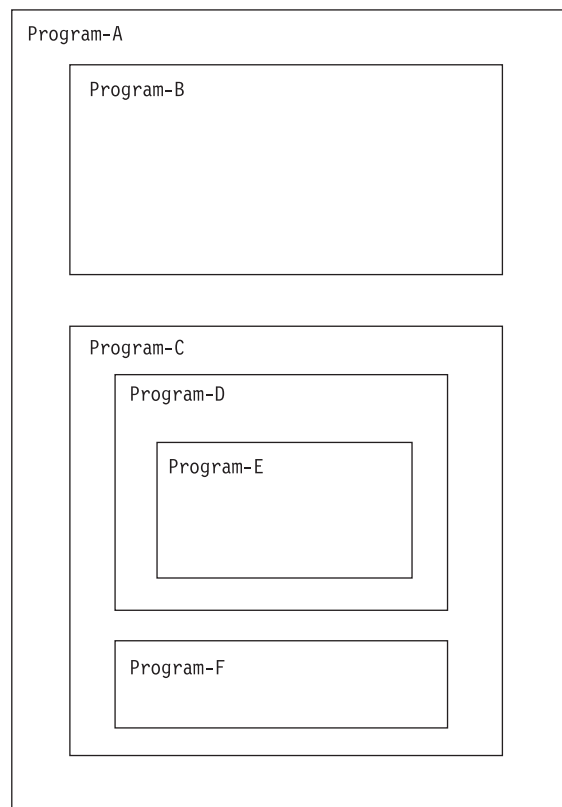
program and any programs directly or indirectly contained within that containing program, except that program possessing the COMMON attribute and any programs contained within it.

- If the program-name is that of a program that is separately compiled, that program-name can be referenced by statements included in any other program in the run unit, except programs it directly or indirectly contains.

The mechanism used to determine which program to call is as follows:

- If one of two programs that have the same name as that specified in the CALL statement is directly contained within the program that includes the CALL statement, that program is called.
- If one of two programs that have the same name as that specified in the CALL statement possesses the COMMON attribute and is directly contained within another program that directly or indirectly contains the program that includes the CALL statement, that common program is called unless the calling program is contained within that common program.
- Otherwise, the separately compiled program is called.

The following rules apply to referencing a program-name of a program that is contained within another program. For this discussion, Program-A contains Program-B and Program-C; Program-C contains Program-D and Program-F; and Program-D contains Program-E.



If Program-D does not possess the COMMON attribute, then Program-D can be referenced only by the program that directly contains Program-D, that is, Program-C.

If Program-D does possess the COMMON attribute, then Program-D can be referenced by Program-C (because Program-C contains Program-D) and by any programs contained in Program-C except for programs contained in Program-D. In other words, if Program-D possesses the COMMON attribute, Program-D can be referenced in Program-C and Program-F but not by statements in Program-E, Program-A, or Program-B.

Chapter 12. COBOL class definition structure

COBOL for AIX provides object-oriented syntax to facilitate interoperation of COBOL and Java programs.

You can use object-oriented syntax to:

- Define classes, with methods and data implemented in COBOL
- Create instances of Java or COBOL classes
- Invoke methods on Java or COBOL objects
- Write classes that inherit from Java classes or from other COBOL classes
- Define and invoke overloaded methods

Basic Java-oriented object capabilities are accessed directly through COBOL language features. Additional capabilities are available to the COBOL programmer by calling services through the Java Native Interface (JNI), as described in *Accessing JNI services* in the *COBOL for AIX Programming Guide*.

Java programs can be multithreaded, and Java interoperation requires toleration of asynchronous signals. Therefore, to mix COBOL with these Java programs, you must use the thread enablement provided by the THREAD compiler option, as described in *THREAD* in the *COBOL for AIX Programming Guide*.

Java String data is represented at run time in Unicode. The Unicode support provided in COBOL for AIX with the national data type enables COBOL programs to exchange String data with Java.

The following are the entities and concepts used in object-oriented COBOL for Java interoperability:

Class The entity that defines operations and state for zero, one, or more object instances and defines operations and state for a common object (a factory object) that is shared by multiple object instances.

You create object instances using the NEW operand of the COBOL INVOKE statement or using a Java class instance creation expression.

Object instances are automatically freed by the Java runtime system's garbage collection when they are no longer in use. You cannot explicitly free individual objects.

Instance method

Procedural code that defines one of the operations supported for the object instances of a class. Instance methods introduced by a COBOL class are defined within the object paragraph of the class definition.

COBOL instance methods are equivalent to public nonstatic methods in Java.

You execute instance methods on a particular object instance by using a COBOL INVOKE statement or a Java method invocation expression.

Instance data

Data that defines the state of an individual object instance. Instance data in a COBOL class is defined in the working-storage section of the data division within the object paragraph of a class definition.

COBOL instance data is equivalent to private nonstatic member data in a Java class.

The state of an object also includes the state of the instance data introduced by inherited classes. Each instance object has its own copy of the instance data defined within its class definition and its own copy of the instance data defined in inherited classes.

You can access COBOL object instance data only from within COBOL instance methods defined in the class definition that defines the data.

You can initialize object instance data with VALUE clauses or you can write an instance method to perform custom initialization.

Factory method, static method

Procedural code that defines one of the operations supported for the common factory object of the class. COBOL factory methods are defined within the factory paragraph of a class definition. Factory methods are associated with a class, not with any individual instance object of the class.

COBOL factory methods are equivalent to public static methods in Java.

You execute COBOL factory methods from COBOL using an INVOKE statement that specifies the class-name as the first operand. You execute them from a Java program using a static method invocation expression.

A factory method cannot operate directly on instance data of its class, even though the data is described in the same class definition; a factory method must invoke instance methods to act on instance data.

COBOL factory methods are typically used to define customized methods that create object instances. For example, you can code a customized factory method that accepts initial values as parameters, creates an instance object using the NEW operand of the INVOKE statement, and then invokes a customized instance method passing those initial values as arguments for use in initializing the instance object.

Factory data, static data

Data associated with a class, rather than with an individual object instance. COBOL factory data is defined in the working-storage section of the data division within the factory paragraph of a class definition.

COBOL factory data is equivalent to private static data in Java.

There is a single copy of factory data for a class. Factory data is associated only with the class and is shared by all object instances of the class. It is not associated with any particular instance object. A factory data item might be used, for example, to keep a count of the number of instance objects that have been created.

You can access COBOL factory data only within COBOL factory methods defined in the same class definition.

Inheritance

Inheritance is a mechanism whereby a class definition (the inheriting class) acquires the methods, data descriptions, and file descriptions written in another class definition (the inherited class). When two classes in an inheritance relationship are considered together, the inheriting class is the *subclass* (or derived class or child class); the inherited class is the *superclass* (or parent class). The inheriting class also indirectly acquires the methods, data descriptions, and file descriptions that the parent class inherited from its parent class.

A COBOL class must inherit from exactly one parent class, which can be implemented in COBOL or Java.

Every COBOL class must inherit directly or indirectly from the `java.lang.Object` class.

Instance variable

An individual data item defined in the data division of an object paragraph.

Java Native Interface (JNI)

A facility of Java designed for interoperation with non-Java programs.

Java Native Interface (JNI) environment pointer

A pointer used to obtain the address of the JNI environment structure used for calling JNI services. The COBOL special register `JNIENVPTR` is provided for referencing the JNI environment pointer.

Object reference

A data item that contains information used to identify and reference an individual object. An object reference can refer to an object that is an instance of a Java or COBOL class.

Subclass

A class that inherits from another class; also called a *derived class* or *child class* of the inherited class.

Superclass

A class that is inherited by another class; also called a *parent class* of the inheriting class.

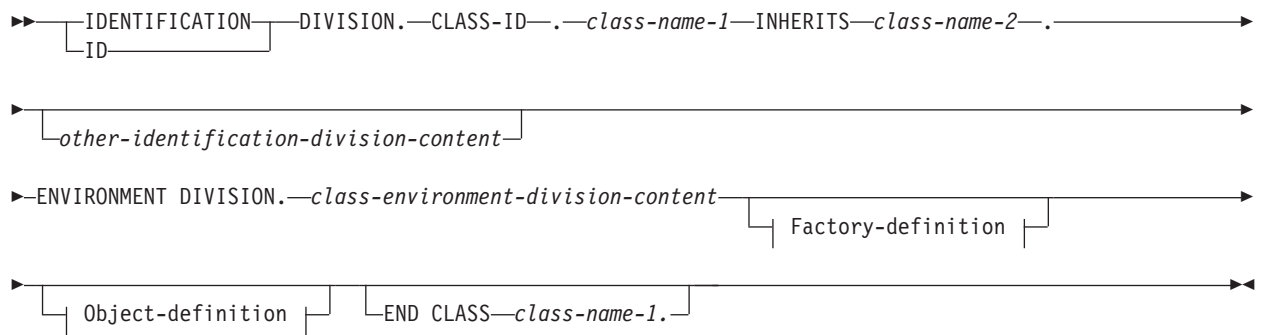
With the exception of the `COPY` and `REPLACE` statements and the `END CLASS` marker, the statements, entries, paragraphs, and sections of a COBOL class definition are grouped into the following structure:

- Identification division
- Environment division (configuration section only)
- Factory definition
 - Identification division
 - Data division
 - Procedure division (containing one or more method definitions)
- Object definition
 - Identification division
 - Data division
 - Procedure division (containing one or more method definitions)

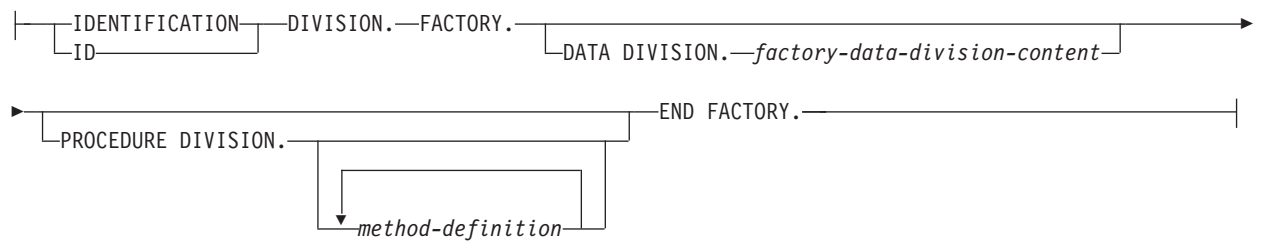
The end of a COBOL class definition is indicated by the `END CLASS` marker.

The following is the format for a COBOL class definition.

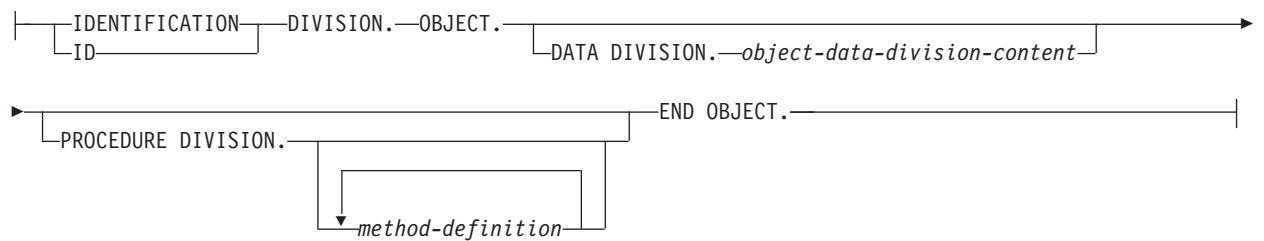
Format: COBOL class definition



Factory-definition:



Object-definition:



END CLASS

Specifies the end of a class definition.

END FACTORY

Specifies the end of a factory definition.

END OBJECT

Specifies the end of an object definition.

Chapter 13. COBOL method definition structure

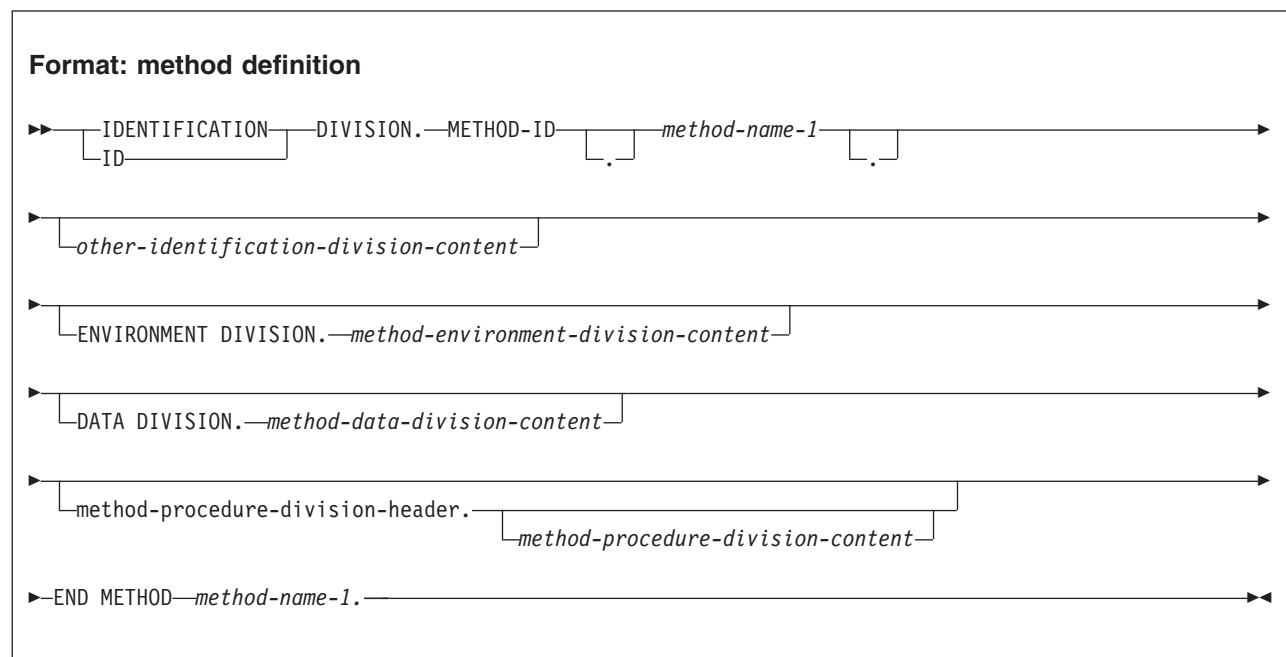
A COBOL method definition describes a method. You can specify method definitions only within the factory paragraph and the object paragraph of a class definition.

With the exception of COPY and REPLACE statements and the END METHOD marker, the statements, entries, paragraphs, and sections of a COBOL method definition are grouped into the following four divisions:

- Identification division
- Environment division (input-output section only)
- Data division
- Procedure division

The end of a COBOL method definition is indicated by the END METHOD marker.

The following is the format for a COBOL method definition.



METHOD-ID

Identifies a method definition. See “METHOD-ID paragraph” on page 102 for details.

method-procedure-division-header

Indicates the start of the procedure division and identifies method parameters and the returning item, if any. See “The procedure division header” on page 251 for details.

END METHOD

Specifies the end of a method definition.

Methods defined in an object definition are *instance methods*. An instance method in a given class can access:

- Data defined in the data division of the object paragraph of that class (*instance data*)
- Data defined in the data division of that instance method (*method data*)

An instance method cannot directly access instance data defined in a parent class, factory data defined in its own class, or method data defined in another method of its class. It must invoke a method to access such data.

Methods defined in a factory definition are *factory methods*. A factory method in a given class can access:

- Data defined in the data division of the factory paragraph of that class (*factory data*)
- Data defined in the data division of that factory method (*method data*)

A factory method cannot directly access factory data defined in a parent class, instance data defined in its own class, or method data defined in another method of its class. It must invoke a method to access such data.

Methods can be invoked from COBOL programs and methods, and they can be invoked from Java programs. A method can execute an INVOKE statement that directly or indirectly invokes itself. Therefore, COBOL methods are implicitly recursive (unlike COBOL programs, which support recursion only if the RECURSIVE attribute is specified in the program-ID paragraph.)

Part 3. Identification division

Chapter 14. Identification division	95
PROGRAM-ID paragraph	98
CLASS-ID paragraph	100
General rules	101
Inheritance	101
FACTORY paragraph	101
OBJECT paragraph	102
METHOD-ID paragraph	102
Method signature	102
Method overloading, overriding, and hiding	102
Method overloading	102
Method overriding (for instance methods)	102
Method hiding (for factory methods)	103
Optional paragraphs	103

Chapter 14. Identification division

The identification division must be the first division in each COBOL source program, factory definition, object definition, and method definition. The identification division names the program, class, or method and identifies the factory definition and object definition. The identification division can include the date a program, class, or method was written, the date of compilation, and other such documentary information.

Program IDENTIFICATION DIVISION

For a program, the first paragraph of the identification division must be the PROGRAM-ID paragraph. The other paragraphs are optional and can appear in any order.

Class IDENTIFICATION DIVISION

For a class, the first paragraph of the identification division must be the CLASS-ID paragraph. The other paragraphs are optional and can appear in any order.

Factory IDENTIFICATION DIVISION

A factory IDENTIFICATION DIVISION contains only a factory paragraph header.

Object IDENTIFICATION DIVISION

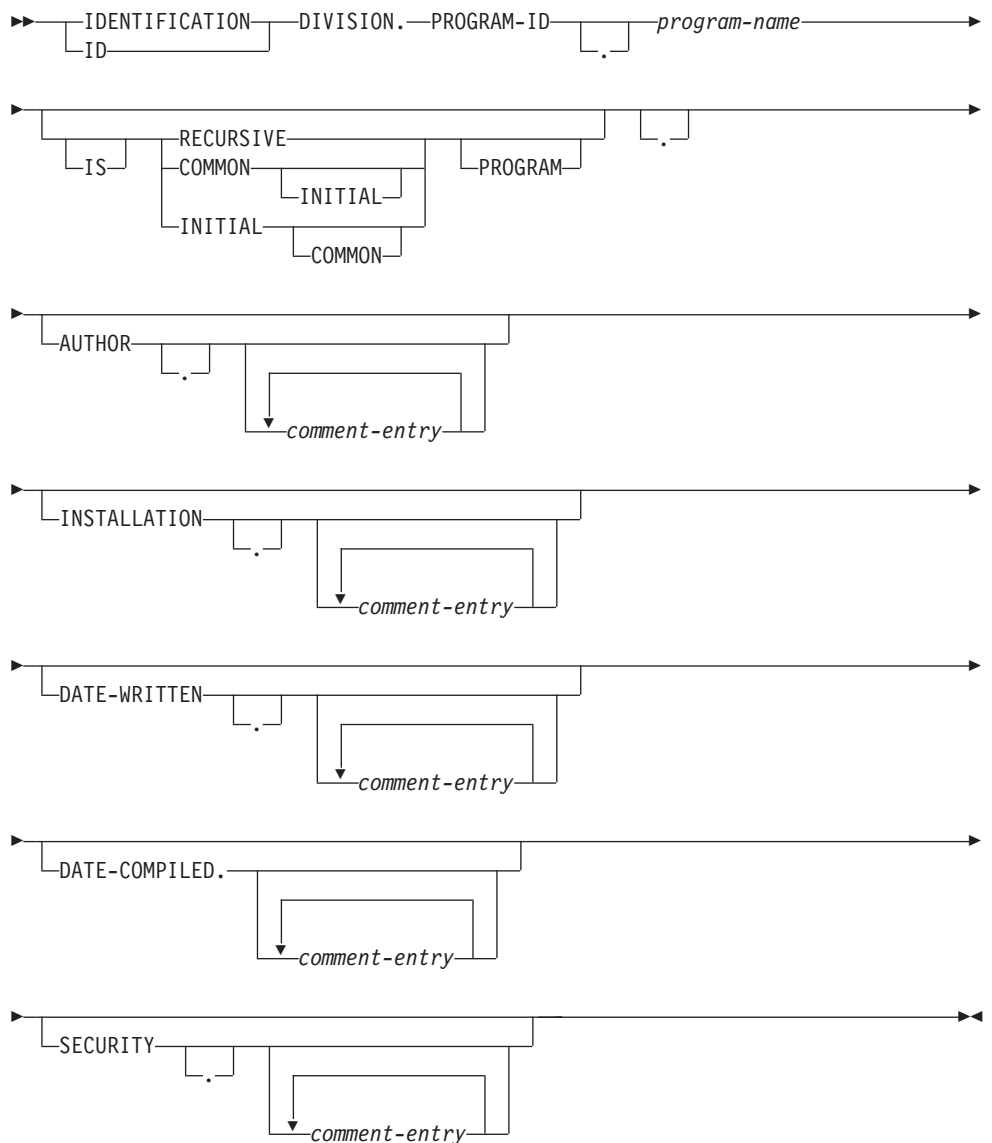
An object IDENTIFICATION DIVISION contains only an object paragraph header.

Method IDENTIFICATION DIVISION

For a method, the first paragraph of the identification division must be the METHOD-ID paragraph. The other paragraphs are optional and can appear in any order.

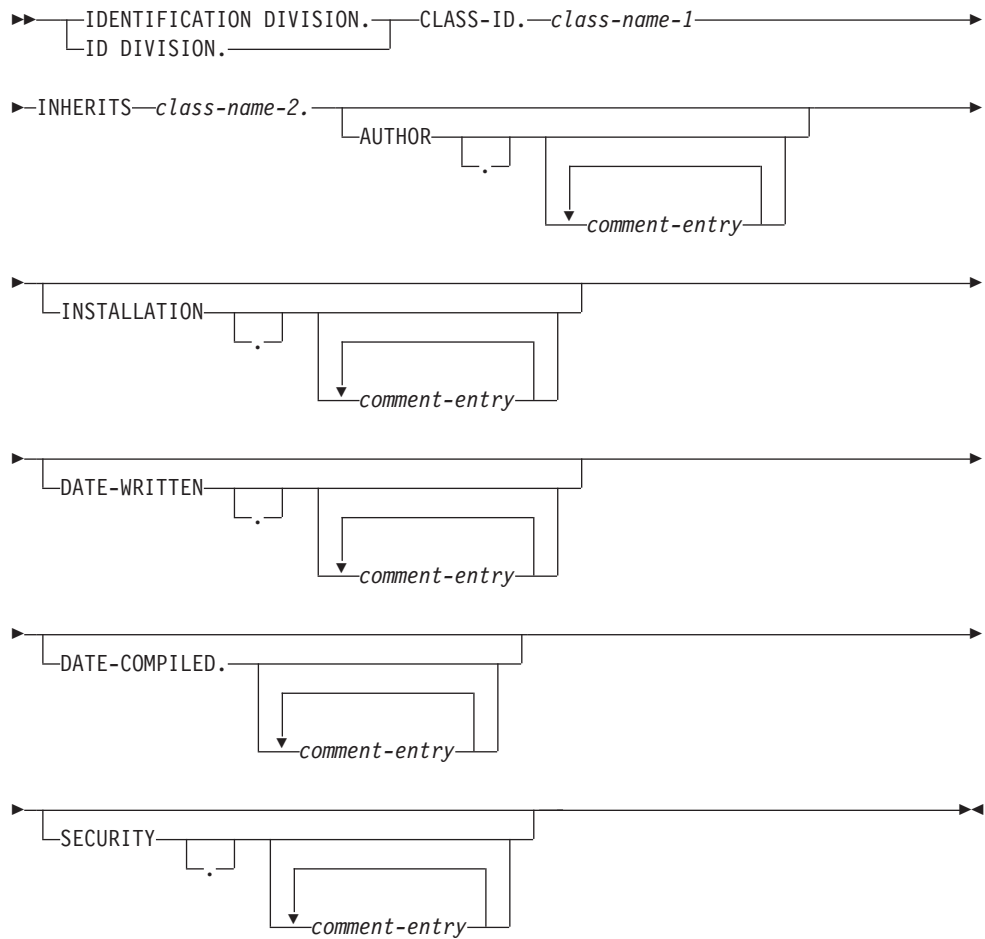
The following is the format for a program IDENTIFICATION DIVISION.

Format: program identification division



The following is the format for a class IDENTIFICATION DIVISION.

Format: class identification division



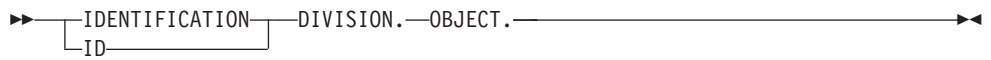
The following is the format for a factory IDENTIFICATION DIVISION.

Format: factory identification division



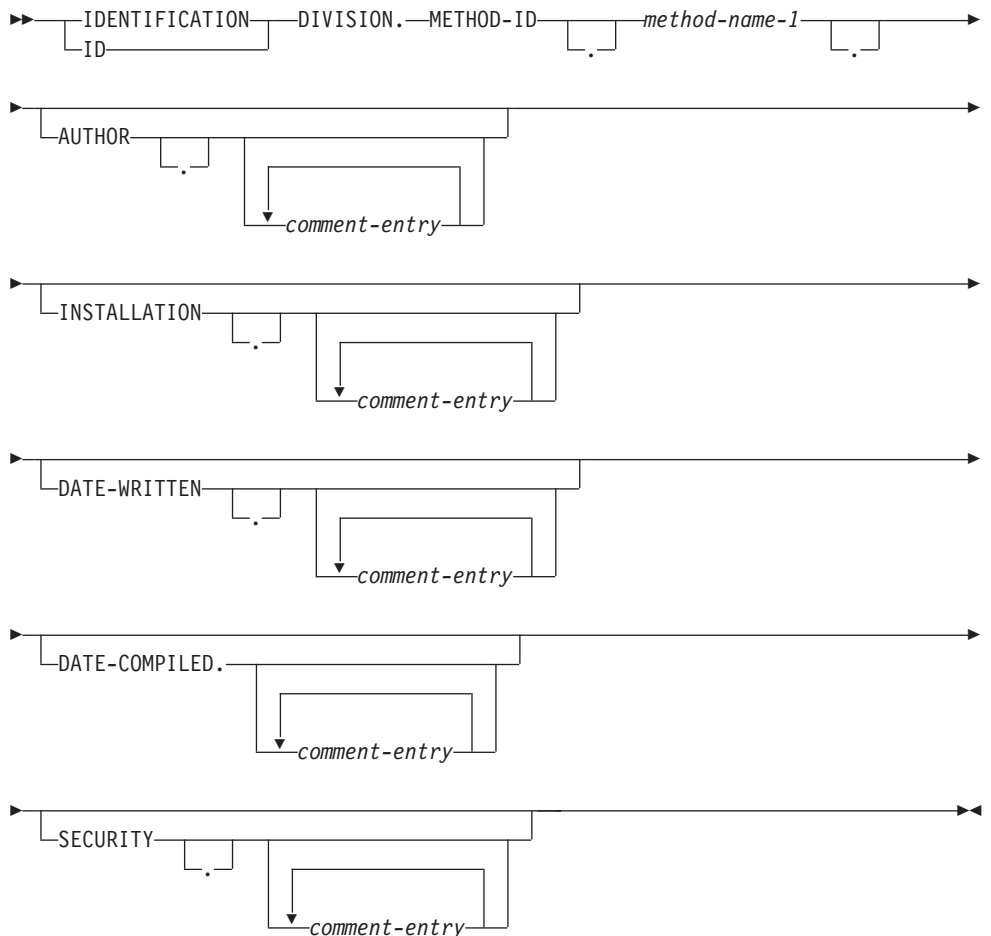
The following is the format for an object IDENTIFICATION DIVISION.

Format: object identification division



The following is the format for a method IDENTIFICATION DIVISION.

Format: method identification division



PROGRAM-ID paragraph

The PROGRAM-ID paragraph specifies the name by which the program is known and assigns selected program attributes to that program. It is required and must be the first paragraph in the identification division.

program-name

A user-defined word or alphanumeric literal, but not a figurative constant,

that identifies your program. It must follow the following rules of formation, depending on the setting of the PGMNAME compiler option:

PGMNAME (LONGUPPER)

If *program-name* is a user-defined word, it can be up to 30 characters in length.

If *program-name* is an alphanumeric literal, the literal can be up to 160 characters in length. The literal cannot be a figurative constant.

Only the hyphen, underscore, digits 0-9, and alphabetic characters are allowed in the name when the name is specified as a user-defined word.

At least one character must be alphabetic.

The hyphen cannot be the first or last character.

If *program-name* is an alphanumeric literal, the underscore character can be the first character.

External program-names are processed with alphabetic characters folded to uppercase.

PGMNAME (LONGMIXED)

program-name must be specified as a literal, which can be up to 160 characters in length. The literal cannot be a figurative constant.

Wherever alphabetic characters are allowed, you can use multibyte characters.

For information about the PGMNAME compiler option and how the compiler processes the names, see *PGMNAME* in the *COBOL for AIX Programming Guide*.

RECURSIVE

An optional clause that allows COBOL programs to be recursively reentered.

You can specify the RECURSIVE clause only on the outermost program of a compilation unit. Recursive programs cannot contain nested subprograms.

If the RECURSIVE clause is specified, *program-name* can be recursively reentered while a previous invocation is still active. If the RECURSIVE clause is not specified, an active program cannot be recursively reentered.

The working-storage section of a recursive program defines storage that is statically allocated and initialized on the first entry to a program and is available in a last-used state to any of the recursive invocations.

The local-storage section of a recursive program (as well as a nonrecursive program) defines storage that is automatically allocated, initialized, and deallocated on a per-invocation basis.

Internal file connectors that correspond to an FD in the file section of a recursive program are statically allocated. The status of internal file connectors is part of the last-used state of a program that persists across invocations.

The following language elements are not supported in a recursive program:

- ALTER
- GO TO without a specified procedure-name

- RERUN
- SEGMENT-LIMIT
- USE FOR DEBUGGING

Methods are always recursive by default. The RECURSIVE clause cannot be specified in the METHOD-ID paragraph.

COMMON

Specifies that the program named by *program-name* is contained (that is, nested) within another program and can be called from siblings of the common program and programs contained within them. The COMMON clause can be used only in nested programs. For more information about conventions for program names, see “Conventions for program-names” on page 84.

INITIAL

Specifies that when *program-name* is called, *program-name* and any programs contained (nested) within it are placed in their initial state.

A program is in the initial state:

- The first time the program is called in a run unit
- Every time the program is called, if it possesses the initial attribute
- The first time the program is called after the execution of a CANCEL statement that references the program or a CANCEL statement that references a program that directly or indirectly contains the program
- The first time the program is called after the execution of a CALL statement that references a program that possesses the initial attribute and that directly or indirectly contains the program

When a program is in the initial state, the following occur:

- The program's internal data contained in the working-storage section is initialized. If a VALUE clause is used in the description of the data item, the data item is initialized to the defined value. If a VALUE clause is not associated with a data item, the initial value of the data item is undefined.
- Files with internal file connectors associated with the program are not in the open mode.
- The control mechanisms for all PERFORM statements contained in the program are set to their initial states.
- An altered GO TO statement contained in the program is set to its initial state.

For the rules governing nonunique program names, see “Rules for program-names” on page 84.

CLASS-ID paragraph

The CLASS-ID paragraph specifies the name by which the class is known and assigns selected attributes to that class. It is required and must be the first paragraph in a class identification division.

class-name-1

A user-defined word that identifies the class. *class-name-1* can optionally have an entry in the REPOSITORY paragraph of the configuration section of the class definition.

INHERITS

A clause that defines *class-name-1* to be a subclass (or derived class) of *class-name-2* (the parent class). *class-name-1* cannot directly or indirectly inherit from *class-name-1*.

class-name-2

The name of a class inherited by *class-name-1*. You must specify *class-name-2* in the REPOSITORY paragraph of the configuration section of the class definition.

General rules

class-name-1 and *class-name-2* must conform to the normal rules of formation for a COBOL user-defined word, using single-byte characters.

See “REPOSITORY paragraph” on page 118 for details on specifying a class-name that is part of a Java package or for using non-COBOL naming conventions for class-names.

You cannot include a class definition in a sequence of programs or other class definitions in a single compilation group. Each class must be specified as a separate source file; that is, a class definition cannot be included in a batch compile.

Inheritance

Every method available on instances of a class is also available on instances of any subclass directly or indirectly derived from that class. A subclass can introduce new methods that do not exist in the parent or ancestor class and can override a method from the parent or ancestor class. When a subclass overrides an existing method, it defines a new implementation for that method, which replaces the inherited implementation.

The instance data of *class-name-1* is the instance data declared in *class-name-2* together with the data declared in the working-storage section of *class-name-1*. Note, however, that instance data is always private to the class that introduces it.

The semantics of inheritance are as defined by Java. All classes must be derived directly or indirectly from the `java.lang.Object` class.

Java supports single inheritance; that is, no class can inherit *directly* from more than one parent. Only one class-name can be specified in the INHERITS phrase of a class definition.

FACTORY paragraph

The factory IDENTIFICATION DIVISION introduces the factory definition, which is the portion of a class definition that defines the factory object of the class. A *factory object* is the single common object that is shared by all object instances of the class.

The factory definition contains factory data and factory methods.

OBJECT paragraph

The object IDENTIFICATION DIVISION introduces the object definition, which is the portion of a class definition that defines the instance objects of the class.

The object definition contains object data and object methods.

METHOD-ID paragraph

The METHOD-ID paragraph specifies the name by which a method is known and assigns selected attributes to that method. The METHOD-ID paragraph is required and must be the first paragraph in a method identification division.

method-name-1

An alphanumeric literal or national literal that contains the name of the method. The name must conform to the rules of formation for a Java method name. Method names are used directly, without translation. The method name is processed in a case-sensitive manner.

Method signature

The *signature* of a method consists of the name of the method and the number and types of the formal parameters to the method as specified in the procedure division USING phrase.

Method overloading, overriding, and hiding

COBOL methods can be *overloaded*, *overridden*, or *hidden*, based on the rules of the Java language.

Method overloading

Method names that are defined for a class are not required to be unique. (The set of methods defined for a class includes the methods introduced by the class definition and the methods inherited from parent classes.)

Method names defined for a class must have unique signatures. Two methods defined for a class and that have the same name but different signatures are said to be *overloaded*.

The type of the method return value, if any, is not included in the method signature.

A class must not define two methods with the same signature but different return value types, or with the same signature but where one method specifies a return value and the other does not.

The rules for overloaded method definitions and resolution of overloaded method invocations are based on the corresponding rules for Java.

Method overriding (for instance methods)

An instance method in a subclass *overrides* an instance method with the same name that is inherited from a parent class if the two methods have the same signature.

When a method overrides an instance method defined in a parent class, the presence or absence of a method return value (the procedure division RETURNING data-name) must be consistent in the two methods. Further, when method return values are specified, the return values in the overridden method and the overriding method must have identical data types.

An instance method must not override a factory method in a COBOL parent class, or a static method in a Java parent class.

Method hiding (for factory methods)

A factory method is said to *hide* any and all methods with the same signature in the superclasses of the method definition that would otherwise be accessible. A factory method must not hide an instance method.

Optional paragraphs

These optional paragraphs in the identification division can be omitted:

AUTHOR

Name of the author of the program.

INSTALLATION

Name of the company or location.

DATE-WRITTEN

Date the program was written.

DATE-COMPILED

The DATE-COMPILED paragraph provides the compilation date in the source listing. If a comment-entry is specified, the entire entry is replaced with the current date, even if the entry spans lines. If the comment entry is omitted, the compiler adds the current date to the line on which DATE-COMPILED is printed. For example:

DATE-COMPILED. 06/30/10.

SECURITY

Level of confidentiality of the program.

The *comment-entry* in any of the optional paragraphs can be any combination of characters from the character set of the computer. The comment-entry is written in Area B on one or more lines.

Comment-entries serve only as documentation; they do not affect the meaning of the program. A hyphen in the indicator area (column 7) is not permitted in comment-entries.

You can include multibyte as well as single-byte characters from an EUC, UTF-8, or DBCS code page in comment-entries in the identification division of your program. Multiple lines are allowed in a comment-entry that contains multibyte characters.

Part 4. Environment division

Chapter 15. Configuration section.	107
SOURCE-COMPUTER paragraph	108
OBJECT-COMPUTER paragraph	108
SPECIAL-NAMES paragraph	110
ALPHABET clause	113
SYMBOLIC CHARACTERS clause	116
CLASS clause	116
CURRENCY SIGN clause	117
DECIMAL-POINT IS COMMA clause	118
REPOSITORY paragraph	118
General rules	120
Identifying and referencing a class	120
Chapter 16. Input-Output section	121
FILE-CONTROL paragraph.	122
SELECT clause	126
ASSIGN clause	126
Assignment name for user-defined words and literals.	128
Assignment name for data-names and environment variables	130
File concatenation	131
RESERVE clause	133
ORGANIZATION clause	133
File organization	133
Sequential organization	134
Indexed organization	134
Relative organization	134
Line-sequential organization	134
Language elements treated as comments	135
PADDING CHARACTER clause	136
RECORD DELIMITER clause	136
ACCESS MODE clause	137
File organization and access modes	137
Access modes	138
Relationship between data organizations and access modes	138
RECORD KEY clause	139
ALTERNATE RECORD KEY clause	140
RELATIVE KEY clause	141
PASSWORD clause	141
FILE STATUS clause	141
I-O-CONTROL paragraph	143
RERUN clause	145
SAME AREA clause	146
SAME RECORD AREA clause	146
SAME SORT AREA clause	146
SAME SORT-MERGE AREA clause	147
MULTIPLE FILE TAPE clause	147
APPLY WRITE-ONLY clause	147

Chapter 15. Configuration section

The configuration section is an optional section for programs and classes, and can describe the computer environment on which the program or class is compiled and executed.

Program configuration section

The configuration section can be specified only in the environment division of the outermost program of a COBOL source program.

You should not specify the configuration section in a program that is contained within another program. The entries specified in the configuration section of a program apply to any program contained within that program.

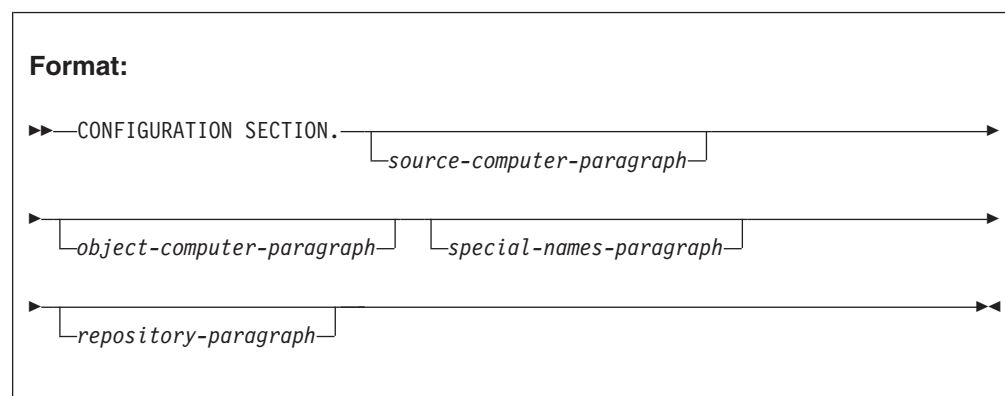
Class configuration section

Specify the configuration section in the environment division of a class definition. The repository paragraph can be specified in the environment division of a class definition.

Entries in a class configuration section apply to the entire class definition, including all methods introduced by that class.

Method configuration section

The input-output section can be specified in a method configuration section. The entries apply only to the method in which the configuration section is specified.



The configuration section can:

- Relate IBM-defined environment-names to user-defined mnemonic names
- Specify the collating sequence
- Specify a currency sign value, and the currency symbol used in the PICTURE clause to represent the currency sign value
- Exchange the functions of the comma and the period in PICTURE clauses and numeric literals
- Relate alphabet-names to character sets or collating sequences
- Specify symbolic characters
- Relate class-names to sets of characters

SEGMENT-LIMIT IS

The SEGMENT-LIMIT clause is syntax checked but has no effect on the execution of the program.

priority-number

An integer ranging from 1 through 49. All sections with priority-numbers 0 through 49 are fixed permanent segments. See “Procedures” on page 256 for a description of priority-numbers and segmentation support.

All of the OBJECT-COMPUTER paragraph is syntax checked, but only the PROGRAM COLLATING SEQUENCE clause has an effect on the execution of the program.

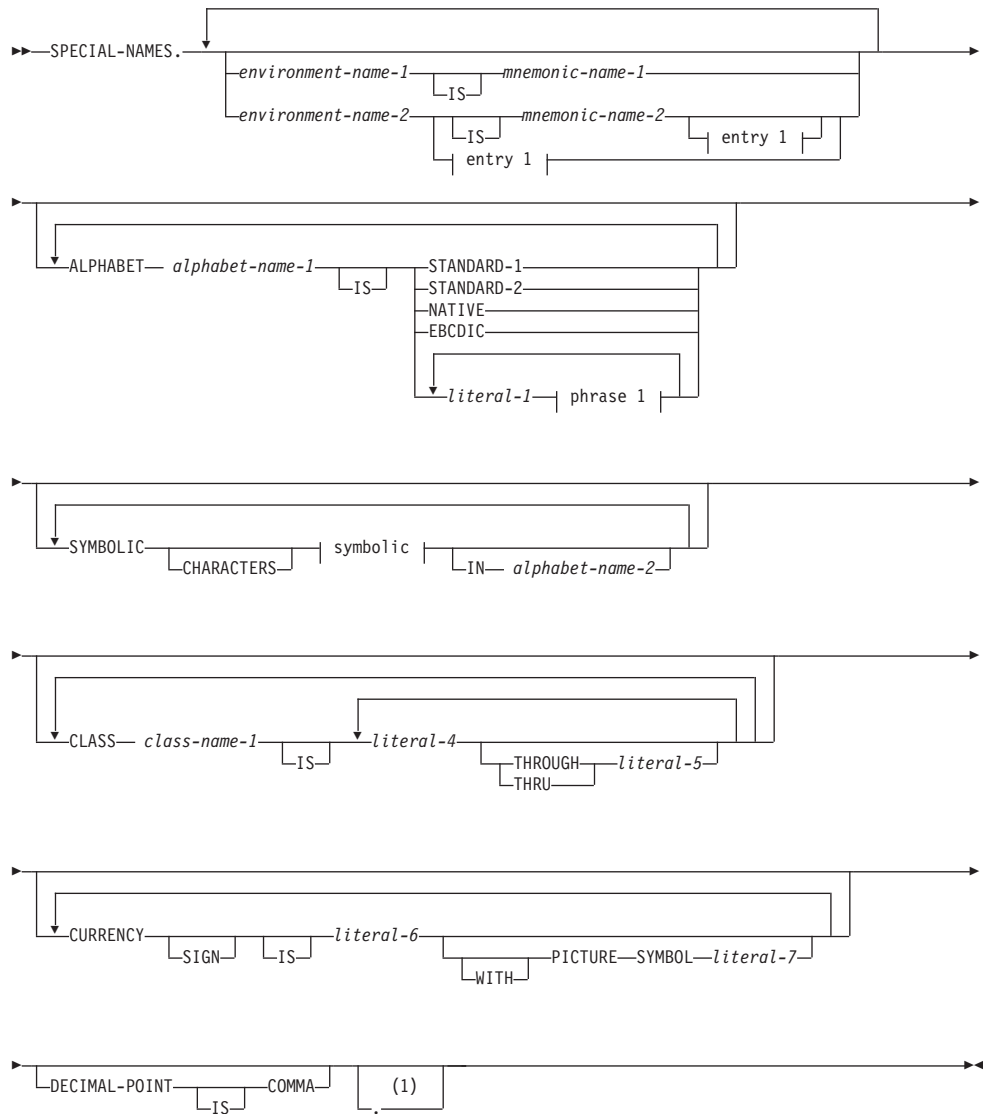
SPECIAL-NAMES paragraph

The SPECIAL-NAMES paragraph:

- Relates IBM-specified environment-names to user-defined mnemonic-names
- Relates alphabet-names to character sets or collating sequences
- Specifies symbolic characters
- Relates class names to sets of characters
- Specifies one or more currency sign values and defines a picture symbol to represent each currency sign value in PICTURE clauses
- Specifies that the functions of the comma and decimal point are to be interchanged in PICTURE clauses and numeric literals

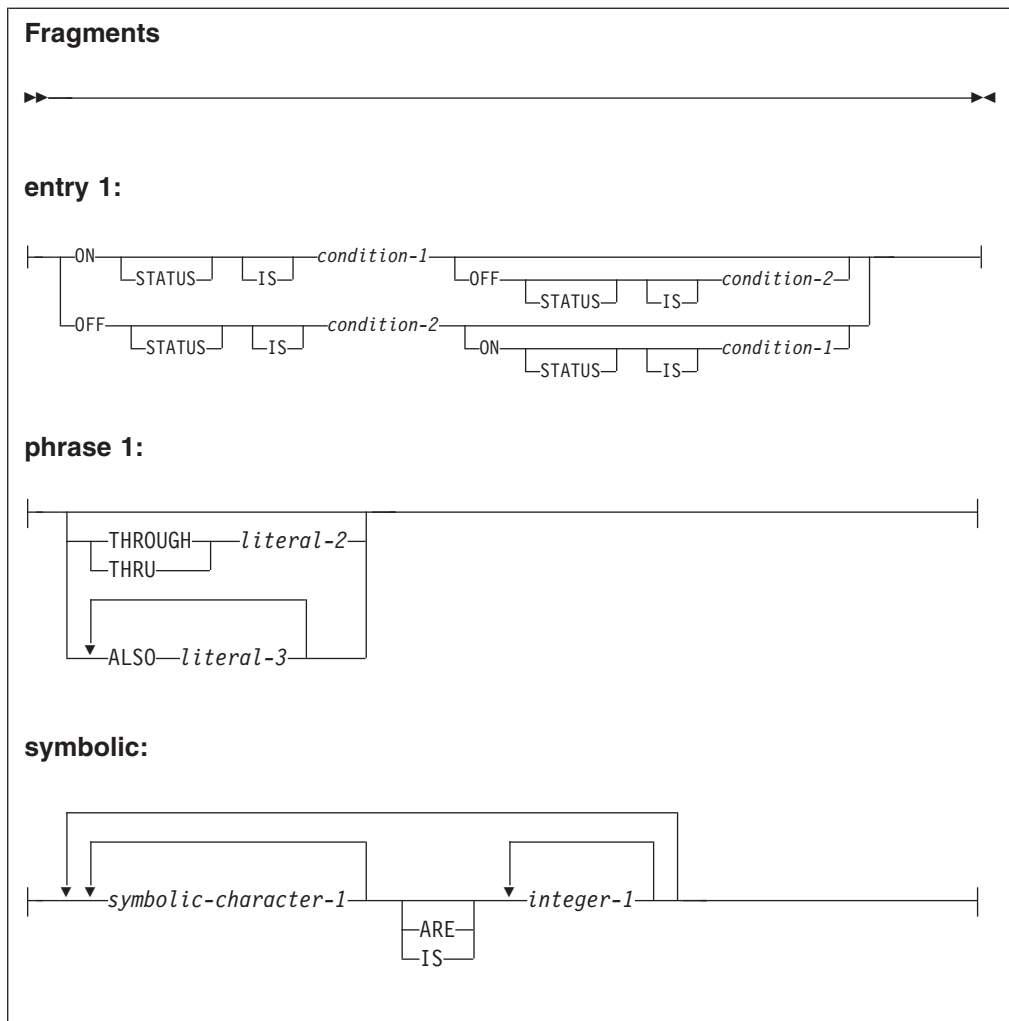
The clauses in the SPECIAL-NAMES paragraph can appear in any order.

Format: SPECIAL-NAMES paragraph



Notes:

- 1 This separator period is optional when no clauses are selected. If you use any clauses, you must code the period after the last clause.



When the source code page is a multibyte code page, the following clauses cannot be specified:

- ALPHABET clause
- CLASS clause
- SYMBOLIC characters clause

environment-name-1

System devices or standard system actions taken by the compiler.

Valid specifications for *environment-name-1* are shown in the following table.

Table 5. Meanings of environment names

<i>environment-name-1</i>	Meaning	Allowed in
SYSIN SYSIPT	System logical input unit	ACCEPT
SYSOUT SYSLIST SYSLST	System logical output unit	DISPLAY
SYSPUNCH SYSPCH	System punch device	DISPLAY

Table 5. Meanings of environment names (continued)

<i>environment-name-1</i>	Meaning	Allowed in
CONSOLE	Console	ACCEPT and DISPLAY
C01 through C12	Skip to channel 1 through channel 12, respectively	WRITE ADVANCING With C01 through C12, one line is advanced.
CSP	Suppress spacing	WRITE ADVANCING
S01 through S05	Pocket select 1 through 5 on punch devices	WRITE ADVANCING With S01 through S05, one line is advanced.
AFP-5A	Advanced Function Printing	WRITE ADVANCING

environment-name-2

A 1-byte user-programmable status indicator (UPSI) switch. Valid specifications for *environment-name-2* are UPSI-0 through UPSI-7.

mnemonic-name-1 , *mnemonic-name-2*

mnemonic-name-1 and *mnemonic-name-2* follow the rules of formation for user-defined names. *mnemonic-name-1* can be used in ACCEPT, DISPLAY, and WRITE statements. *mnemonic-name-2* can be referenced only in the SET statement. *mnemonic-name-2* can qualify *condition-1* or *condition-2* names.

Mnemonic-names and environment-names need not be unique. If you choose a mnemonic-name that is also an environment-name, its definition as a mnemonic-name will take precedence over its definition as an environment-name.

ON STATUS IS, OFF STATUS IS

UPSI switches process special conditions within a program, such as year-beginning or year-ending processing. For example, at the beginning of the procedure division, an UPSI switch can be tested; if it is ON, the special branch is taken. (See “Switch-status condition” on page 280.)

condition-1, *condition-2*

Condition-names follow the rules for user-defined names. At least one character must be alphabetic. The value associated with the condition-name is considered to be alphanumeric. A condition-name can be associated with the on status or off status of each UPSI switch specified.

In the procedure division, the UPSI switch status is tested through the associated condition-name. Each condition-name is the equivalent of a level-88 item; the associated mnemonic-name, if specified, is considered the conditional variable and can be used for qualification.

Condition-names specified in the SPECIAL-NAMES paragraph of a containing program can be referenced in any contained program.

ALPHABET clause

The ALPHABET clause provides a means of relating an alphabet-name to a specified character code set or collating sequence.

The related character code set or collating sequence can be used for alphanumeric data, but not for national data.

ALPHABET *alphabet-name-1* IS

alphabet-name-1 specifies a *collating sequence* when used in:

- The PROGRAM COLLATING SEQUENCE clause of the object-computer paragraph
- The COLLATING SEQUENCE phrase of the SORT or MERGE statement

alphabet-name-1 specifies a character code set when used in:

- The FD entry CODE-SET clause
- The SYMBOLIC CHARACTERS clause

You cannot specify the ALPHABET clause if the source code page in effect is a multibyte code page. For details, see *Specifying the collating sequence* in the *COBOL for AIX Programming Guide*.

STANDARD-1

Specifies that the collating sequence is based on the binary code values of the characters, ignoring the locale setting.

STANDARD-2

Specifies that the collating sequence is based on the binary code values of the characters, ignoring the locale setting.

NATIVE

Specifies the native character code set. If the ALPHABET clause is omitted, the *alphabet-name* is associated with the ASCII, UTF-8, or EUC character set indicated by the locale in effect.

EBCDIC

Specifies the EBCDIC character set.

literal-1, literal-2, literal-3

Specifies that the collating sequence for alphanumeric data is determined by the program, according to the following rules:

- The order in which literals appear specifies the ordinal number, in ascending sequence, of the characters in this collating sequence.
- Each numeric literal specified must be an unsigned integer.
- Each numeric literal must have a value that corresponds to a valid ordinal position within the collating sequence in effect.
See Appendix D, "EBCDIC and ASCII collating sequences," on page 585 for the ordinal numbers for characters in the single-byte EBCDIC and ASCII collating sequences.
- Each character in an alphanumeric literal represents that actual character in the character set. (If the alphanumeric literal contains more than one character, each character, starting with the leftmost, is assigned a successively ascending position within this collating sequence.)
- Any characters that are not explicitly specified assume positions in this collating sequence higher than any of the explicitly specified characters. The relative order within the collating sequence of these unspecified characters is their relative order in the collating sequence indicated by the COLLSEQ compiler option.
- Within one *alphabet-name* clause, a given character must not be specified more than once.

- Each alphanumeric literal associated with a THROUGH or ALSO phrase must be one character in length.
- When the THROUGH phrase is specified, the contiguous characters in the native character set beginning with the character specified by *literal-1* and ending with the character specified by *literal-2* are assigned successively ascending positions in this collating sequence.

This sequence can be either ascending or descending within the original native character set. That is, if "Z" THROUGH "A" is specified, the ascending values, left-to-right, for the uppercase letters are:

ZYXWVUTSRQPONMLKJIHGFEDCBA

- When the ALSO phrase is specified, the characters specified as *literal-1*, *literal-3*, ... are assigned to the same position in this collating sequence. For example, if you specify:
"D" ALSO "N" ALSO "%"
the characters D, N, and % are all considered to be in the same position in the collating sequence.
- When the ALSO phrase is specified and *alphabet-name-1* is referenced in a SYMBOLIC CHARACTERS clause, only *literal-1* is used to represent the character in the character set.
- The character that has the highest ordinal position in this collating sequence is associated with the figurative constant HIGH-VALUE. If more than one character has the highest position because of specification of the ALSO phrase, the last character specified (or defaulted to when any characters are not explicitly specified) is considered to be the HIGH-VALUE character for procedural statements such as DISPLAY and as the sending field in a MOVE statement. (If the ALSO phrase example given above were specified as the high-order characters of this collating sequence, the HIGH-VALUE character would be %.)
- The character that has the lowest ordinal position in this collating sequence is associated with the figurative constant LOW-VALUE. If more than one character has the lowest position because of specification of the ALSO phrase, the first character specified is the LOW-VALUE character. (If the ALSO phrase example given above were specified as the low-order characters of the collating sequence, the LOW-VALUE character would be D.)

When *literal-1*, *literal-2*, or *literal-3* is specified, the alphabet-name must not be referred to in a CODE-SET clause (see "CODE-SET clause" on page 180).

literal-1, *literal-2*, and *literal-3* must be alphanumeric or numeric literals. All must have the same category. A floating-point literal, a national literal, a DBCS literal, or a symbolic-character figurative constant must not be specified.

SYMBOLIC CHARACTERS clause

SYMBOLIC CHARACTERS *symbolic-character-1*

Provides a means of specifying one or more symbolic characters.

symbolic-character-1 is a user-defined word and must contain at least one alphabetic character. The same *symbolic-character* can appear only once in a SYMBOLIC CHARACTERS clause.

You cannot use the SYMBOLIC CHARACTERS clause if the source text code page is a multibyte code page.

The SYMBOLIC CHARACTERS clause is applicable only to single-byte character sets. Each character represented is an alphanumeric character.

The internal representation of *symbolic-character-1* is the internal representation of the character that is represented in the specified character set. The following rules apply:

- The relationship between each *symbolic-character-1* and the corresponding *integer-1* is by their position in the SYMBOLIC CHARACTERS clause. The first *symbolic-character-1* is paired with the first *integer-1*; the second *symbolic-character-1* is paired with the second *integer-1*; and so forth.
- There must be a one-to-one correspondence between occurrences of *symbolic-character-1* and occurrences of *integer-1* in a SYMBOLIC CHARACTERS clause.
- If the IN phrase is specified, *integer-1* specifies the ordinal position of the character that is represented in the character set named by *alphabet-name-2*. This ordinal position must exist.
- If the IN phrase is not specified, *symbolic-character-1* represents the character whose ordinal position in the native character set is specified by *integer-1*.

Ordinal positions are numbered starting from 1.

CLASS clause

You cannot specify the CLASS clause if the source code page in effect is a multibyte code page.

CLASS *class-name-1* IS

Provides a means for relating a name to the specified set of characters listed in that clause. *class-name-1* can be referenced only in a class condition. The characters specified by the values of the literals in this clause define the exclusive set of characters of which this class consists.

literal-4, literal-5

Must be category numeric or alphanumeric, and both must be of the same category.

If numeric, *literal-4* and *literal-5* must be unsigned integers and must have a value that is greater than or equal to 1 and less than or equal to the number of characters in the alphabet specified. Each number corresponds to the ordinal position of each character in the single-byte EBCDIC or ASCII collating sequence. They cannot be specified as floating-point literals or as DBCS literals.

If alphanumeric, *literal-4* and *literal-5* are an actual single-byte EBCDIC or single-byte ASCII character.

literal-4 and *literal-5* must not specify a symbolic-character figurative constant. If the value of the alphanumeric literal contains multiple characters, each character in the literal is included in the set of characters identified by class-name.

If the alphanumeric literal is associated with a THROUGH phrase, the literal must be one character in length.

THROUGH, THRU

THROUGH and THRU are equivalent. If THROUGH is specified, class-name includes those characters that begin with the value of *literal-4* and that end with the value of *literal-5*. In addition, the characters specified by a THROUGH phrase can be in either ascending or descending order.

CURRENCY SIGN clause

The CURRENCY SIGN clause affects numeric-edited data items whose PICTURE character-strings contain a *currency symbol*. A currency symbol represents a *currency sign value* that is:

- Inserted in such data items when they are used as receiving items
- Removed from such data items when they are used as sending items for a numeric or numeric-edited receiver

Typically, currency sign values identify the monetary units stored in a data item. For example: '\$', 'EUR', 'CHF', 'JPY', 'HK\$', 'HKD', or X'9F' (hexadecimal code point in some EBCDIC code pages for €, the Euro currency sign). For details on programming techniques for handling the Euro, see *Using currency signs* in the *COBOL for AIX Programming Guide*.

The CURRENCY SIGN clause specifies a currency sign value and the currency symbol used to represent that currency sign value in a PICTURE clause.

The SPECIAL-NAMES paragraph can contain multiple CURRENCY SIGN clauses. Each CURRENCY SIGN clause must specify a different currency symbol. Unlike all other PICTURE clause symbols, currency symbols are case sensitive. For example, 'D' and 'd' specify different currency symbols.

CURRENCY SIGN IS *literal-6*

literal-6 must be an alphanumeric literal. *literal-6* must not be a figurative constant or a null-terminated literal. *literal-6* must not contain a multibyte character.

If the PICTURE SYMBOL phrase is not specified, *literal-6*:

- Specifies both a currency sign value and the currency symbol for this currency sign value
- Must be a single character
- Must not be any of the following:
 - Digits 0 through 9
 - Alphabetic characters A, B, C, D, E, G, N, P, R, S, V, X, Z, their lowercase equivalents, or the space
 - Special characters + - , . * / ; () " = ' (plus sign, minus sign, comma, period, asterisk, slash, semicolon, left parenthesis, right parenthesis, quotation mark, equal sign, apostrophe)

- Can be one of the following lowercase alphabetic characters: f, h, i, j, k, l, m, o, q, t, u, w, y

If the PICTURE SYMBOL phrase is specified, *literal-6*:

- Specifies a currency sign value. *literal-7* in the PICTURE SYMBOL phrase specifies the currency symbol for this currency sign value.
- Can consist of one or more characters.
- Must not contain any of the following:
 - Digits 0 through 9
 - Special characters + - . ,

PICTURE SYMBOL *literal-7*

Specifies a currency symbol that can be used in a PICTURE clause to represent the currency sign value specified by *literal-6*.

literal-7 must be an alphanumeric literal consisting of one single-byte character. *literal-7* must not be any of the following:

- A figurative constant
- Digits 0 through 9
- Alphabetic characters A, B, C, D, E, G, N, P, R, S, V, X, Z, their lowercase equivalents, or the space
- Special characters + - , . * / ; () " = '

If the CURRENCY SIGN clause is specified, the CURRENCY and NOCURRENCY compiler options are ignored. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency sign value and currency symbol. For more information about the CURRENCY and NOCURRENCY compiler options, see *CURRENCY* in the *COBOL for AIX Programming Guide*.

DECIMAL-POINT IS COMMA clause

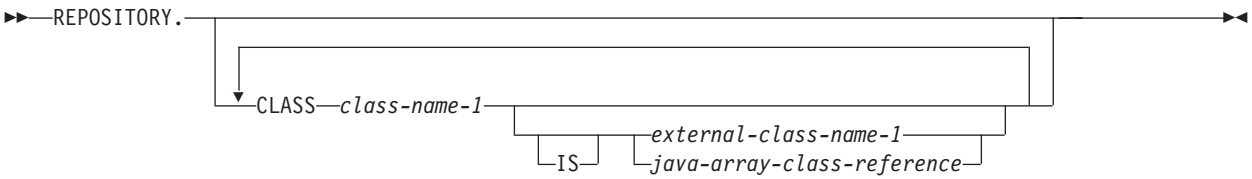
DECIMAL-POINT IS COMMA

Exchanges the functions of the period and the comma in PICTURE character-strings and in numeric literals.

REPOSITORY paragraph

The REPOSITORY paragraph is used in a program or class definition to identify all the object-oriented classes that are intended to be referenced in that program or class definition. Optionally, the REPOSITORY paragraph defines associations between class-names and external class-names.

Format: REPOSITORY paragraph



class-name-1

A user-defined word that identifies the class.

external-class-name-1

An alphanumeric literal containing a name that enables a COBOL program to define or access classes with class-names that are defined using Java rules of formation.

The name must conform to the rules of formation for a fully qualified Java class-name. If the class is part of a Java package, *external-class-name-1* must specify the fully qualified name of the package, followed by a period, followed by the simple name of the Java class.

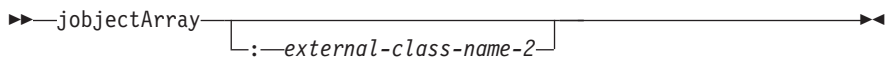
See *Java Language Specification, Third Edition*, by Gosling et al., for Java class-name formation rules.

java-array-class-reference

A reference that enables a COBOL program to access a class that represents an array object, where the elements of the array are themselves objects.

java-array-class-reference must be an alphanumeric literal with content in the following format:

Format



jobjectArray

Specifies a Java object array class.

: A required separator when *external-class-name-2* is specified. The colon must not be preceded or followed by space characters.

external-class-name-2

The external class-name of the type of the elements of the array. *external-class-name-2* must follow the same rules of formation as *external-class-name-1*.

When the repository entry specifies **jobjectArray** without the colon separator and *external-class-name-2*, the elements of the object array are of type `java.lang.Object`.

General rules

1. All referenced class-names must have an entry in the repository paragraph of the COBOL program or class definition that contains the reference. You can specify a given class-name only once in a given repository paragraph.
2. In program definitions, the repository paragraph can be specified only in the outermost program.
3. The repository paragraph of a COBOL class definition can optionally contain an entry for the name of the class itself, but this entry is not required. Such an entry can be used to specify an external class-name that uses non-COBOL characters or that specifies a fully package-qualified class-name when a COBOL class is to be part of a Java package.
4. Entries in a class repository paragraph apply to the entire class definition, including all methods introduced by that class. Entries in a program repository paragraph apply to the entire program, including its contained programs.

Identifying and referencing a class

An external-class-name is used to identify and reference a given class from outside the class definition that defines the class. The external class-name is determined by using the contents of *external-class-name-1*, *external-class-name-2*, or *class-name-1* (as specified in the repository paragraph of a class), as described below:

1. *external-class-name-1* and *external-class-name-2* are used directly, without translation. They are processed in a case-sensitive manner.
2. *class-name-1* is used if *external-class-name-1* or *java-array-class-reference* is not specified. To create an external name that identifies the class and conforms to Java rules of formation, *class-name-1* is processed as follows:
 - The name is converted to uppercase.
 - Hyphens are translated to zero.
 - Underscores are not translated.
 - If the first character of the name is a digit, it is converted as follows:
 - Digits 1 through 9 are changed to A through I.
 - 0 is changed to J.

The class can be implemented in Java or COBOL.

When referencing a class that is part of a Java package, *external-class-name-1* must be specified and must give the fully qualified Java class-name.

For example, the repository entry

Repository.

Class JavaException is "java.lang.Exception"

defines local class-name JavaException for referring to the fully qualified external-class-name "java.lang.Exception."

When defining a COBOL class that is to be part of a Java package, specify an entry in the repository paragraph of that class itself, giving the full Java package-qualified name as the external class-name.

Chapter 16. Input-Output section

The input-output section of the environment division contains two paragraphs:

- FILE-CONTROL paragraph
- I-O-CONTROL paragraph

The exact contents of the input-output section depend on the file organization and access methods used. See “ORGANIZATION clause” on page 133 and “ACCESS MODE clause” on page 137.

Program input-output section

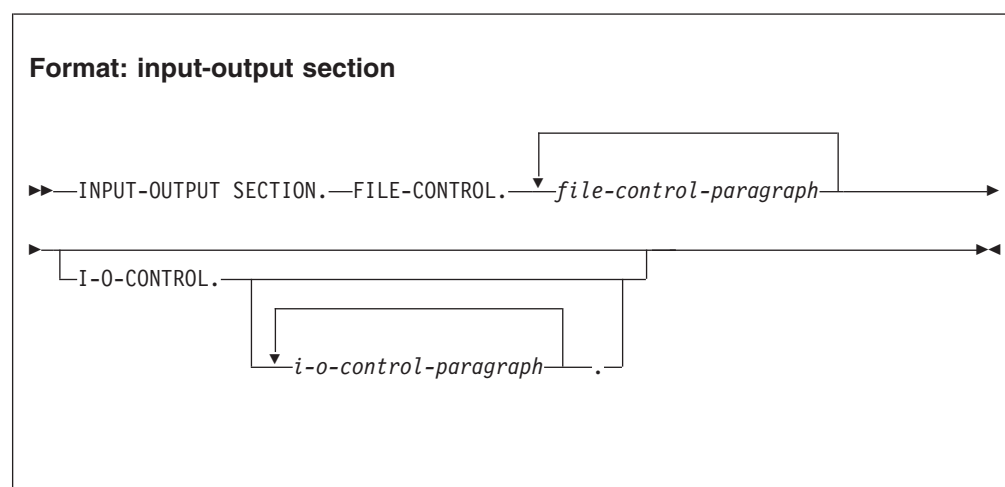
The same rules apply to program and method I-O sections.

Class input-output section

The input-output section is not valid for class definitions.

Method input-output section

The same rules apply to program and method I-O sections.



FILE-CONTROL

The keyword FILE-CONTROL identifies the file-control paragraph. This keyword can appear only once, at the beginning of the FILE-CONTROL paragraph. It must begin in Area A and be followed by a separator period.

The keyword FILE-CONTROL and the period can be omitted if no *file-control-paragraph* is specified and there are no files defined in the program.

file-control-paragraph

Names the files and associates them with the external files.

Must begin in Area B with a SELECT clause. It must end with a separator period. See “FILE-CONTROL paragraph” on page 122.

file-control-paragraph can be omitted if there are no files defined in the program, even if the FILE-CONTROL keyword is specified.

I-O-CONTROL

The keyword I-O-CONTROL identifies the I-O-CONTROL paragraph.

i-o-control-paragraph

Specifies information needed for efficient transmission of data between the external file and the COBOL program. The series of entries must end with a separator period. See “I-O-CONTROL paragraph” on page 143.

FILE-CONTROL paragraph

The FILE-CONTROL paragraph associates each file in the COBOL program with an external file, and specifies file organization, access mode, and other information.

The following are the formats for the FILE-CONTROL paragraph:

- Sequential file entries
- Indexed file entries
- Relative file entries
- Line-sequential file entries

The table below lists the different type of files available to programs and methods.

Table 6. Types of files

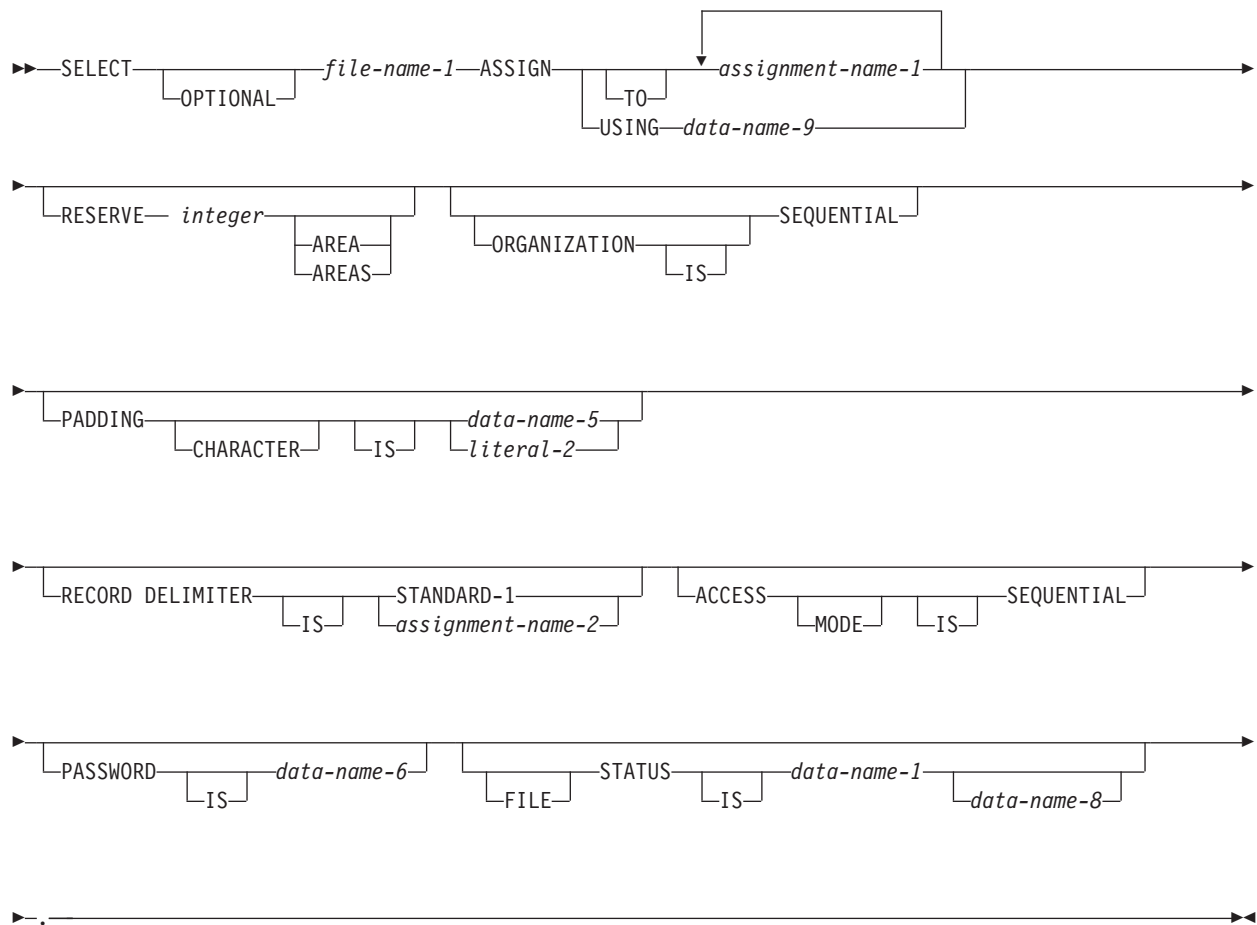
File organization	File system
Sequential	VSAM ¹ , SdU, SFS, STL, RSD ² , DB2
Relative	VSAM ¹ , SdU, SFS, STL, DB2
Indexed	VSAM ¹ , SdU, SFS, STL, DB2
Line sequential	Native
1. VSAM is a generic reference for either the SdU or SFS file system.	
2. The RSD file system supports only fixed-length record sequential files.	

The FILE-CONTROL paragraph begins with the word FILE-CONTROL followed by a separator period. It must contain one and only one entry for each file described in an FD or SD entry in the data division.

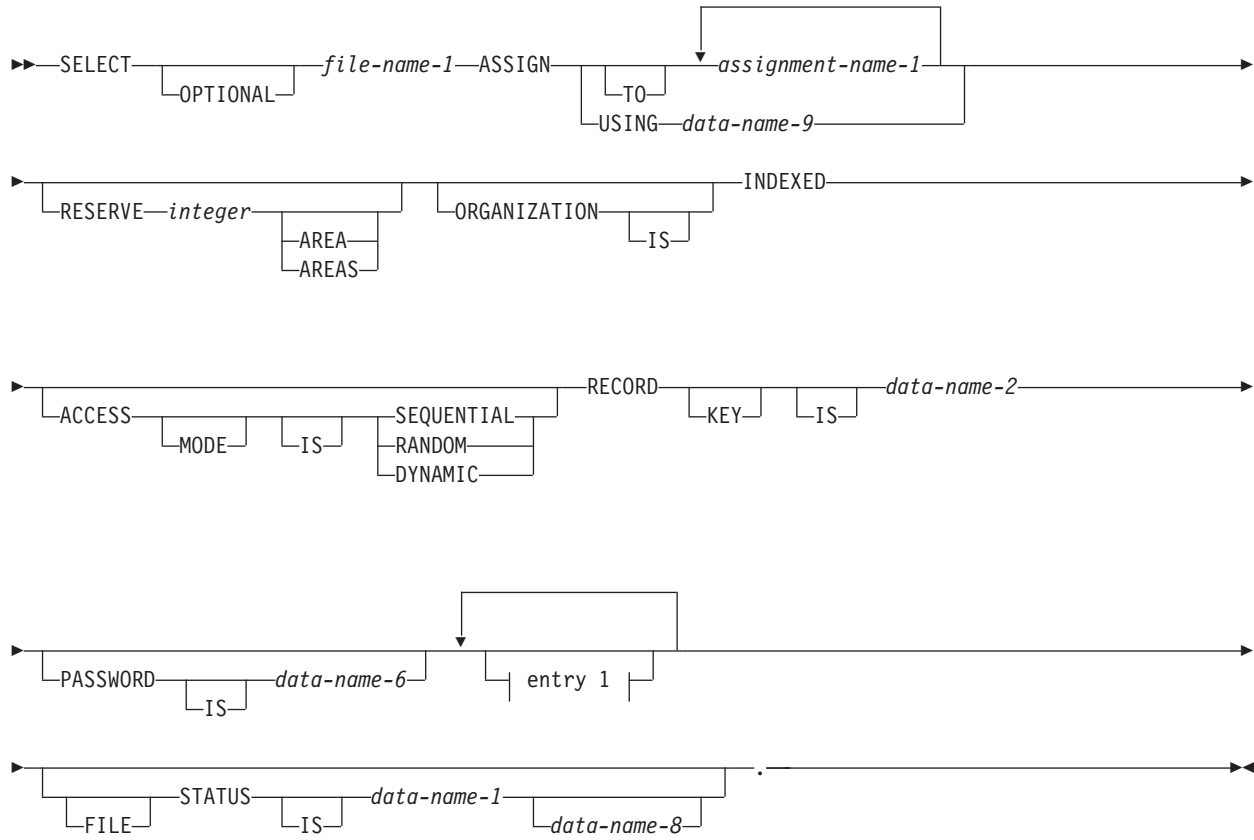
Within each entry, the SELECT clause must appear first. The other clauses can appear in any order.

The underscore is allowed in the *external-file-name* component of *assignment-name-1*.

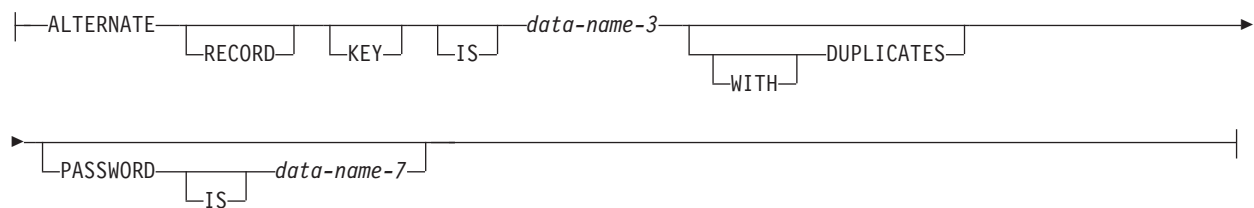
Format 1: sequential-file-control-entry



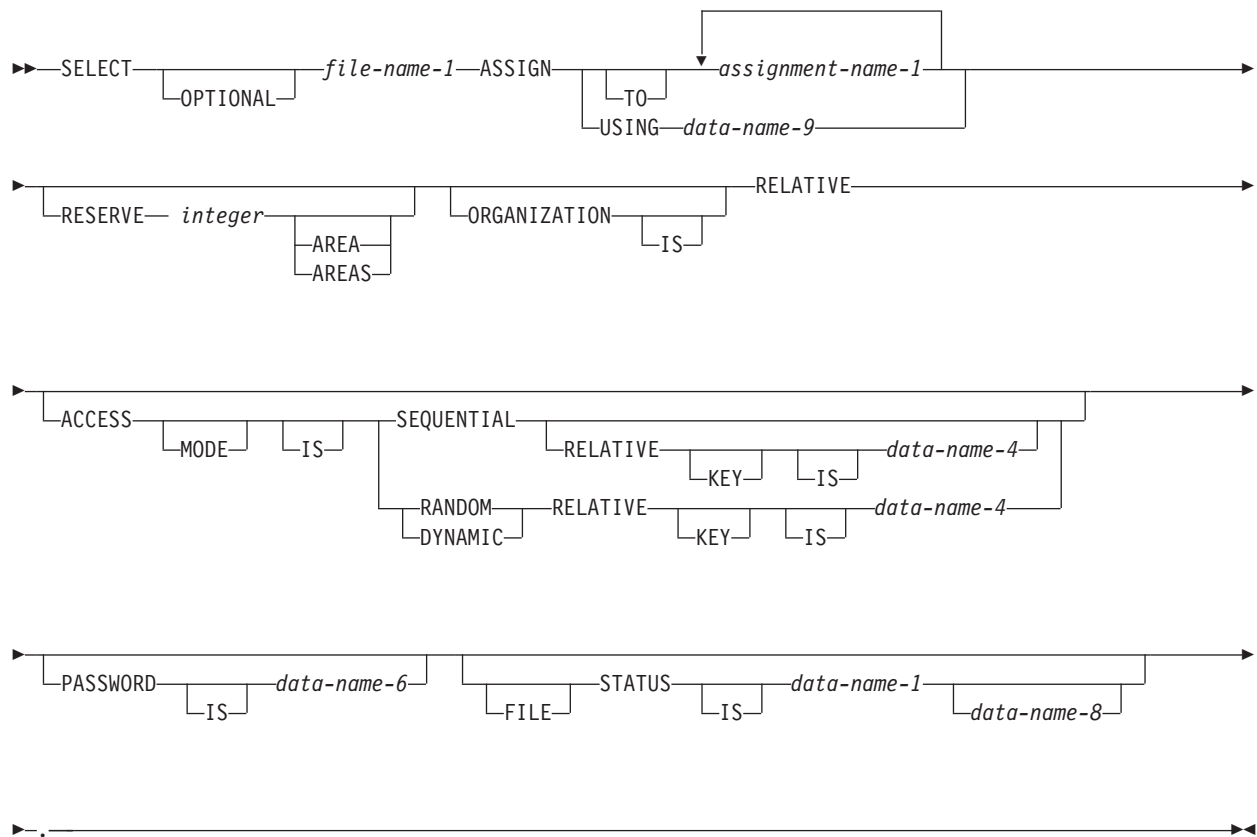
Format 2: indexed-file-control-entry



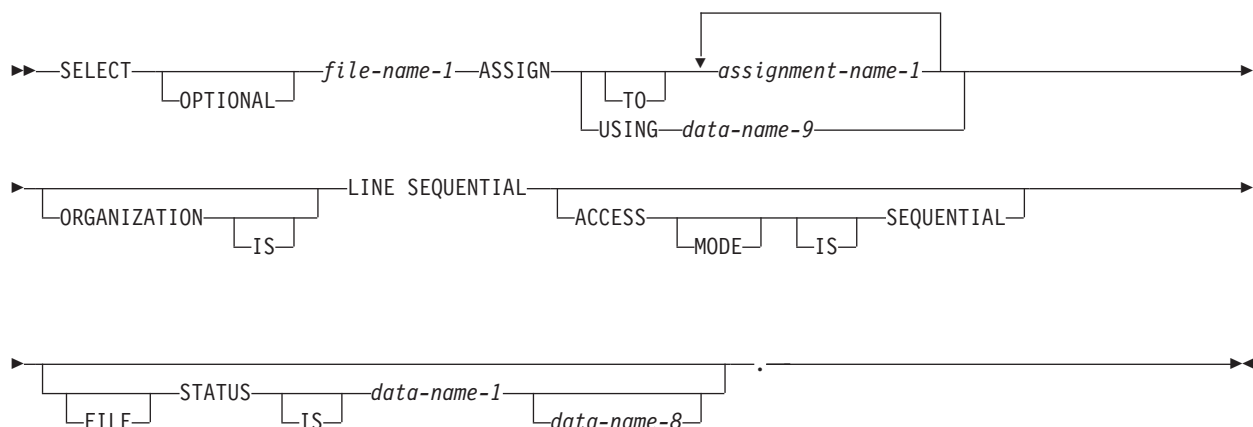
entry 1:



Format 3: relative-file-control-entry



Format 4: line-sequential-file-control-entry



SELECT clause

The `SELECT` clause identifies a file in the COBOL program to be associated with an external file.

SELECT OPTIONAL

Can be specified only for files opened in the input, I-O, or extend mode. You must specify `SELECT OPTIONAL` for those input files that are not necessarily available each time the object program is executed. For more information, see “OPEN statement notes” on page 393.

file-name-1

Must be identified by an FD or SD entry in the data division. A file-name must conform to the rules for a COBOL user-defined name, must contain at least one alphabetic character, and must be unique within this program.

When *file-name-1* specifies a sort or a merge file, only the `ASSIGN` clause can follow the `SELECT` clause.

If the file connector referenced by *file-name-1* is an external file connector, all file-control entries in the run unit that reference this file connector must have the same specification for the `OPTIONAL` phrase.

ASSIGN clause

The `ASSIGN` clause associates an internal file-name in a COBOL program with a system file-name.

assignment-name-1

Can be specified either as a user-defined word or as an alphanumeric literal. Both forms consist of up to three components, separated by hyphens. From left to right:

1. An optional comment string
2. An optional file-system ID

3. A required external file-name if *assignment-name-1* was specified as a user-defined word; a required system file-name if *assignment-name-1* was specified as a literal

For further details about how the external file-name component of *assignment-name-1* is determined, see “Assignment name for user-defined words and literals” on page 128.

User-defined word

assignment-name-1 must follow the rules for a COBOL word, and can be up to 30 single-byte characters in length. The user-defined word cannot be the same as a reserved word, but can be the same as any other user-defined word in the program, including the name of a data item. The external file-name component is treated at run time initially as the name of an environment variable; then, if the environment variable is not set, or is set to the empty string, the component is treated directly as the system file-name:

- **External file-name:** Each time a COBOL file is opened, the external file-name is used as the name of an environment variable. If the environment variable is set to a nonempty value, the value is treated as the system file-name, or as the system file-name preceded by a file-system ID. For details, see “Assignment name for data-names and environment variables” on page 130. If a file-system ID is specified in the user-defined word and also in the environment variable, the environment variable takes precedence. For more information about file-system precedence, see *Precedence of file-system determination* in the *COBOL for AIX Programming Guide*.
- **System file-name:** If the environment variable indicated by the external file-name is not set, the user-defined word is treated as either:
 - The system file-name
 - The system file-name preceded by a file-system ID
 - The system file-name preceded by a file-system ID preceded by a comment

For details, see “Assignment name for user-defined words and literals” on page 128.

Literal *assignment-name-1* must follow the rules for a COBOL alphanumeric literal. All characters specified within the literal delimiters are used without any mapping. No check is made at run time for the existence of an environment variable. For details, see “Assignment name for user-defined words and literals” on page 128.

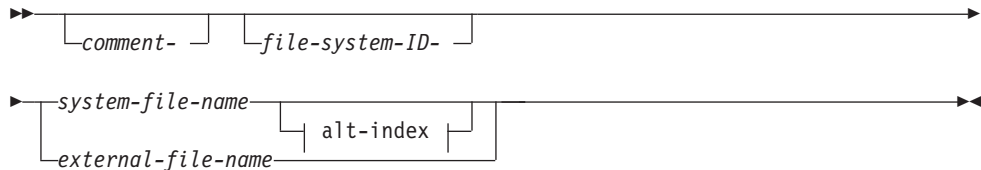
USING *data-name-9*

Must be defined in the working-storage section as a data item of category alphanumeric, and must not be subordinate to the file description for *file-name-1*. The content of *data-name-9* is evaluated each time the file is opened to determine the assignment information. As for an environment variable value, the assignment information is treated as the system file-name, or as the system file-name preceded by a file-system ID. For details, see “Assignment name for data-names and environment variables” on page 130.

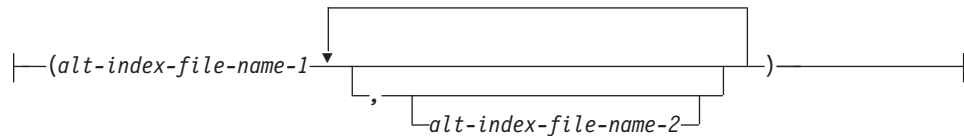
Assignment name for user-defined words and literals

If USING is not specified, the user-defined word or literal that is specified for the assignment name is processed as described below.

Format: assignment-name



alt-index:



If the user-defined word does not contain any hyphens, the entire string is treated as the external file-name. If the literal does not contain any hyphens, the entire string is treated as *file name information*. File name information can be a single system file-name, or a single system file-name followed by a list of alternate index file-names, or a concatenation of system file-names that are separated by colons. For each of these alternatives, each system file-name can be qualified by a path name. Otherwise, the user-defined word or literal is partitioned into up to three components separated by hyphens:

comment

Characters to the left of the file-system ID if specified, or characters to the left of the external file-name or system file-name if there is no valid file-system ID are interpreted as a comment.

file-system-ID

If the next-to-rightmost component meets all the following criteria, the first three characters of the string, folded to uppercase, are interpreted as the file-system ID:

- The string consists of three or more characters.
- The first (leftmost) three characters are alphanumeric (A-Z, a-z, or 0-9).
- The first character is alphabetic (A-Z or a-z).

If the string does not meet all of these criteria, it is treated as part of the comment.

external-file-name or *file name information*

The rightmost component is the external file-name or one of the three forms of file name information.

Because the hyphen is used as the separator, an external file-name or system file-name that contains one or more hyphens cannot be specified directly with a user-defined word or a literal. To specify an external file-name or system file-name that contains one or more hyphens, use either of the following approaches:

- Set the environment variable indicated by the external file-name to an appropriate value at run time.
- Specify the USING *data-name-9* form of the ASSIGN clause, and move an appropriate value to *data-name-9* before opening the file.

If *assignment-name-1* is specified as a user-defined word, *external-file-name* is interpreted at run time as the name of an environment variable set to a nonempty value or, if there is no such environment variable, directly as the system file-name.

If *assignment-name-1* is specified as a literal, the rightmost component is always interpreted as file name information directly.

Examples:

Table 7. Assignment name for user-defined words

User-defined word	Comment	File-system ID	External file-name
Read-Only-VSAM-Orders	Read-Only	SdU	ORDERS
This-is-my-file	This-is-my	(Default)	FILE
Comment-STL--file	Comment-STL-	(Default)	FILE
Watch-for-this-file	Watch-for	THI (invalid)	FILE

Table 8. Assignment name for literals

Literal	Comment	File-system ID	System file-name
Read-Only-Vsam-Orders	Read-Only	SdU	Orders
Eh?-What's this?	Eh?	(Default)	What's this?
This-is-a-DB2-CICS.FILE	This-is-a	DB2	FILE (under schema CICS)
I-Like-STL!-	I-Like	STL	(Null)
vsa-././cics/sfs/svr/W123	(None)	SFS	W123 (on SFS server svr)

Specifying alternate indexes: The compiler normally assigns appropriate alternate index file-names; however, you must provide alternate index file-names for:

- Indexed SFS files that have alternate indexes. Each index file-name is of this form:
././cics/sfs/sfsServer/base-file-name;index-file-name
- Indexed SdU files that have alternate index files with names other than the COBOL compiler defaults. For example, a file that is created through a different language, such as PL/I.

Alternate index file-names, if specified, must be specified in the same order as the alternate record keys are specified in the source program. You can omit alternate index file-names, but any other alternate index file-names must correspond to the position in the file definition. The following example shows how to specify the first and third alternate index file-names:

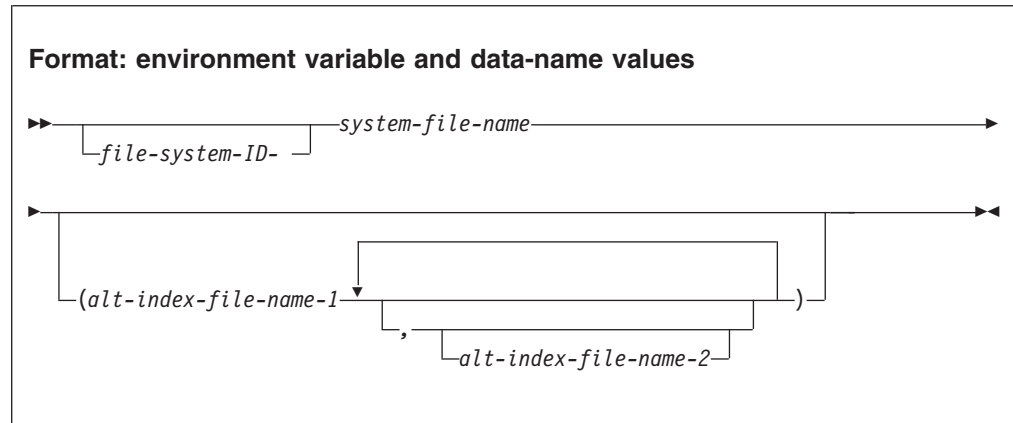
```
base-file-name(first-index-file-name,,third-index-file-name)
```

In this example, the compiler assigns a default file-name for the second alternate index file.

Alternate index file-names are ignored for file systems other than SdU or SFS.

Assignment name for data-names and environment variables

If an environment variable or data-name is specified for the assignment name, the value of the data-name or the environment variable is processed as described below.



The value in *data-name-9* or the environment variable is evaluated each time that an OPEN statement for the corresponding COBOL file is executed. If the value does not contain any hyphens, the entire string is treated as *file name information*. File name information can be a single system file-name, or a single system file-name followed by a list of alternate index file-names, or a concatenation of system file-names that are separated by colons. For each of these alternatives, each system file-name can be qualified by a path name. If the value contains one or more hyphens with at least one character after the leftmost hyphen, and the string to the left of the leftmost hyphen meets all the following criteria, the value is interpreted as *file-system-ID*, followed by file name information:

- The string consists of three or more characters.
- The first (leftmost) three characters are alphanumeric (A-Z, a-z, or 0-9).
- The first character is alphabetic (A-Z or a-z).

file-system-ID

The first three characters of *file-system-ID*, folded to uppercase, are interpreted as the file-system ID, subject to validation at run time. The value of a file-system ID that is specified at run time overrides any file-system ID that is specified in the user-defined word of *assignment-name-1*.

file name information

One of the three forms of file name information. For information about specifying alternate indexes, see "Specifying alternate indexes" under "Assignment name for user-defined words and literals" on page 128.

The examples in the following table assume that runtime option FILESYS has been set to SdU, and thus the file-system ID defaults to SdU.

Table 9. Assignment name for data-names and environment variables

Environment variable value or data-name value	File-system ID	System file-name
my-fyle	SdU	my-fyle
abc/my-fyle	ABC (invalid)	fyle
payroll-fyle	PAY (invalid)	fyle
rsd-payroll-file	RSD	payroll-file
\\Strange (...) name!	SdU	\\Strange (...) name!
db2-File1	DB2	File1
./db2-File2	SdU	./db2-File2
sfs-././cics/sfs/svr/F1	SFS	F1 (on SFS server svr)
STL-././cics/sfs/svr/F1	STL	(Invalid path)
stl-file1:file2	STL	file1 and file2 (concatenated)

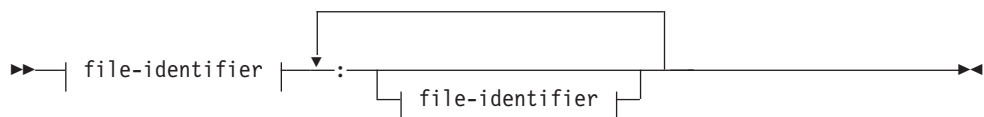
For more information about identifying files, see *Identifying files* in the *COBOL for AIX Programming Guide*.

For more information about file concept and terminology, see *File concepts and terminology* in the *COBOL for AIX Programming Guide*.

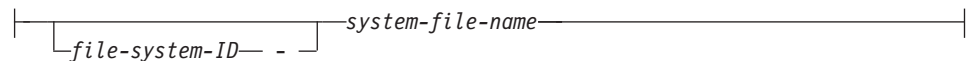
File concatenation

COBOL for AIX supports assignment of a concatenation of files to an internal file (*file-name-1* in the SELECT clause). The file concatenation is specified by multiple file identifiers that are separated by colons (:).

Format: concatenated file identifier



file-identifier:



file-system-ID

The file system in which *system-file-name* exists if *file-system-ID* consists of at least three alphanumeric characters, and the first character is alphabetic. Otherwise, the entire file identifier is interpreted as the name of the file in the default file system.

system-file-name

The name of the file in the specified file system if *file-system-ID* is specified;

otherwise, the entire file identifier is interpreted as the name of the file in the default file system. *system-file-name* can include the drive and path information for the file.

For example:

```
export MYFILE='STL-/home/user1/file1:STL-/home/user1/file2'
```

In this example, MYFILE is the result of the file concatenation of /home/user1/file1 and /home/user1/file2, both in the STL file system.

A file identifier cannot contain a colon (:) because the file identifier would then be interpreted as a concatenation of two separate files.

Up to 256 file identifiers can be specified in a concatenation. File identifiers need not be unique. For example, a concatenation can consist of 256 occurrences of the same file identifier. Adjacent colons are treated as a single colon.

File concatenation is supported for the environment variable value, the ASSIGN USING data item content, and the literal form of the assignment name.

A COBOL internal file assigned to a concatenation of files must meet the following criteria:

- FILE ORGANIZATION is SEQUENTIAL or LINE SEQUENTIAL.
- ACCESS MODE is SEQUENTIAL.
- The mode of any OPEN statements is INPUT.

These criteria are checked at run time, even if the concatenation is specified in the literal form of the assignment name.

If a colon precedes or follows a single file identifier, the file becomes a member of a trivial concatenation, and is read-only in any programs that use the file, regardless of its file permissions.

Concatenation is supported for all file systems. The file-system ID can be specified on any or all of the file identifiers in a given concatenation. However, all file identifiers in a concatenation must specify or default to the same file system at run time, and the files must have consistent attributes.

A concatenation can include individual generation files or entire generation data groups (GDGs). If a GDG is specified as a member of a concatenation, the individual files in the group are read in generation order, that is, from the most current generation to the oldest generation. For more information about generation data groups or generation files, see *Generation data groups* in the *COBOL for AIX Programming Guide*.

A concatenation can also include empty files, that is, files that contain no records. When empty files are read, they immediately return end-of-file internally, and are thus invisible to the program that reads the concatenated files. If the concatenation contains only empty files, the first READ operation returns end-of-file, and the AT END condition exists.

If a COBOL file is defined with the OPTIONAL phrase in a SELECT clause, the concatenation can include unavailable files. A file is *unavailable* if it does not exist or the file permissions do not allow the requested operation. An unavailable file is treated like an empty file, thus is invisible to the program.

| For more information about file availability, see Table 51 on page 393.

| For more information about concatenating files, see *Concatenating files* in the
| *COBOL for AIX Programming Guide*.

RESERVE clause

The RESERVE clause is syntax checked, but has no effect on the execution of the program.

ORGANIZATION clause

The ORGANIZATION clause identifies the logical structure of the file. The logical structure is established at the time the file is created and cannot subsequently be changed.

You can find a discussion of the different ways in which data can be organized and of the different access methods that you can use to retrieve the data under “File organization and access modes” on page 137.

ORGANIZATION IS SEQUENTIAL (format 1)

A predecessor-successor relationship among the records in the file is established by the order in which records are placed in the file when it is created or extended.

ORGANIZATION IS INDEXED (format 2)

The position of each logical record in the file is determined by indexes created with the file and maintained by the system. The indexes are based on embedded keys within the file's records.

ORGANIZATION IS RELATIVE (format 3)

The position of each logical record in the file is determined by its relative record number.

ORGANIZATION IS LINE SEQUENTIAL (format 4)

A predecessor-successor relationship among the records in the file is established by the order in which records are placed in the file when it is created or extended. A record in a LINE SEQUENTIAL file can consist only of printable characters.

If you omit the ORGANIZATION clause, the compiler assumes ORGANIZATION IS SEQUENTIAL.

If the file connector referenced by *file-name-1* in the SELECT clause is an external file connector, the same organization must be specified for all file-control entries in the run unit that reference this file connector.

File organization

You establish the organization of the data when you create a file. Once the file has been created, you can expand the file, but you cannot change the organization.

Sequential organization

The physical order in which the records are placed in the file determines the sequence of records. The relationships among records in the file do not change, except that the file can be extended. Records can be fixed length or variable length; there are no keys.

Each record in the file except the first has a unique predecessor record; and each record except the last has a unique successor record.

Indexed organization

Each record in the file has one or more embedded keys (referred to as *key data items*); each key is associated with an index. An index provides a logical path to the data records according to the contents of the associated embedded record key data items. Indexed files must be direct-access storage files. Records can be fixed length or variable length.

Each record in an indexed file must have an embedded prime key data item. When records are inserted, updated, or deleted, they are identified solely by the values of their prime keys. Thus, the value in each prime key data item must be unique and must not be changed when the record is updated. You tell COBOL the name of the prime key data item in the RECORD KEY clause of the file-control paragraph.

In addition, each record in an indexed file can contain one or more embedded alternate key data items. Each alternate key provides another means of identifying which record to retrieve. You tell COBOL the name of any alternate key data items on the ALTERNATE RECORD KEY clause of the file-control paragraph.

The key used for any specific input-output request is known as the *key of reference*.

Relative organization

Think of the file as a string of record areas, each of which contains a single record. Each record area is identified by a relative record number; the access method stores and retrieves a record based on its relative record number. For example, the first record area is addressed by relative record number 1 and the 10th is addressed by relative record number 10. The physical sequence in which the records were placed in the file has no bearing on the record area in which they are stored, and thus no effect on each record's relative record number. Relative files must be direct-access files. Records can be fixed length or variable length.

Line-sequential organization

In a line-sequential file, each record contains a sequence of characters that ends with a record delimiter. The delimiter is not counted in the length of the record.

When a record is written, any trailing blanks are removed prior to adding the record delimiter. The characters in the record area from the first character up to and including the added record delimiter constitute one record and are written to the file.

When a record is read, characters are read one at a time into the record area until:

- The first record delimiter is encountered. The record delimiter is discarded and the remainder of the record is filled with spaces.

- The entire record area is filled with characters. If the first unread character is the record delimiter, it is discarded. Otherwise, the first unread character becomes the first character read by the next READ statement.
- End-of-file is encountered. The remainder of the record area is filled with spaces.

Records written to line-sequential files must consist of data items described as USAGE DISPLAY or DISPLAY-1 or a combination of DISPLAY and DISPLAY-1 items. If the CHAR(EBCDIC) compiler option is in effect, a DISPLAY or DISPLAY-1 item can be encoded in either ASCII or EBCDIC, depending on the presence or absence of the NATIVE phrase in the USAGE clause of the data item. A zoned decimal data item either must be unsigned or, if signed, must be declared with the SEPARATE CHARACTER phrase.

A line-sequential file can contain printable characters and control characters other than X'00'. Be aware though that if your file contains a line-feed character (X'0A'), the line-feed character will function as a record delimiter.

The following are not supported for line-sequential files:

- APPLY WRITE-ONLY clause
- CODE-SET clause
- DATA RECORDS clause
- LABEL RECORDS clause
- LINAGE clause
- I-O phrase of the OPEN statement
- PADDING CHARACTER clause
- RECORD CONTAINS 0 clause
- RECORD CONTAINS clause format 2 (for example: RECORD CONTAINS 100 to 200 CHARACTERS)
- RECORD DELIMITER clause
- RECORDING MODE clause
- RERUN clause
- RESERVE clause
- REVERSED phrase of the OPEN statement
- REWRITE statement
- VALUE OF clause of file description entry
- WRITE ... AFTER ADVANCING *mnemonic-name*
- WRITE ... AT END-OF-PAGE
- WRITE ... BEFORE ADVANCING

Language elements treated as comments

For other files (sequential, relative, and indexed), the following language elements are syntax checked, but have no effect on the execution of the program:

- APPLY WRITE-ONLY clause
- CLOSE ... FOR REMOVAL
- CLOSE ... WITH NO REWIND
- CODE-SET clause
- DATA RECORDS clause
- LABEL RECORDS clause

- MULTIPLE FILE TAPE clause
- OPEN ... REVERSE
- PADDING CHARACTER clause
- PASSWORD clause
- RECORD CONTAINS 0 clause
- RECORD DELIMITER clause
- RECORDING MODE clause (for relative and indexed files)
- RERUN clause
- RESERVE clause
- SAME AREA clause
- SAME SORT AREA clause
- SAME SORT-MERGE AREA clause
- VALUE OF clause of file description entry

No error messages are generated (with the exception of the data name option for the LABEL RECORDS, USE ... AFTER ... LABEL PROCEDURE, and GO TO MORE-LABELS clauses).

PADDING CHARACTER clause

The PADDING CHARACTER clause specifies a character to be used for block padding on sequential files.

data-name-5

Must be defined in the data division as a one-character data item of category alphabetic, alphanumeric, or national, and must not be defined in the file section. *data-name-5* can be qualified.

literal-2

Must be a one-character alphanumeric literal or national literal.

For external files, *data-name-5*, if specified, must reference an external data item.

The PADDING CHARACTER clause is syntax checked, but has no effect on the execution of the program.

RECORD DELIMITER clause

The RECORD DELIMITER clause indicates the method of determining the length of a variable-length record on an external medium. It can be specified only for variable-length records.

STANDARD-1

If STANDARD-1 is specified, the external medium must be a magnetic tape file.

assignment-name-2

Can be any COBOL word.

The RECORD DELIMITER clause is syntax checked, but has no effect on the execution of the program.

ACCESS MODE clause

The ACCESS MODE clause defines the manner in which the records of the file are made available for processing. If the ACCESS MODE clause is not specified, sequential access is assumed.

For sequentially accessed relative files, the ACCESS MODE clause does not have to precede the RELATIVE KEY clause.

ACCESS MODE IS SEQUENTIAL

Can be specified in all formats.

Format 1: sequential

Records in the file are accessed in the sequence established when the file is created or extended. Format 1 supports only sequential access.

Format 2: indexed

Records in the file are accessed in the sequence of ascending record key values according to the collating sequence of the file.

Format 3: relative

Records in the file are accessed in the ascending sequence of relative record numbers of existing records in the file.

Format 4: line-sequential

Records in the file are accessed in the sequence established when the file is created or extended. Format 4 supports only sequential access.

ACCESS MODE IS RANDOM

Can be specified in formats 2 and 3 only.

Format 2: indexed

The value placed in a record key data item specifies the record to be accessed.

Format 3: relative

The value placed in a relative key data item specifies the record to be accessed.

ACCESS MODE IS DYNAMIC

Can be specified in formats 2 and 3 only.

Format 2: indexed

Records in the file can be accessed sequentially or randomly, depending on the form of the specific input-output statement used.

Format 3: relative

Records in the file can be accessed sequentially or randomly, depending on the form of the specific input-output request.

File organization and access modes

File organization is the permanent logical structure of the file. You tell the computer how to retrieve records from the file by specifying the *access mode* (sequential, random, or dynamic). For details on the access methods and data organization, see Table 6 on page 122.

Sequentially organized data can be accessed only sequentially; however, data that has indexed or relative organization can be accessed in any of the three access modes.

Access modes

Sequential-access mode

Allows reading and writing records of a file in a serial manner; the order of reference is implicitly determined by the position of a record in the file.

Random-access mode

Allows reading and writing records in a programmer-specified manner; the control of successive references to the file is expressed by specifically defined keys supplied by the user.

Dynamic-access mode

Allows the specific input-output statement to determine the access mode. Therefore, records can be processed sequentially or randomly or both.

For external files, every file-control entry in the run unit that is associated with that external file must specify the same access mode. In addition, for relative file entries, *data-name-4* must reference an external data item, and the RELATIVE KEY phrase in each associated file-control entry must reference that same external data item.

Relationship between data organizations and access modes

This section discusses which access modes are valid for each type of data organization.

Sequential files

Files with sequential organization can be accessed only sequentially. The sequence in which records are accessed is the order in which the records were originally written.

Line-sequential files

Same as for sequential files (described above).

Indexed files

All three access modes are allowed.

In the sequential access mode, the sequence in which records are accessed is the ascending order (or, optionally, descending order) of the record key value. The order of retrieval within a set of records that have duplicate alternate record key values is the order in which records were written into the set.

In the random access mode, you control the sequence in which records are accessed. A specific record is accessed by placing the value of its key or keys in the RECORD KEY data item (and the ALTERNATE RECORD KEY data item). If a set of records has duplicate alternate record key values, only the first record written is available.

In the dynamic access mode, you can change as needed from sequential access to random access by using appropriate forms of input-output statements.

Relative files

All three access modes are allowed.

In the sequential access mode, the sequence in which records are accessed is the ascending order (or, optionally, descending order) of the relative record numbers of all records that exist within the file.

In the random access mode, you control the sequence in which records are accessed. A specific record is accessed by placing its relative record number in the RELATIVE KEY data item; the RELATIVE KEY must not be defined within the record description entry for the file.

In the dynamic access mode, you can change as needed from sequential access to random access by using appropriate forms of input-output statements.

RECORD KEY clause

The RECORD KEY clause (format 2) specifies the data item within the record that is the prime RECORD KEY for an indexed file. The values contained in the prime RECORD KEY data item must be unique among records in the file.

data-name-2

The prime RECORD KEY data item.

data-name-2 must be described within a record description entry associated with the file. The key can have any of the following data categories:

- Alphanumeric
- Numeric
- Numeric-edited (with usage DISPLAY or NATIONAL)
- Alphanumeric-edited
- Alphabetic
- External floating-point (with usage DISPLAY or NATIONAL)
- Internal floating-point
- DBCS
- National
- National-edited

Regardless of the category of the key data item, the key is treated as an alphanumeric item. The collation order of the key is determined by the item's binary value order when the key is used for locating a record or for setting the file position indicator associated with the file.

data-name-2 cannot be a windowed date field.

data-name-2 must not reference a group item that contains a variable-occurrence data item. *data-name-2* can be qualified.

If the indexed file contains variable-length records, *data-name-2* need not be contained within the minimum record size specified for the file. That is, *data-name-2* can exceed the minimum record size, but this is not recommended.

The data description of *data-name-2* and its relative location within the record must be the same as those used when the file was defined.

If the file has more than one record description entry, *data-name-2* need be described in only one of those record description entries. The identical character positions referenced by *data-name-2* in any one record description entry are implicitly referenced as keys for all other record description entries for that file.

For files defined with the EXTERNAL clause, all file description entries in the run unit that are associated with the file must have data description entries for *data-name-2* that specify the same relative location in the record and the same length.

ALTERNATE RECORD KEY clause

The ALTERNATE RECORD KEY clause (format 2) specifies a data item within the record that provides an alternative path to the data in an indexed file.

data-name-3

An ALTERNATE RECORD KEY data item.

data-name-3 must be described within a record description entry associated with the file. The key can have any of the following data categories:

- Alphanumeric
- Numeric
- Numeric-edited (with usage DISPLAY or NATIONAL)
- Alphanumeric-edited
- Alphabetic
- External floating-point (with usage DISPLAY or NATIONAL)
- Internal floating-point
- DBCS
- National
- National-edited

Regardless of the category of the key data item, the key is treated as an alphanumeric item. The collation order of the key is determined by the item's binary value order when the key is used for locating a record or for setting the file position indicator associated with the file.

data-name-3 cannot be a windowed date field.

data-name-3 must not reference a group item that contains a variable-occurrence data item. *data-name-3* can be qualified.

If the indexed file contains variable-length records, *data-name-3* need not be contained within the minimum record size specified for the file. That is, *data-name-3* can exceed the minimum record size, but this is not recommended.

If the file has more than one record description entry, *data-name-3* need be described in only one of these record description entries. The identical character positions referenced by *data-name-3* in any one record description entry are implicitly referenced as keys for all other record description entries of that file.

The data description of *data-name-3* and its relative location within the record must be the same as those used when the file was defined. The number of alternate record keys for the file must also be the same as that used when the file was created.

The leftmost character position of *data-name-3* must not be the same as the leftmost character position of the prime RECORD KEY or of any other ALTERNATE RECORD KEY.

If the Duplicates phrase is not specified, the values contained in the ALTERNATE RECORD KEY data item must be unique among records in the file.

If the Duplicates phrase is specified, the values contained in the ALTERNATE RECORD KEY data item can be duplicated within any records in the file. In sequential access, the records with duplicate keys are retrieved in the order in which they were placed in the file. In random access, only the first record written in a series of records with duplicate keys can be retrieved.

For files defined with the EXTERNAL clause, all file description entries in the run unit that are associated with the file must have data description entries for *data-name-3* that specify the same relative location in the record and the same length. The file description entries must specify the same number of alternate record keys and the same Duplicates phrase.

RELATIVE KEY clause

The RELATIVE KEY clause (format 3) identifies a data-name that specifies the relative record number for a specific logical record within a relative file.

data-name-4

Must be defined as an unsigned integer data item whose description does not contain the PICTURE symbol P. *data-name-4* must not be defined in a record description entry associated with this relative file. That is, the RELATIVE KEY is not part of the record. *data-name-4* can be qualified.

data-name-4 cannot be a windowed date field.

data-name-4 is required for ACCESS IS SEQUENTIAL only when the START statement is to be used. It is always required for ACCESS IS RANDOM and ACCESS IS DYNAMIC. When the START statement is issued, the system uses the contents of the RELATIVE KEY data item to determine the record at which sequential processing is to begin.

If a value is placed in *data-name-4*, and a START statement is not issued, the value is ignored and processing begins with the first record in the file.

If a relative file is to be referenced by a START statement, you must specify the RELATIVE KEY clause for that file.

For external files, *data-name-4* must reference an external data item, and the RELATIVE KEY phrase in each associated file-control entry must reference that same external data item in each case.

The ACCESS MODE IS RANDOM clause must not be specified for file-names specified in the USING or GIVING phrase of a SORT or MERGE statement.

PASSWORD clause

The PASSWORD clause is syntax checked, but has no effect on the execution of the program.

FILE STATUS clause

The FILE STATUS clause monitors the execution of each input-output operation for the file.

When the FILE STATUS clause is specified, the system moves a value into the file status key data item after each input-output operation that explicitly or implicitly refers to this file. The value indicates the status of execution of the statement. (See the file status key description under “Common processing facilities” on page 296.)

data-name-1

The file status key data item can be defined in the working-storage, local-storage, or linkage section as one of the following:

- A two-character data item of category alphanumeric
- A two-character data item of category national
- A two-digit data item of category numeric with usage DISPLAY or NATIONAL (an external decimal data item)

data-name-1 must not contain the PICTURE symbol 'P'.

data-name-1 can be qualified.

The file status key data item must not be variably located; that is, the data item cannot follow a data item that contains an OCCURS DEPENDING ON clause.

data-name-8

Represents information returned from the file system. Because the definitions are specific to the file systems and platforms, applications that depend on the specific values in *data-name-8* might not be portable across platforms.

How you define *data-name-8* is dependent on the file system you are using.

STL, RSD, and native platform file systems

You must define *data-name-8* with PICTURE 9(6) and USAGE DISPLAY attributes. However, you can define an additional field with PICTURE X(*n*). The file system defines the feedback values, which are converted to the six-digit external decimal representation with leading zeros when the file systems feedback value is less than 100000. If you have defined an additional field using PICTURE X(*n*), then X(*n*) contains additional information that describes any nonzero feedback code. (For most programs, an *n* value of 100 should be adequate to show the complete message text. If the file is defined with a large number of alternate keys, then allow 100 bytes plus 20 bytes per alternate key.)

SFS and SdU file systems

You must define *data-name-8* with PICTURE X(*n*) and USAGE DISPLAY attributes, where *n* is 6 or greater. The PICTURE string value represents the first *n* bytes of the reply message structure as described in *COBOL for AIX VSAM File System Reply Messages*. If the size of the reply message structure (*m*) is shorter than *n*, only the first *m* bytes contain useful information.

DB2 file system

You must define *data-name-8* as a group item such as the following example:

```
01 FileStatus2.  
  02 FS2-SQLCODE COMP PICTURE S9(9).  
  02 FS2-SQLSTATE PICTURE X(5).
```

The runtime values in FS2-SQLCODE and FS2-SQLSTATE represent SQL feedback information for the operation that is previously completed.

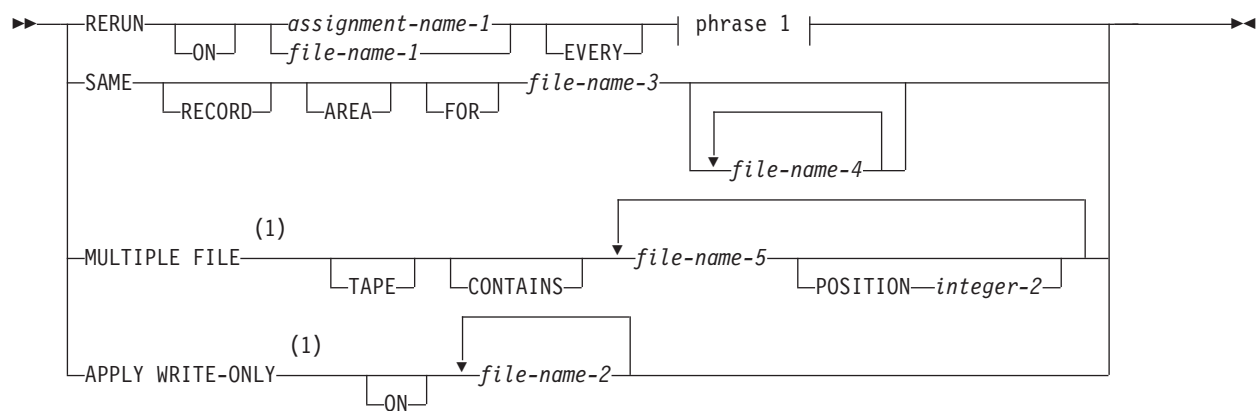
I-O-CONTROL paragraph

The I-O-CONTROL paragraph of the input-output section specifies when checkpoints are to be taken and the storage areas to be shared by different files. This paragraph is optional in a COBOL program.

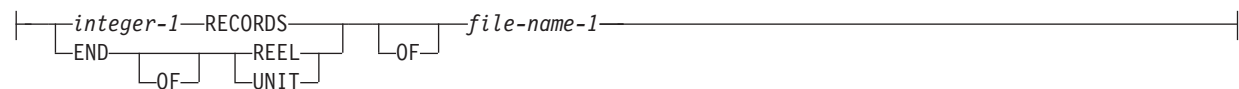
The keyword I-O-CONTROL can appear only once, at the beginning of the paragraph. The word I-O-CONTROL must begin in Area A and must be followed by a separator period.

The order in which I-O-CONTROL paragraph clauses are written is not significant. The I-O-CONTROL paragraph ends with a separator period.

Format: sequential- i-o-control-entry



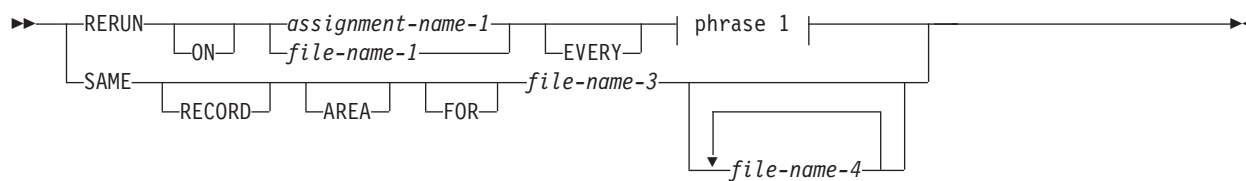
phrase 1:



Notes:

- 1 The MULTIPLE FILE clause and APPLY WRITE-ONLY clause are syntax checked, but have no effect on the execution of the program.

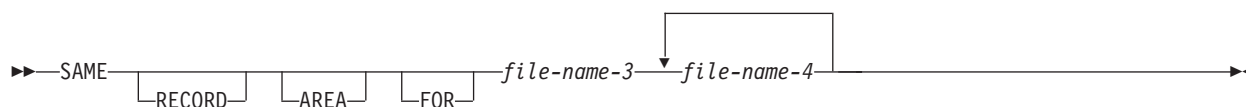
Format: relative-and-indexed- i-o-control-entry



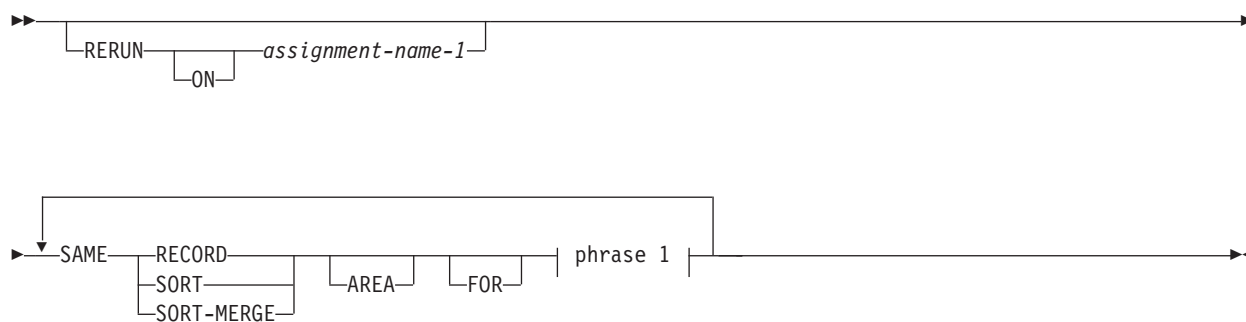
phrase 1:



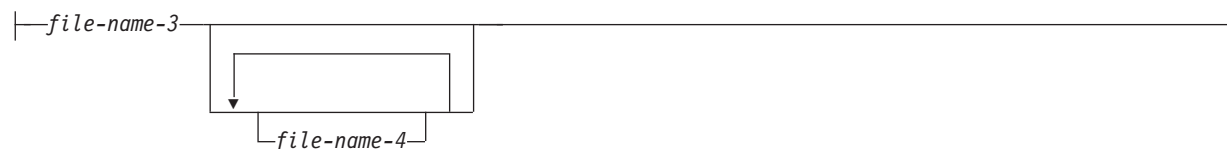
Format: line-sequential-i-o-control-entry



Format: sort/merge-i-o-control-entry



phrase 1:



RERUN clause

The RERUN clause specifies that checkpoint records are to be taken. Subject to the restrictions given with each phrase, more than one RERUN clause can be specified.

The RERUN clause is syntax checked, but has no effect on the execution of programs compiled with the NOTHREAD compiler option.

The RERUN clause is invalid syntax for programs compiled with the THREAD compiler option.

Do not use the RERUN clause:

- For files described with the EXTERNAL clause
- In programs with the RECURSIVE clause specified
- In programs compiled with the THREAD option
- In methods

file-name-1

Must be a sequentially organized file.

assignment-name-1

The external file for the checkpoint file. It must not be the same assignment-name as that specified in any ASSIGN clause throughout the entire program, including contained and containing programs.

SORT/MERGE considerations:

When the RERUN clause is specified in the I-O-CONTROL paragraph, checkpoint records are written at logical intervals determined by the sort/merge program during execution of each SORT or MERGE statement in the program. When the RERUN clause is omitted, checkpoint records are not written.

There can be only one SORT/MERGE I-O-CONTROL paragraph in a program, and it cannot be specified in contained programs. It will have a global effect on all SORT and MERGE statements in the program unit.

EVERY *integer-1* RECORDS

A checkpoint record is to be written for every *integer-1* records in *file-name-1* that are processed.

When multiple *integer-1* RECORDS phrases are specified, no two of them can specify the same value for *file-name-1*.

If you specify the *integer-1* RECORDS phrase, you must specify *assignment-name-1*.

EVERY END OF REEL/UNIT

A checkpoint record is to be written whenever end-of-volume for *file-name-1* occurs. The terms REEL and UNIT are interchangeable.

When multiple END OF REEL/UNIT phrases are specified, no two of them can specify the same value for *file-name-1*.

The END OF REEL/UNIT phrase can be specified only if *file-name-1* is a sequentially organized file.

SAME AREA clause

The SAME AREA clause is syntax checked, but has no effect on the execution of the program.

SAME RECORD AREA clause

The SAME RECORD AREA clause specifies that two or more files are to use the same main storage area for processing the current logical record.

The files named in a SAME RECORD AREA clause need not have the same organization or access.

file-name-3, file-name-4

Must be specified in the file-control paragraph of the same program.

file-name-3 and *file-name-4* must not reference a file that is defined with the EXTERNAL clause.

All of the files can be open at the same time. A logical record in the shared storage area is considered to be both of the following:

- A logical record of each opened output file in the SAME RECORD AREA clause
- A logical record of the most recently read input file in the SAME RECORD AREA clause

More than one SAME RECORD AREA clause can be included in a program. However:

- A specific file-name must not appear in more than one SAME RECORD AREA clause.
- If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all the file-names in that SAME AREA clause must appear in that SAME RECORD AREA clause. However, the SAME RECORD AREA clause can contain additional file-names that do not appear in the SAME AREA clause.
- The rule that in the SAME AREA clause only one file can be open at one time takes precedence over the SAME RECORD AREA rule that all the files can be open at the same time.
- If the SAME RECORD AREA clause is specified for several files, the record description entries or the file description entries for these files must not include the GLOBAL clause.
- The SAME RECORD AREA clause must not be specified when the RECORD CONTAINS 0 CHARACTERS clause is specified.

The files named in the SAME RECORD AREA clause need not have the same organization or access.

SAME SORT AREA clause

The SAME SORT AREA clause is syntax checked but has no effect on the execution of the program.

file-name-3, file-name-4

Must be specified in the file-control paragraph of the same program.

file-name-3 and *file-name-4* must not reference a file that is defined with the EXTERNAL clause.

When the SAME SORT AREA clause is specified, at least one file-name specified must name a sort file. Files that are not sort files can also be specified. The following rules apply:

- More than one SAME SORT AREA clause can be specified. However, a given sort file must not be named in more than one such clause.
- If a file that is not a sort file is named in both a SAME AREA clause and in one or more SAME SORT AREA clauses, all the files in the SAME AREA clause must also appear in that SAME SORT AREA clause.
- Files named in a SAME SORT AREA clause need not have the same organization or access.
- Files named in a SAME SORT AREA clause that are not sort files do not share storage with each other unless they are named in a SAME AREA or SAME RECORD AREA clause.
- During the execution of a SORT or MERGE statement that refers to a sort or merge file named in this clause, any nonsort or nonmerge files associated with file-names named in this clause must not be in the open mode.

SAME SORT-MERGE AREA clause

The SAME SORT-MERGE AREA clause is equivalent to the SAME SORT AREA clause (see “SAME SORT AREA clause” on page 146).

MULTIPLE FILE TAPE clause

The MULTIPLE FILE TAPE clause (format 1) specifies that two or more files share the same physical reel of tape.

This clause is syntax checked, but has no effect on the execution of the program.

APPLY WRITE-ONLY clause

The APPLY WRITE-ONLY clause is syntax checked, but has no effect on the execution of the program.

Part 5. Data division

Chapter 17. Data division overview	151
File section	152
Working-storage section	153
Local-storage section	154
Linkage section	155
Data units	155
File data	155
Program data	156
Method data	156
Factory data	156
Instance data	156
Data relationships	156
Levels of data	157
Levels of data in a record description entry	157
Special level-numbers	159
Indentation	159
Classes and categories of group items	159
Classes and categories of data	160
Category descriptions	162
Alphabetic	162
Alphanumeric	162
Alphanumeric-edited	162
DBCS	163
External floating-point	163
Internal floating-point	163
National	163
National-edited	163
Numeric	164
Numeric-edited	164
Alignment rules	164
Character-string and item size	165
Signed data	166
Operational signs	166
Editing signs	166

Chapter 18. Data division--file description entries	167
File section	172
EXTERNAL clause	172
GLOBAL clause	173
BLOCK CONTAINS clause	173
RECORD clause	174
Format 1	175
Format 2	175
Format 3	175
LABEL RECORDS clause	177
VALUE OF clause	177
DATA RECORDS clause	178
LINAGE clause	178
LINAGE-COUNTER special register	180
RECORDING MODE clause	180
CODE-SET clause	180

Chapter 19. Data division--data description entry	181
Format 1	181

Format 2	182
Format 3	182
Level-numbers	182
BLANK WHEN ZERO clause	184
DATE FORMAT clause	184
Semantics of windowed date fields	185
Restrictions on using date fields	186
Combining the DATE FORMAT clause with other clauses	186
Group items that are date fields	187
Language elements that treat date fields as nondates	187
Language elements that do not accept windowed date fields as arguments	188
Language elements that do not accept date fields as arguments	188
EXTERNAL clause	189
GLOBAL clause	189
JUSTIFIED clause	190
GROUP-USAGE clause	191
OCCURS clause	192
Fixed-length tables	193
ASCENDING KEY and DESCENDING KEY phrases	193
INDEXED BY phrase	195
Variable-length tables	195
OCCURS DEPENDING ON clause	196
PICTURE clause	198
Symbols used in the PICTURE clause	199
P symbol	202
Currency symbol	203
Character-string representation	203
Data categories and PICTURE rules	204
Alphabetic items	204
Numeric items	205
Numeric-edited items	206
Alphanumeric items	206
Alphanumeric-edited items	207
DBCS items	207
National items	208
National-edited items	209
External floating-point items	210
PICTURE clause editing	211
Simple insertion editing	212
Special insertion editing	212
Fixed insertion editing	213
Floating insertion editing	214
Representing floating insertion editing	214
Zero suppression and replacement editing	215
Representing zero suppression	215
REDEFINES clause	216
REDEFINES clause considerations	218
REDEFINES clause examples	218
Undefined results	219
RENAMES clause	220
SIGN clause	222

SYNCHRONIZED clause	223
Slack bytes	226
Slack bytes within records	226
Slack bytes between records	228
USAGE clause	229
Computational items	231
DISPLAY phrase	233
Effect of CHAR(EBCDIC) compiler option	234
DISPLAY-1 phrase	234
FUNCTION-POINTER phrase	234
INDEX phrase	234
NATIONAL phrase	235
OBJECT REFERENCE phrase	235
POINTER phrase	236
PROCEDURE-POINTER phrase	237
NATIVE phrase	238
VALUE clause	239
Format 1	239
Rules for literal values	240
Format 2	241
Rules for condition-name entries	242
Format 3	244

Chapter 17. Data division overview

This overview describes the structure of the data division for programs, object definitions, factory definitions, and methods. Each section in the data division has a specific logical function within a COBOL program, object definition, factory definition, or method and can be omitted when that logical function is not needed. If included, the sections must be written in the order shown. The data division is optional.

Program data division

The data division of a COBOL source program describes, in a structured manner, all the data to be processed by the program.

Object data division

The object data division contains data description entries for instance object data (instance data). Instance data is defined in the working-storage section of the object paragraph of a class definition.

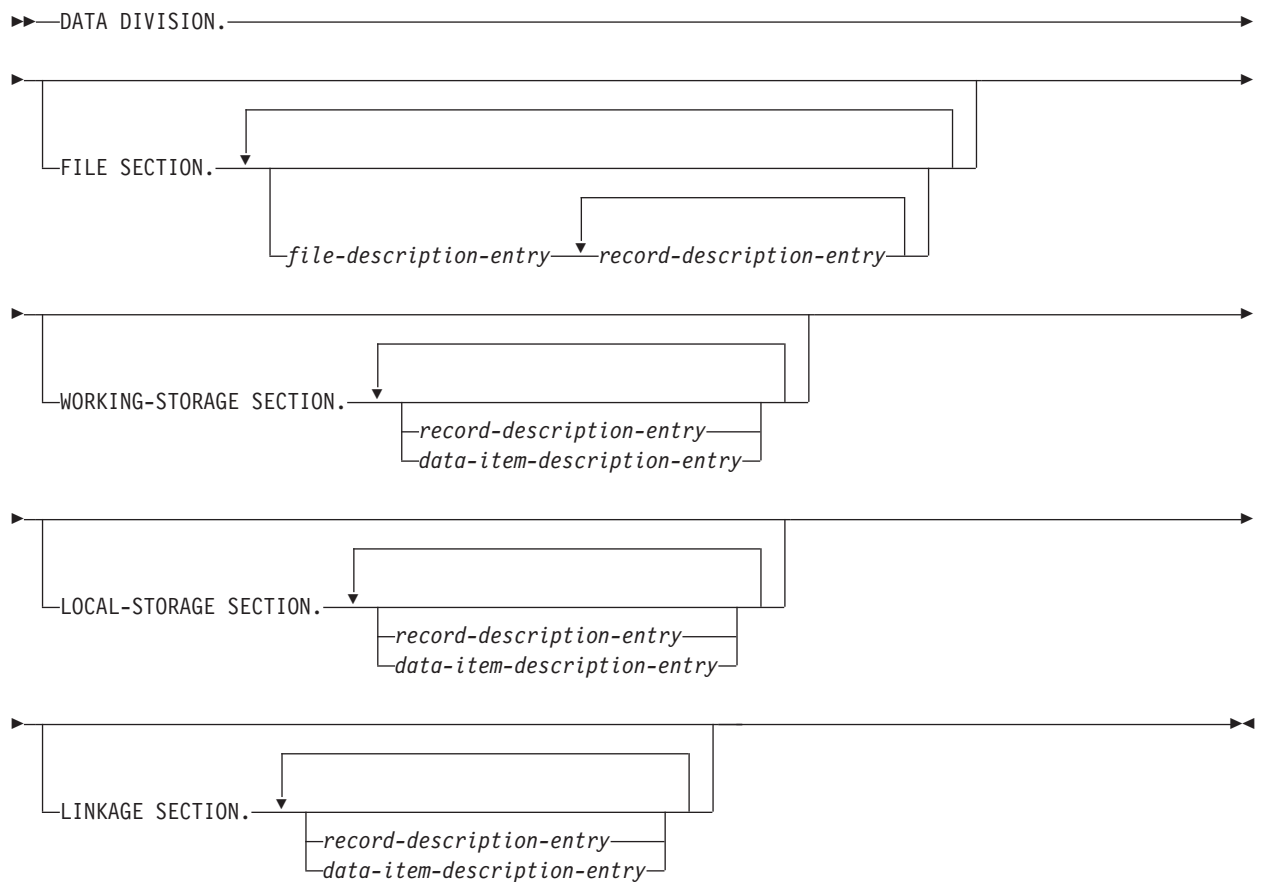
Factory data division

The factory data division contains data description entries for factory object data (factory data). Factory data is defined in the working-storage section of the factory paragraph of a class definition.

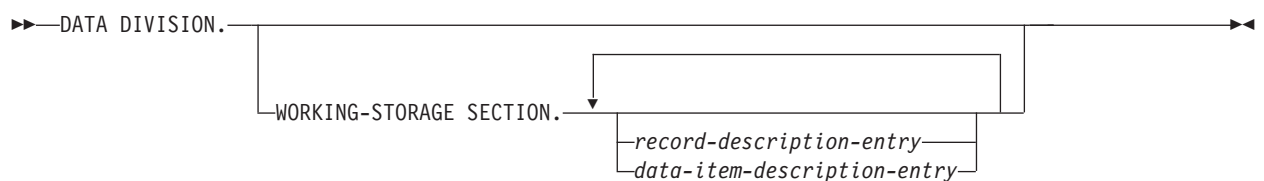
Method data division

A method data division contains data description entries for data accessible within the method. A method data division can contain a local-storage section or a working-storage section, or both. The term *method data* applies to both. Method data in local-storage is dynamically allocated and initialized on each invocation of the method; method data in working-storage is static and persists across invocations of the method.

Format: program and method data division



Format: object and factory data division



File section

The file section defines the structure of data files. The file section must begin with the header **FILE SECTION**, followed by a separator period.

file-description-entry

Represents the highest level of organization in the file section. It provides information about the physical structure and identification of a file, and gives the record-names associated with that file. For the format and the

clauses required in a file description entry, see Chapter 18, “Data division--file description entries,” on page 167.

record-description-entry

A set of data description entries (described in Chapter 19, “Data division--data description entry,” on page 181) that describe the particular records contained within a particular file.

A record in the file section must be described as an alphanumeric group item, a national group item, or an elementary data item of class alphabetic, alphanumeric, DBCS, national, or numeric.

More than one record description entry can be specified; each is an alternative description of the same record storage area.

Data areas described in the file section are not available for processing unless the file that contains the data area is open.

A method file section can define external files only. A single run-unit-level file connector is shared by all programs and methods that contain a declaration of a given external file.

Working-storage section

The working-storage section describes data records that are not part of data files but are developed and processed by a program or method. It also describes data items whose values are assigned in the source program or method and do not change during execution of the object program.

The working-storage section must begin with the section header **WORKING-STORAGE SECTION**, followed by a separator period.

Program working-storage

The working-storage section for programs (and methods) can also describe external data records, which are shared by programs and methods throughout the run unit. All clauses that are used in record descriptions in the file section and also the **VALUE** and **EXTERNAL** clauses (which might not be specified in record description entries in the file section) can be used in record descriptions in the working-storage section.

Method working-storage

A single copy of the working-storage for a method is statically allocated on the first invocation of the method and persists in a last-used state for the duration of the run unit. The same copy is used whenever the method is invoked regardless of which object instance the method is invoked upon.

If a **VALUE** clause is specified on a method working-storage data item, the data item is initialized to the **VALUE** clause value on the first invocation.

If the **EXTERNAL** clause is specified on a data description entry in a method working-storage section, a single copy of the storage for that data item is allocated once for the duration of the run unit. That storage is shared by all programs and methods in the run unit that contain a definition for the external data item.

Object working-storage

The data described in the working-storage section of an object paragraph is object instance data, usually called *instance data*. A separate copy of instance data is statically allocated for each object instance when the object

is instantiated. Instance data persists in a last-used state until the object instance is freed by the Java runtime system.

Instance data can be initialized by VALUE clauses specified in data declarations or by logic specified in an instance method.

Factory working-storage

The data described in the working-storage section of a factory paragraph is factory data. A single copy of factory data is statically allocated when the factory object for the class is created. Factory data persists in a last-used state for the duration of the run unit.

Factory data can be initialized by VALUE clauses specified in data declarations or by logic specified in a factory method.

The working-storage section contains record description entries and data description entries for independent data items, called *data item description entries*.

record-description-entry

Data entries in the working-storage section that bear a definite hierarchic relationship to one another must be grouped into records structured by level number. See Chapter 19, “Data division--data description entry,” on page 181 for more information.

data-item-description-entry

Independent items in the working-storage section that bear no hierarchic relationship to one another need not be grouped into records provided that they do not need to be further subdivided. Instead, they are classified and defined as independent elementary items. Each is defined in a separate data-item description entry that begins with either the level number 77 or 01. See Chapter 19, “Data division--data description entry,” on page 181 for more information.

Local-storage section

The local-storage section defines storage that is allocated and freed on a per-invocation basis. On each invocation, data items defined in the local-storage section are reallocated. Each data item that has a VALUE clause is initialized to the value specified in that clause.

For nested programs, data items defined in the local-storage section are allocated upon each invocation of the containing outermost program. However, each data item is reinitialized to the value specified in its VALUE clause each time the nested program is invoked.

For methods, a separate copy of the data defined in local-storage is allocated and initialized on each invocation of the method. The storage allocated for the data is freed when the method returns.

Data items defined in the local-storage section cannot specify the EXTERNAL clause.

The local-storage section must begin with the header LOCAL-STORAGE SECTION, followed by a separator period.

You can specify the local-storage section in recursive programs, in nonrecursive programs, and in methods.

Method local-storage content is the same as program local-storage content except that the GLOBAL clause has no effect (because methods cannot be nested).

Linkage section

The linkage section describes data made available from another program or method.

record-description-entry

See “Working-storage section” on page 153 for a description.

data-item-description-entry

See “Working-storage section” on page 153 for a description.

Record description entries and data item description entries in the linkage section provide names and descriptions, but storage within the program or method is not reserved because the data area exists elsewhere.

Any data description clause can be used to describe items in the linkage section with the following exceptions:

- You cannot specify the VALUE clause for items other than level-88 items.
- You cannot specify the EXTERNAL clause.

You can specify the GLOBAL clause in the linkage section. The GLOBAL clause has no effect for methods, however.

Data units

Data is grouped into the following conceptual units:

- File data
- Program data
- Method data
- Factory data
- Instance data

File data

File data is contained in files. (See “File section” on page 172.) A *file* is a collection of data records that exist on some input-output device. A file can be considered as a group of physical records; it can also be considered as a group of logical records. The data division describes the relationship between physical and logical records.

A *physical record* is a unit of data that is treated as an entity when moved into or out of storage. The size of a physical record is determined by the particular input-output device on which it is stored. The size does not necessarily have a direct relationship to the size or content of the logical information contained in the file.

A *logical record* is a unit of data whose subdivisions have a logical relationship. A logical record can itself be a physical record (that is, be contained completely within one physical unit of data); several logical records can be contained within one physical record, or one logical record can extend across several physical records.

File description entries specify the physical aspects of the data (such as the size relationship between physical and logical records, the size and names of the logical records, labeling information, and so forth).

Record description entries describe the logical records in the file (including the category and format of data within each field of the logical record), different values the data might be assigned, and so forth.

After the relationship between physical and logical records has been established, only logical records are made available to you. For this reason, a reference in this information to "records" means logical records, unless the term "physical records" is used.

Program data

Program data is created by a program instead of being read from a file.

The concept of logical records applies to program data as well as to file data. Program data can thus be grouped into logical records, and be defined by a series of record description entries. Items that need not be so grouped can be defined in independent data description entries (called *data item description entries*).

Method data

Method data is defined in the data division of a method and is processed by the procedural code in that method. Method data is organized into logical records and independent data description entries in the same manner as program data.

Factory data

Factory data is defined in the data division in the factory paragraph of a class definition and is processed by procedural code in the factory methods of that class. Factory data is organized into logical records and independent data description entries in the same manner as program data.

There is one factory object for a given class in a run unit, and therefore only one instance of factory data in a run unit for that class.

Instance data

Instance data is defined in the data division in the object paragraph of a class definition and is processed by procedural code in the instance methods of that class. Instance data is organized into logical records and independent data description entries in the same manner as program data.

There is one copy of instance data in each object instance of a given class. There can be many object instances for a given class. Each has its own separate copy of instance data.

Data relationships

The relationships among all data to be used in a program are defined in the data division through a system of level indicators and level-numbers.

A *level indicator*, with its descriptive entry, identifies each file in a program. Level indicators represent the highest level of any data hierarchy with which they are associated. FD is the file description level indicator and SD is the sort-merge file description level indicator.

A *level-number*, with its descriptive entry, indicates the properties of specific data. Level-numbers can be used to describe a data hierarchy; they can indicate that this data has a special purpose. Although they can be associated with (and subordinate to) level indicators, they can also be used independently to describe internal data or data common to two or more programs. (See “Level-numbers” on page 182 for level-number rules.)

Levels of data

After a record has been defined, it can be subdivided to provide more detailed data references.

For example, in a customer file for a department store, one complete record could contain all data that pertains to one customer. Subdivisions within that record could be, for example, customer name, customer address, account number, department number of sale, unit amount of sale, dollar amount of sale, previous balance, and other pertinent information.

The basic subdivisions of a record (that is, those fields not further subdivided) are called *elementary items*. Thus a record can be made up of a series of elementary items or can itself be an elementary item.

It might be necessary to refer to a set of elementary items; thus, elementary items can be combined into *group items*. Groups can also be combined into a more inclusive group that contains one or more subgroups. Thus within one hierarchy of data items, an elementary item can belong to more than one group item.

A system of level-numbers specifies the organization of elementary and group items into records. Special level-numbers are also used to identify data items used for special purposes.

Levels of data in a record description entry

Each group and elementary item in a record requires a separate entry, and each must be assigned a level-number.

A level-number is a one-digit or two-digit integer between 01 and 49, or one of three special level-numbers: 66, 77, or 88. The following level-numbers are used to structure records:

01 This level-number specifies the record itself, and is the most inclusive level-number possible. A level-01 entry can be an alphanumeric group item, a national group item, or an elementary item. The level number must begin in Area A.

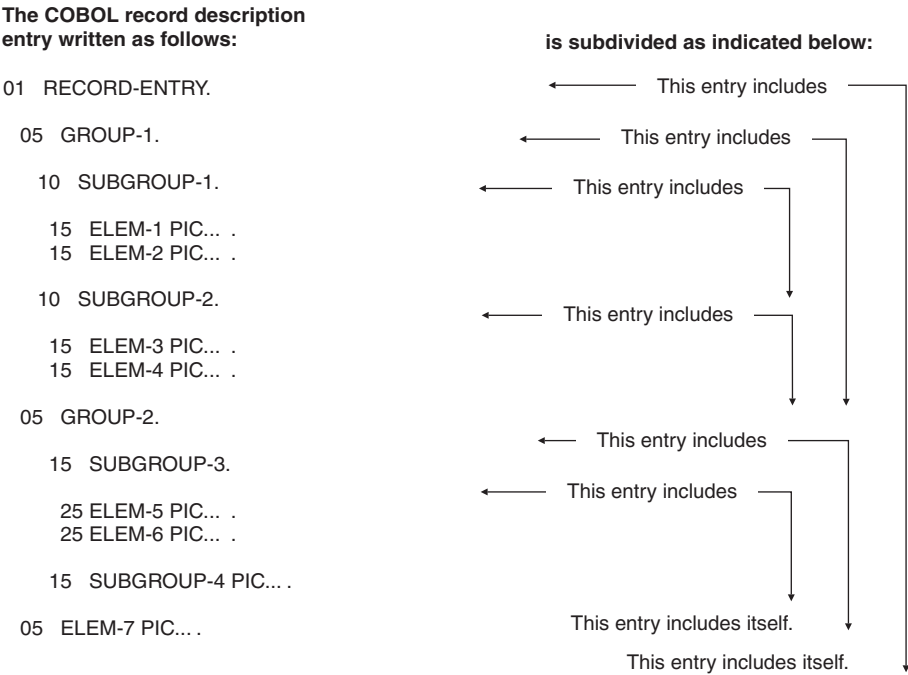
02 through 49

These level-numbers specify group and elementary items within a record. They can begin in Area A or Area B. Less inclusive data items are assigned higher (not necessarily consecutive) level-numbers in this series.

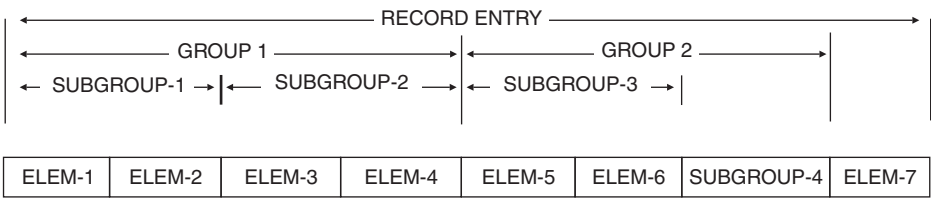
The relationship between level-numbers within a group item defines the hierarchy of data within that group.

A group item includes all group and elementary items that follow it until a level-number less than or equal to the level-number of that group is encountered.

The following figure illustrates a group wherein all groups immediately subordinate to the level-01 entry have the same level-number.



The storage arrangement of the record description entry is illustrated below:



You can also define groups with subordinate items that have different level-numbers for the same level in the hierarchy. For example, 05 EMPLOYEE-NAME and 04 EMPLOYEE-ADDRESS in EMPLOYEE-RECORD below define the same level in the hierarchy. The compiler rennumbers the levels in a relative fashion, as shown in MAP output.

```
01 EMPLOYEE-RECORD.  
  05 EMPLOYEE-NAME.  
    10 FIRST-NAME PICTURE X(10).  
    10 LAST-NAME PICTURE X(10).  
  04 EMPLOYEE-ADDRESS.  
    08 STREET PICTURE X(10).  
    08 CITY PICTURE X(10).
```

The following record description entry defines the same data hierarchy as the preceding record description entry:

```

01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        03  FIRST-NAME PICTURE  X(10).
        03  LAST-NAME  PICTURE  X(10).
    02  EMPLOYEE-ADDRESS.
        03  STREET      PICTURE  X(10).
        03  CITY        PICTURE  X(10).

```

Elementary items can be specified at any level within the hierarchy.

Special level-numbers

Special level-numbers identify items that do not structure a record. The special level-numbers are:

- 66** Identifies items that must contain a RENAME clause; such items regroup previously defined data items. (For details, see “RENAME clause” on page 220.)
- 77** Identifies data item description entries that are independent working-storage, local-storage, or linkage section items; they are not subdivisions of other items and are not subdivided themselves. Level-77 items must begin in Area A.
- 88** Identifies any condition-name entry that is associated with a particular value of a conditional variable. (For details, see “VALUE clause” on page 239.)

Level-77 and level-01 entries in the working-storage, local-storage, and linkage sections that are referenced in a program or method must be given unique data-names because level-77 and level-01 entries cannot be qualified. Subordinate data-names that are referenced in the program or method must be either uniquely defined, or made unique through qualification. Unreferenced data-names need not be uniquely defined.

Indentation

Successive data description entries can begin in the same column as preceding entries, or can be indented.

Indentation is useful for documentation but does not affect the action of the compiler.

Classes and categories of group items

COBOL for AIX has two types of groups: alphanumeric groups and national groups.

Groups that do not specify a GROUP-USAGE clause are alphanumeric groups. An alphanumeric group has class and category alphanumeric and is treated as though its usage were DISPLAY, regardless of the representation of the elementary data items that are contained within the group. In many operations, such as moves and compares, alphanumeric groups are treated as though they were elementary items of category alphanumeric, except that no editing or conversion of data representation takes place. In other operations, such as MOVE CORRESPONDING and ADD CORRESPONDING, the subordinate data items are processed as separate elementary items.

The content of an alphanumeric group is treated as though it were represented in native single-byte characters when the CHAR(NATIVE) compiler option is used, and as single-byte EBCDIC characters when the CHAR(EBCDIC) compiler option is used.

National groups are defined by a GROUP-USAGE clause with the NATIONAL phrase at the group level. All subordinate data items must be explicitly or implicitly described with usage NATIONAL, and subordinate groups must be explicitly or implicitly described with GROUP-USAGE NATIONAL.

Unless stated otherwise, a national group item is processed exactly as though it were an elementary data item of usage national, class and category national, described with PICTURE N(m), where m is the length of the group in national character positions. Because national groups contain only national characters, data is converted as necessary for moves and compares. The compiler ensures proper truncation and padding. In other operations, such as MOVE CORRESPONDING and ADD CORRESPONDING, the subordinate data items are processed as separate elementary items. See “GROUP-USAGE clause” on page 191 for details.

The table below summarizes the classes and categories of group items.

Table 10. Classes and categories of group items

Group description	Class of group	Category of group	USAGE of elementary items within a group	USAGE of a group
Without a GROUP-USAGE clause	Alphanumeric	Alphanumeric (even though the elementary items in the group can have any category)	Any	Treated as DISPLAY when usage is relevant
With explicit or implicit GROUP-USAGE clause	National	National	NATIONAL	NATIONAL

Classes and categories of data

Most data and all literals used in a COBOL program are divided into classes and categories. Data classes are groupings of data categories. Data categories are determined by the attributes of data description entries or function definitions, as described in “Category descriptions” on page 162.

The following elementary data items do not have a class and category:

- Index data items
- Items described with USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE

All other types of elementary data items have a class and category as shown in Table 11 on page 161.

A function references an elementary data item and belongs to the data class and category associated with the type of the function, as shown in Table 12.

Literals have a class and category as shown in Table 13 on page 162. Figurative constants (except NULL) have a class and category that depends on the literal or value represented by the figurative constant in the context of its use. For details, see “Figurative constants” on page 13.

All group items have a class and category, even if the subordinate elementary items belong to another class and category. For the classification of group items, see “Classes and categories of group items” on page 159.

Table 11. Class, category, and usage of elementary data items

Class	Category	Usage
Alphabetic	Alphabetic	DISPLAY
Alphanumeric	Alphanumeric	DISPLAY
	Alphanumeric-edited	DISPLAY
	Numeric-edited	DISPLAY
DBCS	DBCS	DISPLAY-1
National	National	NATIONAL
	National-edited	NATIONAL
	Numeric-edited	NATIONAL
Numeric	Numeric	DISPLAY (type zoned decimal)
		NATIONAL (type national decimal)
		PACKED-DECIMAL (type internal decimal)
		COMP-3 (type internal decimal)
		BINARY
		COMP
		COMP-4
		COMP-5
	Internal floating-point	COMP-1
		COMP-2
	External floating-point	DISPLAY
		NATIONAL

Table 12. Classes and categories of functions

Function type	Class and category
Alphanumeric	Alphanumeric
National	National
Integer	Numeric
Numeric	Numeric

Table 13. Classes and categories of literals

Literal	Class and category
Alphanumeric (including hexadecimal formats)	Alphanumeric
DBCS	DBCS
National (including hexadecimal formats)	National
Numeric (fixed-point and floating-point)	Numeric

Category descriptions

The category of a data item is established by the attributes of its data description entry (such as its PICTURE character-string or USAGE clause) or by its function definition. The meaning of each category is given below.

Alphabetic

A data item is described as category alphabetic by its PICTURE character-string. For PICTURE character-string details, see “Alphabetic items” on page 204.

A data item of category alphabetic is referred to as an alphabetic data item.

Alphanumeric

Each of the following is a data item of category alphanumeric:

- An elementary data item described as alphanumeric by its PICTURE character-string. For PICTURE character-string details, see “Alphanumeric items” on page 206.
- An alphanumeric group item.
- An alphanumeric function.
- The following special registers:
 - DEBUG-ITEM
 - SHIFT-OUT
 - SHIFT-IN
 - SORT-CONTROL
 - SORT-MESSAGE
 - WHEN-COMPILED
 - XML-EVENT
 - XML-TEXT

Alphanumeric-edited

A data item is described as category alphanumeric-edited by its PICTURE character-string. For PICTURE character-string details, see “Alphanumeric-edited items” on page 207.

A data item of category alphanumeric-edited is referred to as an alphanumeric-edited data item.

DBCS

A data item is described as category DBCS by its PICTURE character-string and the NSYMBOL(DBCS) compiler option or by an explicit USAGE DISPLAY-1 clause. For PICTURE character-string details, see “DBCS items” on page 207.

A data item of category DBCS is referred to as a DBCS data item.

External floating-point

A data item is described as category external floating-point by its PICTURE character-string. For PICTURE character-string details, see “External floating-point items” on page 210. An external floating-point data item can be described with USAGE DISPLAY or USAGE NATIONAL.

When the usage is DISPLAY, the item is referred to as a display floating-point data item.

When the usage is NATIONAL, the item is referred to as a national floating-point data item.

An external floating-point data item is of class numeric and, unless specifically excluded, is included in a reference to a numeric data item.

Internal floating-point

A data item is described as category internal floating-point by a USAGE clause with the COMP-1 or COMP-2 phrase.

A data item of category internal floating-point is referred to as an internal floating-point data item. An internal floating-point data item is of class numeric and, unless specifically excluded, is included in a reference to a numeric data item.

National

Each of the following is a data item of category national:

- A data item that is described as category national by its PICTURE character-string and the NSYMBOL(NATIONAL) compiler option or by an explicit USAGE NATIONAL clause. For PICTURE character-string details, see “National items” on page 208.
- A group item explicitly or implicitly described with a GROUP-USAGE NATIONAL clause.
- A national function.
- The special register XML-NTEXT.

National-edited

A data item is described as category national-edited by its PICTURE character-string. For PICTURE character-string details, see “National-edited items” on page 209.

A data item of category national-edited is referred to as a national-edited data item.

Numeric

Each of the following is a data item of category numeric:

- An elementary data item described as numeric by its PICTURE character-string and not described with a BLANK WHEN ZERO clause. For PICTURE character-string details, see “Numeric items” on page 205.
- An elementary data item described with one of the following usages:
 - BINARY, COMPUTATIONAL, COMPUTATIONAL-4, COMPUTATIONAL-5, COMP, COMP-4, or COMP-5
 - PACKED-DECIMAL, COMPUTATIONAL-3, or COMP-3
- A special register of numeric type:
 - LENGTH OF
 - LINAGE-COUNTER
 - RETURN-CODE
 - SORTCORE-SIZE
 - SORT-FILE-SIZE
 - SORT-MODE-SIZE
 - SORT-RETURN
 - TALLY
 - XML-CODE
- A numeric function.
- An integer function.

A data item of category numeric is referred to as a numeric data item.

Numeric-edited

Each of the following is a data item of category numeric-edited:

- A data item described as numeric-edited by its PICTURE character-string. For PICTURE character-string details, see “Numeric-edited items” on page 206.
- A data item described as numeric by its PICTURE character-string and described with a BLANK WHEN ZERO clause.

Alignment rules

The standard alignment rules for positioning data in an elementary item depend on the category of a receiving item (that is, an item into which the data is moved; see “Elementary moves” on page 381).

Numeric

For numeric receiving items, the following rules apply:

1. The data is aligned on the assumed decimal point and, if necessary, truncated or padded with zeros. (An *assumed decimal point* is one that has logical meaning but that does not exist as an actual character in the data.)
2. If an assumed decimal point is not explicitly specified, the receiving item is treated as though an assumed decimal point is specified immediately to the right of the field. The data is then treated according to the preceding rule.

Numeric-edited

The data is aligned on the decimal point, and (if necessary) truncated or padded with zeros at either end except when editing causes replacement of leading zeros.

Internal floating-point

A decimal point is assumed immediately to the left of the field. The data is then aligned on the leftmost digit position that follows the decimal point, with the exponent adjusted accordingly.

External floating-point

The data is aligned on the leftmost digit position; the exponent is adjusted accordingly.

Alphanumeric, alphanumeric-edited, alphabetic, DBCS

For these receiving items, the following rules apply:

1. The data is aligned at the leftmost character position, and (if necessary) truncated or padded with spaces at the right.
2. If the JUSTIFIED clause is specified for this receiving item, the above rule is modified as described in "JUSTIFIED clause" on page 190.

National, national-edited

For these receiving items, the following rules apply:

1. The data is aligned at the leftmost character position, and (if necessary) truncated or padded with default Unicode spaces (NX'0020') at the right. Truncation occurs at the boundary of a national character position.
2. If the JUSTIFIED clause is specified for this receiving item, the above rule is modified as described in "JUSTIFIED clause" on page 190.

Character-string and item size

For items described with a PICTURE clause, the size of an elementary item is expressed in source code by the number of character positions described in the PICTURE character-string and a SIGN clause (if applicable). Storage size, however, is determined by the actual number of bytes the item occupies as determined by the combination of its PICTURE character-string, SIGN IS SEPARATE clause (if specified), and USAGE clause.

For items described with USAGE DISPLAY (categories alphabetic, alphanumeric, alphanumeric-edited, numeric-edited, numeric, and external floating-point), 1 byte of storage is reserved for each character position described by the item's PICTURE character-string and SIGN IS SEPARATE clause (if applicable).

For items described with USAGE DISPLAY-1 (category DBCS), 2 bytes of storage are reserved for each character position described by the item's PICTURE character-string.

For items described with USAGE NATIONAL (categories national, national-edited, numeric-edited, numeric, and external floating-point), 2 bytes of storage are reserved for each character position described by the item's PICTURE character-string and SIGN IS SEPARATE clause (if specified).

For internal floating-point items, the size of the item in storage is determined by its USAGE clause. USAGE COMPUTATIONAL-1 reserves 4 bytes of storage for the item; USAGE COMPUTATIONAL-2 reserves 8 bytes of storage.

Normally, when an arithmetic item is moved from a longer field into a shorter one, the compiler truncates the data to the number of digits represented in the shorter item's PICTURE character-string by truncating leading digits. For example, if a sending field with PICTURE S99999 that contains the value +12345 is moved to a BINARY receiving field with PICTURE S99, the data is truncated to +45. For additional information, see “USAGE clause” on page 229.

The TRUNC compiler option can affect the value of a binary numeric item. For information about TRUNC, see *TRUNC* in the *COBOL for AIX Programming Guide*.

Signed data

There are two categories of algebraic signs used in COBOL: operational signs and editing signs.

Operational signs

Operational signs are associated with signed numeric items, and indicate their algebraic properties. The internal representation of an algebraic sign depends on the item's USAGE clause, its SIGN clause (if present), and the operating environment. (For further details about the internal representation, see *Examples: number data and internal representation* in the *COBOL for AIX Programming Guide*.)

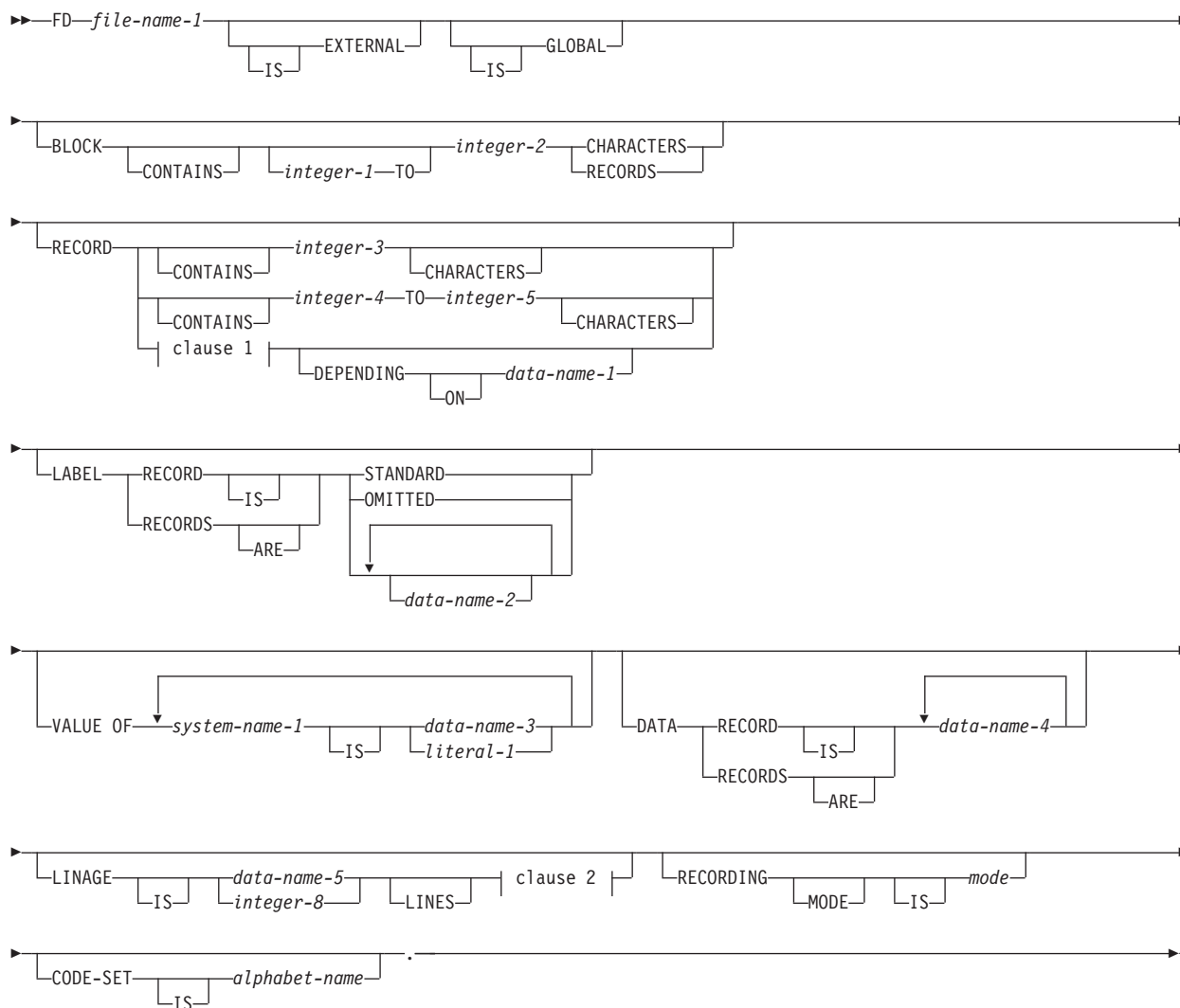
Editing signs

Editing signs are associated with numeric-edited items. Editing signs are PICTURE symbols that identify the sign of the item in edited output.

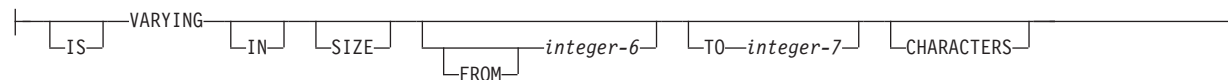
Chapter 18. Data division--file description entries

In a COBOL program, the *File Description (FD) Entry* (or *Sort File Description (SD) Entry* for sort/merge files) represents the highest level of organization in the file section. The order in which the optional clauses follow the FD or SD entry is not important.

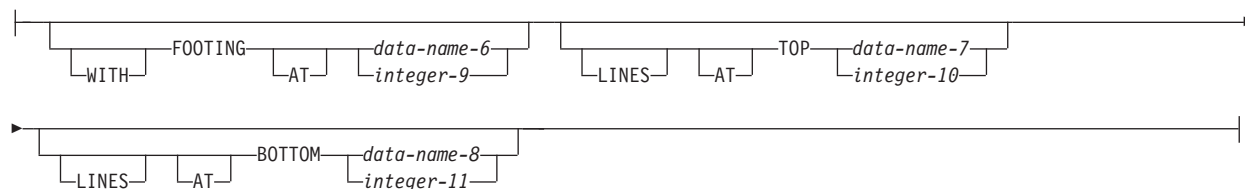
Format 1: sequential file description entry



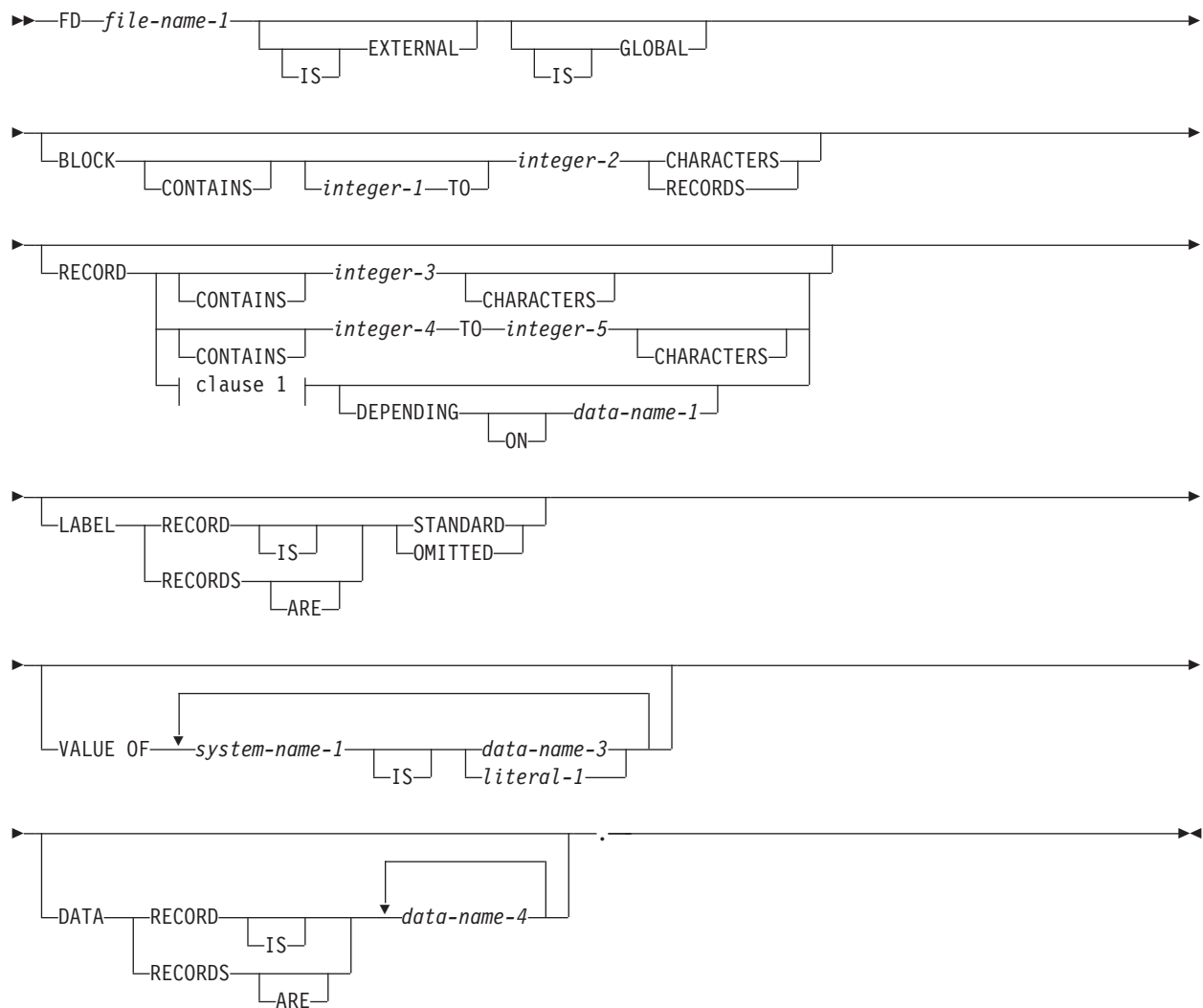
clause 1:



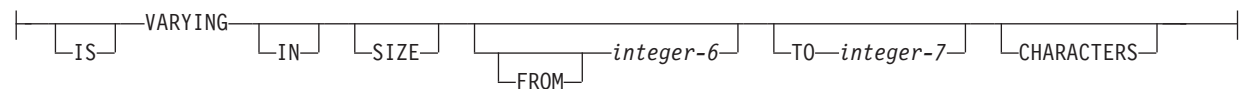
clause 2:



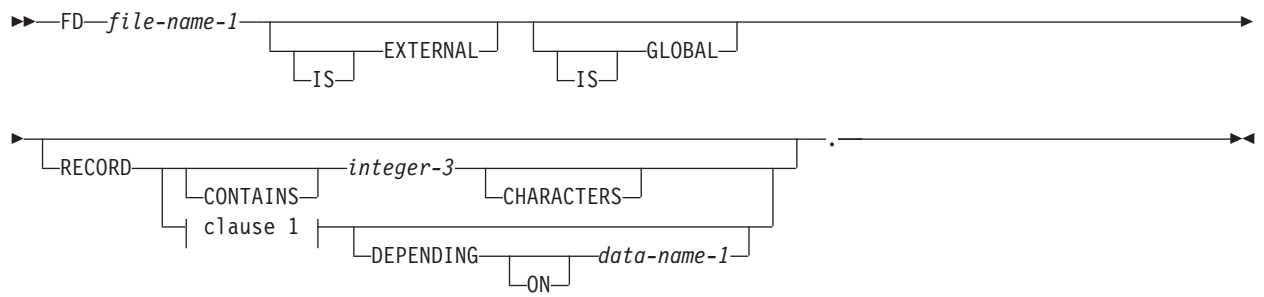
Format 2: relative or indexed file description entry



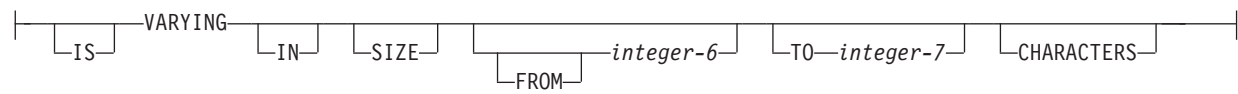
clause 1:



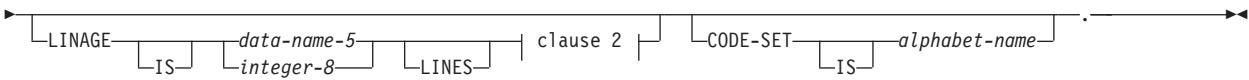
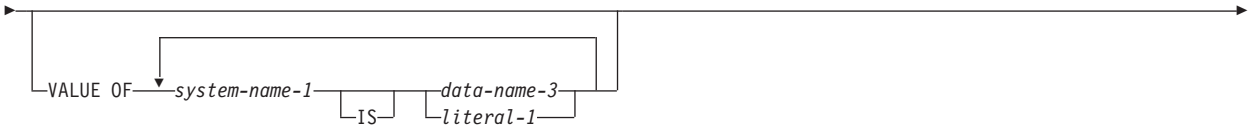
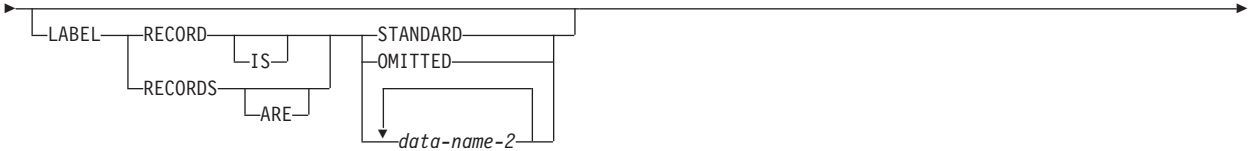
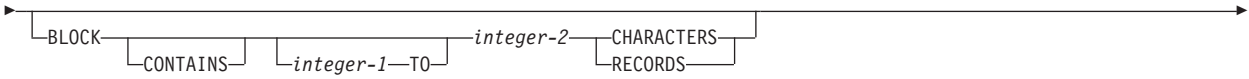
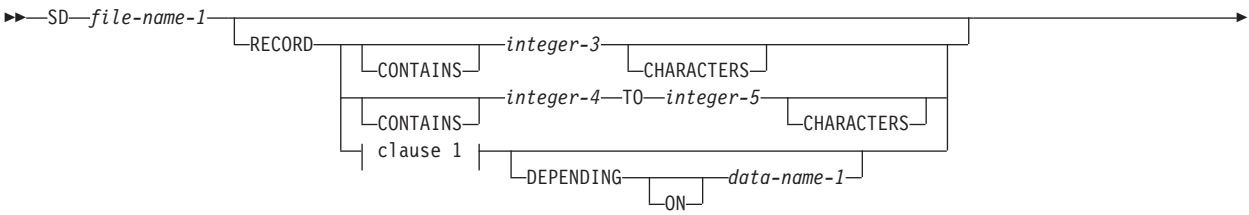
Format 3: line-sequential file description entry



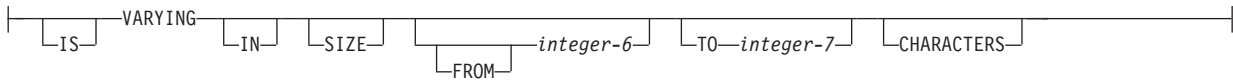
clause 1:



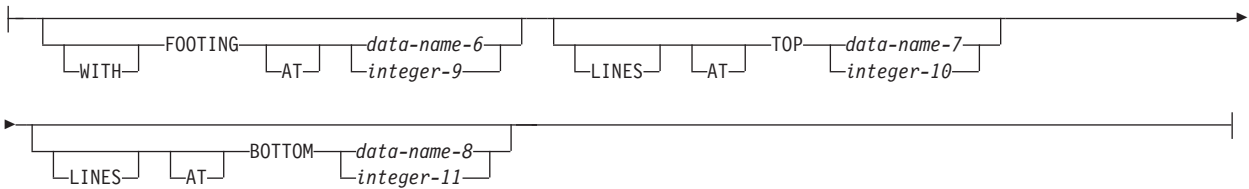
Format 4: sort/merge file description entry



clause 1:



clause 2:



File section

The file section must contain a level-indicator for each input and output file:

- For all files except sort/merge files, the file section must contain an FD entry.
- For each sort or merge file, the file section must contain an SD entry.

file-name

Must follow the level indicator (FD or SD), and must be the same as that specified in the associated SELECT clause. *file-name* must adhere to the rules of formation for a user-defined word; at least one character must be alphabetic. *file-name* must be unique within this program.

One or more record description entries must follow *file-name*. When more than one record description entry is specified, each entry implies a redefinition of the same storage area.

The clauses that follow *file-name* are optional, and they can appear in any order.

FD (formats 1, 2, and 3)

The last clause in the FD entry must be immediately followed by a separator period.

SD (format 4)

An SD entry must be written for each sort or merge file in the program. The last clause in the SD entry must be immediately followed by a separator period.

The following example illustrates the file section entries needed for a sort or merge file:

```
SD  SORT-FILE.  
01  SORT-RECORD  PICTURE X(80).
```

A record in the file section must be described as an alphanumeric group item, a national group item, or an elementary item of class alphabetic, alphanumeric, DBCS, national, or numeric.

All the record description entries under the file description entry (FD) for an RSD file must be described with the same number of bytes. The following example illustrates the file section for an RSD file with level 01 records that are of a fixed length:

```
FILE SECTION.  
FD an-RSD-file . . . RECORD CONTAINS 80 CHARACTERS.  
01 record-1 PIC X(80).  
01 record-2.  
    02 PIC X(10).  
    02 PIC X(70).
```

EXTERNAL clause

The EXTERNAL clause specifies that a file connector is external, and permits communication between two programs by the sharing of files. A file connector is external if the storage associated with that file is associated with the run unit rather than with any particular program within the run unit. An external file can be referenced by any program in the run unit that describes the file. References to an external file from different programs that use separate descriptions of the file are always to the same file. In a run unit, there is only one representative of an external file.

In the file section, the EXTERNAL clause can be specified only in file description entries.

The records appearing in the file description entry need not have the same name in corresponding external file description entries. In addition, the number of such records need not be the same in corresponding file description entries.

Use of the EXTERNAL clause does not imply that the associated file-name is a global name. See *Sharing data by using the EXTERNAL clause* in the *COBOL for AIX Programming Guide* for specific information about the use of the EXTERNAL clause.

GLOBAL clause

The GLOBAL clause specifies that the file connector named by a file-name is a global name. A global file-name is available to the program that declares it and to every program that is contained directly or indirectly in that program.

A file-name is global if the GLOBAL clause is specified in the file description entry for that file-name. A record-name is global if the GLOBAL clause is specified in the record description entry by which the record-name is declared or, in the case of record description entries in the file section, if the GLOBAL clause is specified in the file description entry for the file-name associated with the record description entry. For details on using the GLOBAL clause, see *Using data in input and output operations* and *Scope of names* in the *COBOL for AIX Programming Guide*.

Two programs in a run unit can reference global file connectors in the following circumstances:

- An external file connector can be referenced from any program that describes that file connector.
- If a program is contained within another program, both programs can refer to a global file connector by referring to an associated global file-name either in the containing program or in any program that directly or indirectly contains the containing program.

BLOCK CONTAINS clause

The BLOCK CONTAINS clause is syntax checked but has no effect on the execution of the program.

integer-1, integer-2

Must be nonzero unsigned integers. They specify:

CHARACTERS

Specifies the number of bytes required to store the physical record, no matter what USAGE the data items have within the data record.

If only *integer-2* is specified, it specifies the exact number of bytes in the physical record. When *integer-1* and *integer-2* are both specified, they represent the minimum and maximum number of bytes in the physical record, respectively.

integer-1 and *integer-2* must include any control bytes and padding contained in the physical record. (Logical records do not include padding.)

The CHARACTERS phrase is the default. CHARACTERS must be specified when:

- The physical record contains padding.
- Logical records are grouped so that an inaccurate physical record size could be implied. For example, suppose you describe a variable-length record of 100 bytes, yet each time you write a block of 4, one 50-byte record is written followed by three 100-byte records. If the RECORDS phrase were specified, the compiler would calculate the block size as 420 bytes instead of the actual size, 370 bytes. (This calculation includes block and record descriptors.)

RECORDS

Specifies the number of logical records contained in each physical record.

The compiler assumes that the block size must provide for *integer-2* records of maximum size, and provides any additional space needed for control bytes.

RECORD clause

When the RECORD clause is used, the record size must be specified as the number of bytes needed to store the record internally, regardless of the USAGE of the data items contained within the record.

For example, if you have a record with 10 DBCS characters, the RECORD clause should say RECORD CONTAINS 20 CHARACTERS. For a record with 10 national characters, the RECORD clause should say the same, RECORD CONTAINS 20 CHARACTERS.

The size of a record is determined according to the rules for obtaining the size of a group item. (See “USAGE clause” on page 229 and “SYNCHRONIZED clause” on page 223.)

When the RECORD clause is omitted, the compiler determines the record lengths from the record descriptions. When one of the entries within a record description contains an OCCURS DEPENDING ON clause, the compiler uses the maximum value of the variable-length item to calculate the number of bytes needed to store the record internally.

If the associated file connector is an external file connector, all file description entries in the run unit that are associated with that file connector must specify the same maximum number of bytes.

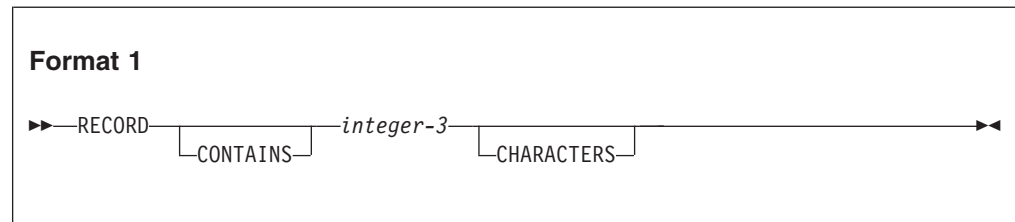
For an RSD file, the RECORD clause, if specified, must describe fixed-length records.

The following sections describe the formats of the RECORD clause:

- “Format 1” on page 175, fixed-length records
- “Format 2” on page 175, fixed-length or variable-length records
- “Format 3” on page 175, variable-length records

Format 1

Format 1 specifies the number of bytes for fixed-length records.



integer-3

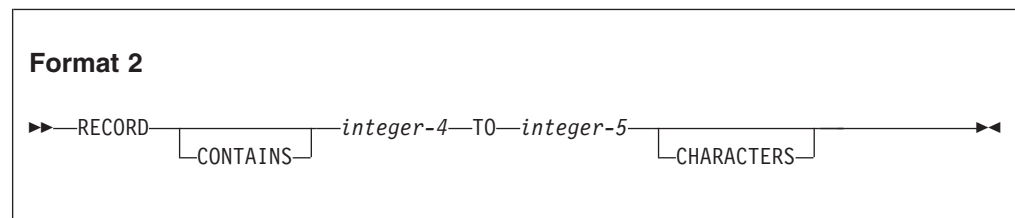
Must be an unsigned integer that specifies the number of bytes contained in each record in the file.

The RECORD CONTAINS 0 characters clause is syntax checked, but has no effect on the execution of the program.

Do not specify the RECORD CONTAINS 0 clause for an SD entry.

Format 2

Format 2 specifies the number of bytes for either fixed-length or variable-length records. Fixed-length records are obtained when all 01 record description entry lengths are the same. The format-2 RECORD CONTAINS clause is never required, because the minimum and maximum record lengths are determined from the record description entries.



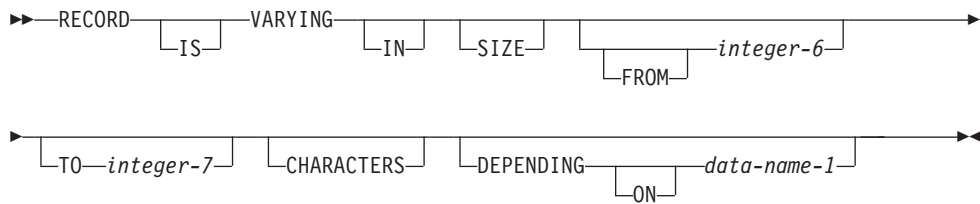
integer-4, integer-5

Must be unsigned integers. *integer-4* specifies the size of the smallest data record, and *integer-5* specifies the size of the largest data record.

Format 3

Format 3 is used to specify variable-length records.

Format 3



integer-6

Specifies the minimum number of bytes to be contained in any record of the file. If *integer-6* is not specified, the minimum number of bytes to be contained in any record of the file is equal to the least number of bytes described for a record in that file.

integer-7

Specifies the maximum number of bytes in any record of the file. If *integer-7* is not specified, the maximum number of bytes to be contained in any record of the file is equal to the greatest number of bytes described for a record in that file.

The number of bytes associated with a record description is determined by the sum of the number of bytes in all elementary data items (excluding redefinitions and renamings), plus any implicit FILLER due to synchronization. If a table is specified:

- The minimum number of table elements described in the record is used in the summation above to determine the minimum number of bytes associated with the record description.
- The maximum number of table elements described in the record is used in the summation above to determine the maximum number of bytes associated with the record description.

If *data-name-1* is specified:

- *data-name-1* must be an elementary unsigned integer.
- *data-name-1* cannot be a windowed date field.
- The number of bytes in the record must be placed into the data item referenced by *data-name-1* before any RELEASE, REWRITE, or WRITE statement is executed for the file.
- The execution of a DELETE, RELEASE, REWRITE, START, or WRITE statement or the unsuccessful execution of a READ or RETURN statement does not alter the content of the data item referenced by *data-name-1*.
- After the successful execution of a READ or RETURN statement for the file, the contents of the data item referenced by *data-name-1* indicate the number of bytes in the record just read.

During the execution of a RELEASE, REWRITE, or WRITE statement, the number of bytes in the record is determined by the following conditions:

- If *data-name-1* is specified, by the content of the data item referenced by *data-name-1*
- If *data-name-1* is not specified and the record does not contain a variable occurrence data item, by the number of bytes positions in the record

- If *data-name-1* is not specified and the record contains a variable occurrence data item, by the sum of the fixed position and that portion of the table described by the number of occurrences at the time of execution of the output statement

During the execution of a READ ... INTO or RETURN ... INTO statement, the number of bytes in the current record that participate as the sending data items in the implicit MOVE statement is determined by the following conditions:

- If *data-name-1* is specified, by the content of the data item referenced by *data-name-1*
- If *data-name-1* is not specified, by the value that would have been moved into the data item referenced by *data-name-1* had *data-name-1* been specified

LABEL RECORDS clause

The LABEL RECORDS clause is syntax checked, but has no effect on the execution of the program. A warning message is issued if you use any of the following language elements:

- LABEL RECORD IS *data-name*
- USE ... AFTER ... LABEL PROCEDURE
- GO TO MORE-LABELS

The LABEL RECORDS clause indicates the presence or absence of labels. If it is not specified for a file, label records for that file must conform to the system label specifications.

STANDARD

Labels conforming to system specifications exist for this file.

STANDARD is permitted for mass storage devices and tape devices.

OMITTED

No labels exist for this file.

OMITTED is permitted for tape devices.

data-name-2

User labels are present in addition to standard labels. *data-name-2* specifies the name of a user label record. *data-name-2* must appear as the subject of a record description entry associated with the file.

A LABEL RECORDS clause under an SD is syntax checked, but has no effect on the execution of the program.

VALUE OF clause

The VALUE OF clause describes an item in the label records associated with the file.

data-name-3

Should be qualified when necessary, but cannot be subscripted. It must be described in the working-storage section. It cannot be described with the USAGE IS INDEX clause.

literal-1

Can be numeric or alphanumeric, or a figurative constant of category numeric or alphanumeric. Cannot be a floating-point literal.

The VALUE OF clause is syntax checked, but has no effect on the execution of the program.

DATA RECORDS clause

The DATA RECORDS clause is syntax checked but serves only as documentation for the names of data records associated with the file.

data-name-4

The names of record description entries associated with the file.

The data-name need not have an associated 01 level number record description with the same name.

LINAGE clause

The LINAGE clause specifies the depth of a logical page in number of lines. Optionally, it also specifies the line number at which the footing area begins and the top and bottom margins of the logical page. (The logical page and the physical page cannot be the same size.)

The LINAGE clause is effective for sequential files opened as OUTPUT or EXTEND, except that the clause cannot be specified for RSD files.

All integers must be unsigned. All data-names must be described as unsigned integer data items.

data-name-5 , integer-8

The number of lines that can be written or spaced on this logical page. The area of the page that these lines represent is called the *page body*. The value must be greater than zero.

WITH FOOTING AT

integer-9 or the value of the data item in *data-name-6* specifies the first line number of the footing area within the page body. The footing line number must be greater than zero, and not greater than the last line of the page body. The footing area extends between those two lines.

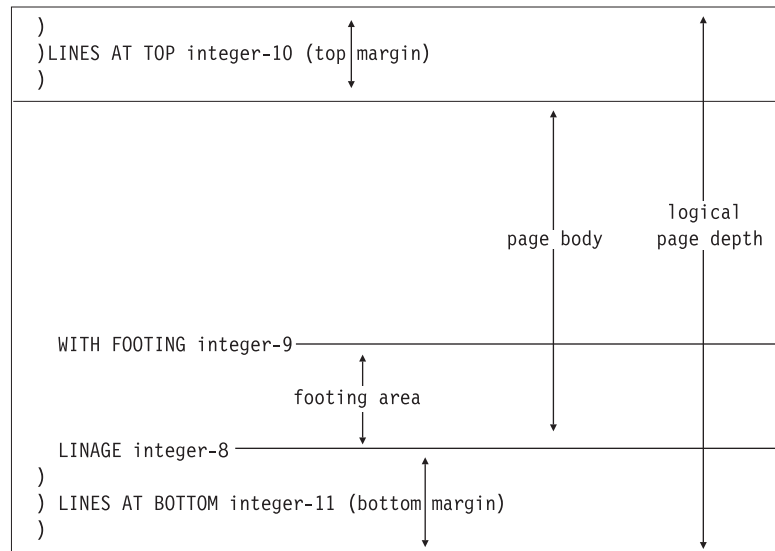
LINES AT TOP

integer-10 or the value of the data item in *data-name-7* specifies the number of lines in the top margin of the logical page. The value can be zero.

LINES AT BOTTOM

integer-11 or the value of the data item in *data-name-8* specifies the number of lines in the bottom margin of the logical page. The value can be zero.

The following figure illustrates the use of each phrase of the LINAGE clause.



The logical page size specified in the LINAGE clause is the sum of all values specified in each phrase except the FOOTING phrase. If the LINES AT TOP phrase is omitted, the assumed value for the top margin is zero. Similarly, if the LINES AT BOTTOM phrase is omitted, the assumed value for the bottom margin is zero. Each logical page immediately follows the preceding logical page, with no additional spacing provided.

If the FOOTING phrase is omitted, its assumed value is equal to that of the page body (*integer-8* or *data-name-5*).

At the time an OPEN OUTPUT statement is executed, the values of *integer-8*, *integer-9*, *integer-10*, and *integer-11*, if specified, are used to determine the page body, first footing line, top margin, and bottom margin of the logical page for this file. (See the figure above.) These values are then used for all logical pages printed for this file during a given execution of the program.

At the time an OPEN statement with the OUTPUT phrase is executed for the file, *data-name-5*, *data-name-6*, *data-name-7*, and *data-name-8* determine the page body, first footing line, top margin, and bottom margin for the first logical page only.

At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, the values of *data-name-5*, *data-name-6*, *data-name-7*, and *data-name-8* if specified, are used to determine the page body, first footing line, top margin, and bottom margin for the next logical page.

If an external file connector is associated with this file description entry, all file description entries in the run unit that are associated with this file connector must have:

- A LINAGE clause, if any file description entry has a LINAGE clause
- The same corresponding values for *integer-8*, *integer-9*, *integer-10*, and *integer-11*, if specified
- The same corresponding external data items referenced by *data-name-5*, *data-name-6*, *data-name-7*, and *data-name-8*

See “ADVANCING phrase” on page 465 for the behavior of carriage control characters in external files.

A LINAGE clause under an SD is syntax checked, but has no effect on the execution of the program.

LINAGE-COUNTER special register

For information about the LINAGE-COUNTER special register, see “LINAGE-COUNTER” on page 20.

RECORDING MODE clause

The RECORDING MODE clause for record sequential files is treated as follows:

- F** Record descriptions are validated as fixed. Do not specify RECORDING MODE F if the record descriptions are variable. This is the only valid value for RSD files.
- V** Variable-length record format is assumed (even if the record descriptions are fixed).
- U** Syntax checked, but has no effect on the execution of the program.
- S** Treated the same as V.

CODE-SET clause

The CODE-SET clause is syntax checked, but has no effect on the execution of the program.

Chapter 19. Data division--data description entry

A *data description entry* specifies the characteristics of a data item. In the sections that follow, sets of data description entries are called *record description entries*. The term *data description entry* refers to data and record description entries.

Data description entries that define independent data items do not make up a record. These entries are known as *data item description entries*.

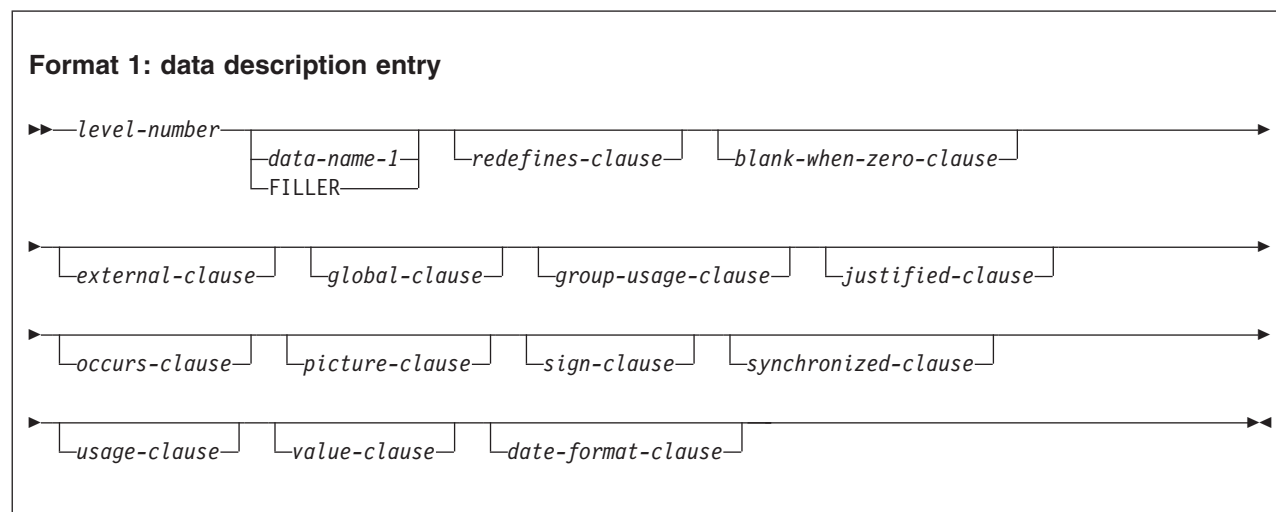
Data description entries have three general formats, which are described in the following sections:

- “Format 1”
- “Format 2” on page 182
- “Format 3” on page 182

All data description entries must end with a separator period.

Format 1

Format 1 is used for data description entries in all data division sections.



The clauses can be written in any order, with two exceptions:

- *data-name-1* or *FILLER*, if specified, must immediately follow the level-number.
- When the *REDEFINES* clause is specified, it must immediately follow *data-name-1* or *FILLER*, if either is specified. If *data-name-1* or *FILLER* is not specified, the *REDEFINES* clause must immediately follow the level-number.

The level-number in format 1 can be any number in the range 01–49, or 77.

A space, a comma, or a semicolon must separate clauses.

Format 2

Format 2 regroups previously defined items.

Format 2: renames

►►66—*data-name-1—renames-clause.*—————►◄

A level-66 entry cannot rename another level-66 entry, nor can it rename a level-01, level-77, or level-88 entry.

All level-66 entries associated with one record must immediately follow the last data description entry in that record.

See “RENAMES clause” on page 220 for further details.

Format 3

Format 3 describes condition-names.

Format 3: condition-name

►►88—*condition-name-1—value-clause.*—————►◄

condition-name-1

A user-specified name that associates a value, a set of values, or a range of values with a conditional variable.

Level-88 entries must immediately follow the data description entry for the conditional variable with which the condition-names are associated.

Format 3 can be used to describe elementary items, national group items, or alphanumeric group items. Additional information about condition-name entries can be found under “VALUE clause” on page 239 and “Condition-name condition” on page 265.

Level-numbers

The level-number specifies the hierarchy of data within a record, and identifies special-purpose data entries. A level-number begins a data description entry, a renamed or redefined item, or a condition-name entry. A level-number has an integer value between 1 and 49, inclusive, or one of the special level-number values 66, 77, or 88.

Format



level-number

01 and 77 must begin in Area A and be followed either by a separator period or by a space followed by its associated data-name, FILLER, or appropriate data description clause.

Level numbers 02 through 49 can begin in Areas A or B and must be followed by a space or a separator period.

Level numbers 66 and 88 can begin in Areas A or B and must be followed by a space.

Single-digit level-numbers 1 through 9 can be substituted for level-numbers 01 through 09.

Successive data description entries can start in the same column as the first entry or can be indented according to the level-number. Indentation does not affect the magnitude of a level-number.

When level-numbers are indented, each new level-number can begin any number of spaces to the right of Area A. The extent of indentation to the right is limited only by the width of Area B.

For more information, see “Levels of data” on page 157.

data-name-1

Explicitly identifies the data being described.

data-name-1, if specified, identifies a data item used in the program. *data-name-1* must be the first word following the level-number.

The data item can be changed during program execution.

data-name-1 must be specified for level-66 and level-88 items. It must also be specified for any entry containing the GLOBAL or EXTERNAL clause, and for record description entries associated with file description entries that have the GLOBAL or EXTERNAL clauses.

FILLER

A data item that is not explicitly referred to in a program. The keyword FILLER is optional. If specified, FILLER must be the first word following the level-number.

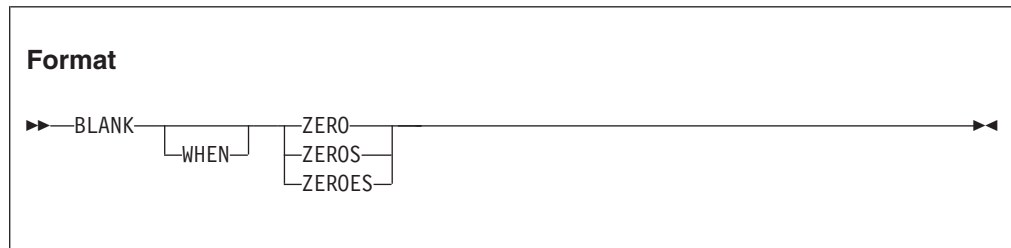
The keyword FILLER can be used with a conditional variable if explicit reference is never made to the conditional variable but only to values that it can assume. FILLER cannot be used with a condition-name.

In a MOVE CORRESPONDING statement or in an ADD CORRESPONDING or SUBTRACT CORRESPONDING statement, FILLER items are ignored. In an INITIALIZE statement, elementary FILLER items are ignored.

If *data-name-1* or the FILLER clause is omitted, the data item being described is treated as though FILLER had been specified.

BLANK WHEN ZERO clause

The BLANK WHEN ZERO clause specifies that an item contains only spaces when its value is zero.



The BLANK WHEN ZERO clause may be specified only for an elementary item described by its picture character string as category numeric-edited or numeric, without the picture symbol S or *. These items must be described, either implicitly or explicitly, as USAGE DISPLAY or USAGE NATIONAL.

A BLANK WHEN ZERO clause that is specified for an item defined as numeric by its picture character string defines the item as category numeric-edited.

The BLANK WHEN ZERO clause must not be specified for date fields.

DATE FORMAT clause

The DATE FORMAT clause specifies that a data item is a windowed or expanded date field:

Windowed date fields

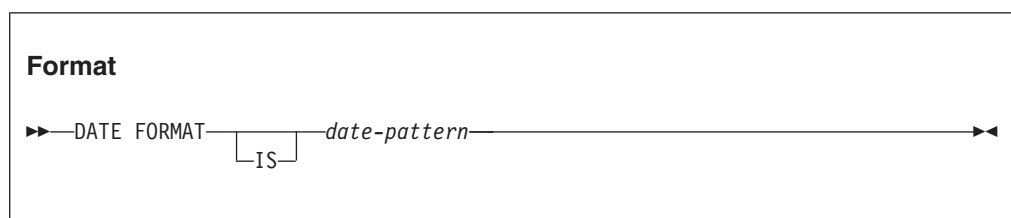
Contain a windowed (two-digit) year, specified by a DATE FORMAT clause that contains YY.

Expanded date fields

Contain an expanded (four-digit) year, specified by a DATE FORMAT clause that contains YYYY.

If the NODATEPROC compiler option is in effect, the DATE FORMAT clause is syntax checked but has no effect on the execution of the program. NODATEPROC disables date processing. The rules and restrictions described in this reference for the DATE FORMAT clause and date fields apply only if the DATEPROC compiler option is in effect.

The DATE FORMAT clause must not be specified for a data item described with USAGE NATIONAL.



date-pattern is a character string, such as YYXXXX, that represents a windowed or expanded year optionally followed or preceded by one to four characters representing other parts of a date such as the month and day:

Date-pattern string	Specifies that the data item contains
YY	A windowed (two-digit) year
YYYY	An expanded (four-digit) year
X	A single character; for example, a digit representing a semester or quarter (1–4)
XX	Two characters; for example, digits representing a month (01–12)
XXX	Three characters; for example, digits representing a day of the year (001–366)
XXXX	Four characters; for example, two digits representing a month (01–12) and two digits representing a day of the month (01–31)

For an introduction to date fields and related terms, see Chapter 10, “Millennium Language Extensions and date fields,” on page 75. For details on using date fields in applications, see *Millennium language extensions (MLE)* in the *COBOL for AIX Programming Guide*.

Semantics of windowed date fields

Windowed date fields undergo automatic expansion relative to the century window when they are used as operands in arithmetic expressions or arithmetic statements. However, the result of incrementing or decrementing a windowed date is still treated as a windowed date for further computation, comparison, and storing.

When used in the following situations, windowed date fields are treated as if they were converted to expanded date format:

- Operands in subtractions in which the other operand is an expanded date
- Operands in relation conditions
- Sending fields in arithmetic or MOVE statements

The details of the conversion to expanded date format depend on whether the windowed date field is numeric or alphanumeric.

Given a century window starting year of 19 nn , the year part (yy) of a numeric windowed date field is treated as if it were expanded as follows:

- If yy is less than nn , then add 2000 to yy .
- If yy is equal to or greater than nn , then add 1900 to yy .

For signed numeric windowed date fields, this means that there can be two representations of some years. For instance, windowed year values 99 and -01 are both treated as 1999, since $1900 + 99 = 2000 + -01$.

Alphanumeric windowed date fields are treated in a similar manner, but use a prefix of 19 or 20 instead of adding 1900 or 2000.

For example, when used as an operand of a relation condition, a windowed date field defined by:

```
01 DATE-FIELD DATE FORMAT YYXXXX PICTURE 9(6)
   VALUE IS 450101.
```

is treated as if it were an expanded date field with a value of:

- 19450101, if the century window starting year is 1945 or earlier
- 20450101, if the century window starting year is later than 1945

Restrictions on using date fields

The following sections describe restrictions on using date fields in these contexts:

- DATE FORMAT clauses combined with other clauses
- Group items consisting only of a date field
- Language elements that treat date fields as nondates
- Language elements that do not accept date fields as arguments

For restrictions on using date fields in other contexts, see:

- “Arithmetic with date fields” on page 260
- “Comparison of date fields” on page 276
- “ADD statement” on page 308
- “SUBTRACT statement” on page 450
- “MOVE statement” on page 380

Combining the DATE FORMAT clause with other clauses

The following phrases are the only phrases of the USAGE clause that can be combined with the DATE FORMAT clause:

- BINARY
- COMPUTATIONAL¹
- COMPUTATIONAL-3
- COMPUTATIONAL-4
- DISPLAY
- PACKED-DECIMAL

¹USAGE COMPUTATIONAL cannot be combined with the DATE FORMAT clause if the TRUNC(BIN) compiler option is in effect.

The PICTURE character-string must specify the same number of characters or digits as the DATE FORMAT clause. For alphanumeric date fields, the only PICTURE character-string symbols allowed are A, 9, and X, with at least one X. For numeric date fields, the only PICTURE character-string symbols allowed are 9 and S.

The following clauses are not allowed for a data item defined with DATE FORMAT:

- BLANK WHEN ZERO
- JUSTIFIED
- SEPARATE CHARACTER phrase of the SIGN clause

The EXTERNAL clause is not allowed for a windowed date field or a group item that contains a windowed date field subordinate item.

Some restrictions apply when combining the following clauses with DATE FORMAT:

- “REDEFINES clause” on page 216

- “VALUE clause” on page 239

Group items that are date fields

If a group item is defined with a DATE FORMAT clause, then the following restrictions apply:

- The elementary items in the group must all be USAGE DISPLAY.
- The length of the group item must be the same number of characters as the *date-pattern* in the DATE FORMAT clause.
- If the group consists solely of a date field with USAGE DISPLAY, and both the group and the single subordinate item have DATE FORMAT clauses, then the DATE FORMAT clauses must be identical.
- If the group item contains subordinate items that subdivide the group, then the following restrictions apply:
 - If a named (not FILLER) subordinate item consists of exactly the year part of the group item date field and has a DATE FORMAT clause, then the DATE FORMAT clause must be YY or YYYY with the same number of year characters as the group item.
 - If the group item is a Gregorian date with a DATE FORMAT clause of YYXXXX, YYYYXXXX, XXXXYY, or XXXXYYYY, and a named subordinate date data item consists of the year and month part of the Gregorian date, then its DATE FORMAT clause must be YYXX, YYYYXX, XXYY, or XYYYYY, respectively (or, for a group date format of YYYYXXXX, a subordinate date format of YYXX as described below).
 - A windowed date field can be subordinate to an expanded date field group item if the subordinate item starts two characters after the group item, neither date is in year-last format, and the date format of the subordinate item either has no Xs or has the same number of Xs following the Ys as the group item, or is YYXX under a group date format of YYYYXXXX.
 - The only subordinate items that can have a DATE FORMAT clause are those that define the year part of the group item, the windowed part of an expanded date field group item, or the year and month part of a Gregorian date group item, as discussed in the above restrictions.

For example, the following defines a valid group item:

```
01 YYMDD    DATE FORMAT YYXXXX.
   02 YYMM   DATE FORMAT YYXX.
       03 YY DATE FORMAT YY PICTURE 99.
       03                PICTURE 99.
   02 DD                PICTURE 99.
```

Language elements that treat date fields as nondates

If date fields are used in the following language elements, they are treated as nondates. That is, the DATE FORMAT is ignored, and the content of the date data item is used without undergoing automatic expansion.

- In the environment division file-control paragraph:
 - SELECT ... ASSIGN USING data-name
 - SELECT ... PASSWORD IS data-name
 - SELECT ... FILE STATUS IS data-name
- In data division entries:
 - LABEL RECORD IS data-name
 - LABEL RECORDS ARE data-name

- LINAGE IS data-name FOOTING data-name TOP data-name BOTTOM data-name
- In class conditions
- In sign conditions
- In DISPLAY statements

Language elements that do not accept windowed date fields as arguments

Windowed date fields cannot be used as:

- A data-name in the following formats of the environment division file-control paragraph:
 - SELECT ... RECORD KEY IS
 - SELECT ... ALTERNATE RECORD KEY IS
 - SELECT ... RELATIVE KEY IS
- A data-name in the RECORD IS VARYING DEPENDING ON clause of a data division file description (FD) or sort description (SD) entry
- The object of an OCCURS DEPENDING ON clause of a data division data definition entry
- The key in an ASCENDING KEY or DESCENDING KEY phrase of an OCCURS clause of a data division data definition entry
- Any data-name or identifier in the following statements:
 - CANCEL
 - GO TO ... DEPENDING ON
 - INSPECT
 - SET
 - SORT
 - STRING
 - UNSTRING
- In the CALL statement, as the identifier containing the program-name
- In the INVOKE statement, as the identifier specifying the object on which the method is invoked, or the identifier containing the method name
- Identifiers in the TIMES and VARYING phrases of the PERFORM statement (windowed date fields *are* allowed in the PERFORM conditions)
- An identifier in the VARYING phrase of a serial (format-1) SEARCH statement, or any identifier in a binary (format-2) SEARCH statement (windowed date fields *are* allowed in the SEARCH conditions)
- An identifier in the ADVANCING phrase of the WRITE statement
- Arguments to intrinsic functions, except the UNDATE intrinsic function

Windowed date fields cannot be used as ascending or descending keys in MERGE or SORT statements.

Language elements that do not accept date fields as arguments

Neither windowed date fields nor expanded date fields can be used:

- In the DIVIDE statement, except as an identifier in the GIVING or REMAINDER clause
- In the MULTIPLY statement, except as an identifier in the GIVING clause

(Date fields cannot be used as operands in division or multiplication.)

EXTERNAL clause

The EXTERNAL clause specifies that the storage associated with a data item is associated with the run unit rather than with any particular program or method within the run unit. An external data item can be referenced by any program or method in the run unit that describes the data item. References to an external data item from different programs or methods using separate descriptions of the data item are always to the same data item. In a run unit, there is only one representative of an external data item.

The EXTERNAL clause can be specified only on data description entries whose level-number is 01. It can be specified only on data description entries that are in the working-storage section of a program or method. It cannot be specified in linkage section or file section data description entries. Any data item described by a data description entry subordinate to an entry that describes an external record also attains the external attribute. Indexes in an external data record do not possess the external attribute.

The data contained in the record named by the data-name clause is external and can be accessed and processed by any program or method in the run unit that describes and, optionally, redefines it. This data is subject to the following rules:

- If two or more programs or methods within a run unit describe the same external data record, each record-name of the associated record description entries must be the same, and the records must define the same number of bytes. However, a program or method that describes an external record can contain a data description entry including the REDEFINES clause that redefines the complete external record, and this complete redefinition need not occur identically in other programs or methods in the run unit.
- Use of the EXTERNAL clause does not imply that the associated data-name is a global name.

GLOBAL clause

The GLOBAL clause specifies that a data-name is available to every program contained within the program that declares it, as long as the contained program does not itself have a declaration for that name. All data-names subordinate to or condition-names or indexes associated with a global name are global names.

A data-name is global if the GLOBAL clause is specified either in the data description entry by which the data-name is declared or in another entry to which that data description entry is subordinate. The GLOBAL clause can be specified in the working-storage section, the file section, the linkage section, and the local-storage section, but only in data description entries whose level-number is 01.

In the same data division, the data description entries for any two data items for which the same data-name is specified must not include the GLOBAL clause.

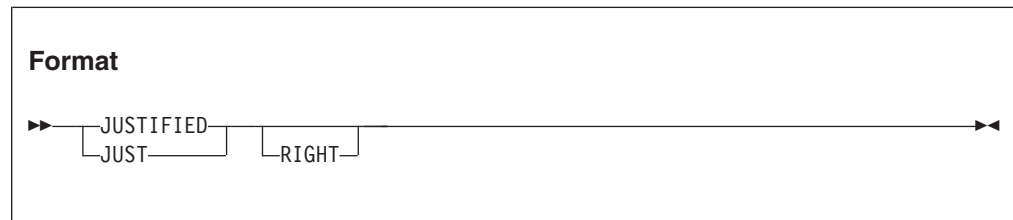
A statement in a program contained directly or indirectly within a program that describes a global name can reference that name without describing it again.

Two programs in a run unit can reference common data in the following circumstances:

- The data content of an external data record can be referenced from any program that describes the data record as external.
- If a program is contained within another program, both programs can refer to data that possesses the global attribute either in the containing program or in any program that directly or indirectly contains the containing program.

JUSTIFIED clause

The JUSTIFIED clause overrides standard positioning rules for receiving items of category alphabetic, alphanumeric, DBCS, or national.



You can specify the JUSTIFIED clause only at the elementary level. JUST is an abbreviation for JUSTIFIED, and has the same meaning.

You cannot specify the JUSTIFIED clause:

- For data items of category numeric, numeric-edited, alphanumeric-edited, or national-edited
- For edited DBCS items
- For index data items
- For items described as USAGE FUNCTION-POINTER, USAGE POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE
- For external floating-point or internal floating-point items
- For date fields
- With level-66 (RENAMES) and level-88 (condition-name) entries

When the JUSTIFIED clause is specified for a receiving item, the data is aligned at the rightmost character position in the receiving item. Also:

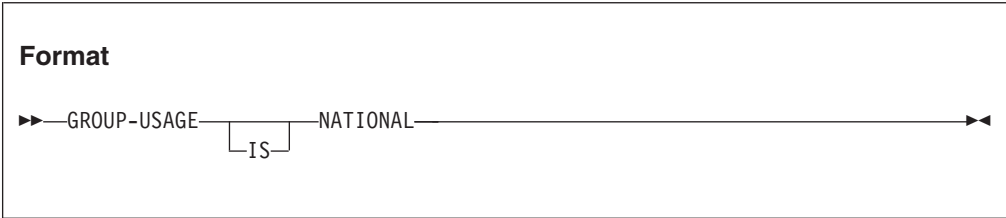
- If the sending item is larger than the receiving item, the leftmost character positions are truncated.
- If the sending item is smaller than the receiving item, the unused character positions at the left are filled with spaces. For a DBCS item, each unused position is filled with a DBCS space; for an item described with usage NATIONAL, each unused position is filled with the default Unicode space (NX'0020'); otherwise, each unused position is filled with an alphanumeric space.

If you omit the JUSTIFIED clause, the rules for standard alignment are followed (see “Alignment rules” on page 164).

The JUSTIFIED clause does not affect initial settings as determined by the VALUE clause.

GROUP-USAGE clause

A GROUP-USAGE clause with the NATIONAL phrase specifies that the group item defined by the entry is a national group item. A national group item contains national characters in all subordinate data items and subordinate group items.



When GROUP-USAGE NATIONAL is specified:

- The subject of the entry is a national group item. The class and category of a national group are national.
- A USAGE clause must not be specified for the subject of the entry. A USAGE NATIONAL clause is implied.
- A USAGE NATIONAL clause is implied for any subordinate elementary data items that are not described with a USAGE NATIONAL clause.
- All subordinate elementary data items must be explicitly or implicitly described with USAGE NATIONAL.
- Any signed numeric data items must be described with the SIGN IS SEPARATE clause.
- A GROUP-USAGE NATIONAL clause is implied for any subordinate group items that are not described with a GROUP-USAGE NATIONAL clause.
- All subordinate group items must be explicitly or implicitly described with a GROUP-USAGE NATIONAL clause.
- The JUSTIFIED clause must not be specified.

Unless stated otherwise, a national group item is processed as though it were an elementary data item of usage national, class and category national, described with PICTURE N(m), where m is the length of the group in national character positions.

Usage note: When you use national groups, the compiler can ensure proper truncation and padding of group items for statements such as MOVE and INSPECT. Groups defined without a GROUP-USAGE NATIONAL clause are alphanumeric groups. The content of alphanumeric groups, including any national characters, is treated as alphanumeric data, possibly leading to invalid truncation or mishandling of national character data.

The table below summarizes the cases where a national group item is processed as a group item.

Table 14. Where national group items are processed as groups

Language feature	Processing of national group items
Name qualification	The name of a national group item can be used to qualify the names of elementary data items and subordinate group items in the national group. The rules of qualification for a national group are the same as the rules of qualification for an alphanumeric group.

Table 14. Where national group items are processed as groups (continued)

Language feature	Processing of national group items
RENAMES clause	The rules for a national group item specified in the THROUGH phrase are the same as the rules for an alphanumeric group item specified in the THROUGH phrase. The result is an alphanumeric group item.
CORRESPONDING phrase	A national group item is processed as a group in accordance with the rules of the CORRESPONDING phrase. Elementary data items within a national group are processed the same as they would be if defined within an alphanumeric group.
INITIALIZE statement	A national group item is processed as a group in accordance with the rules of the INITIALIZE statement. Elementary items within the national group are initialized the same as they would be if defined within an alphanumeric group.
XML GENERATE statement	A national group item specified in the FROM phrase is processed as a group in accordance with the rules of the XML GENERATE statement. Elementary items within the national group are processed the same as they would be if defined within an alphanumeric group.

OCCURS clause

The data division clauses used for table handling are the OCCURS clause and the USAGE IS INDEX clause. For the USAGE IS INDEX description, see “USAGE clause” on page 229.

The OCCURS clause specifies tables whose elements can be referred to by indexing or subscripting. It also eliminates the need for separate entries for repeated data items.

Formats for the OCCURS clause include fixed-length tables and variable-length tables.

The *subject* of an OCCURS clause is the data-name of the data item that contains the OCCURS clause. Except for the OCCURS clause itself, data description clauses used with the subject apply to each occurrence of the item described.

Whenever the subject of an OCCURS clause or any data-item subordinate to it is referenced, it must be subscripted or indexed, with the following exceptions:

- When the subject of the OCCURS clause is used as the subject of a SEARCH statement
- When the subject or a subordinate data item is the object of the ASCENDING/DESCENDING KEY phrase
- When the subordinate data item is the object of the REDEFINES clause

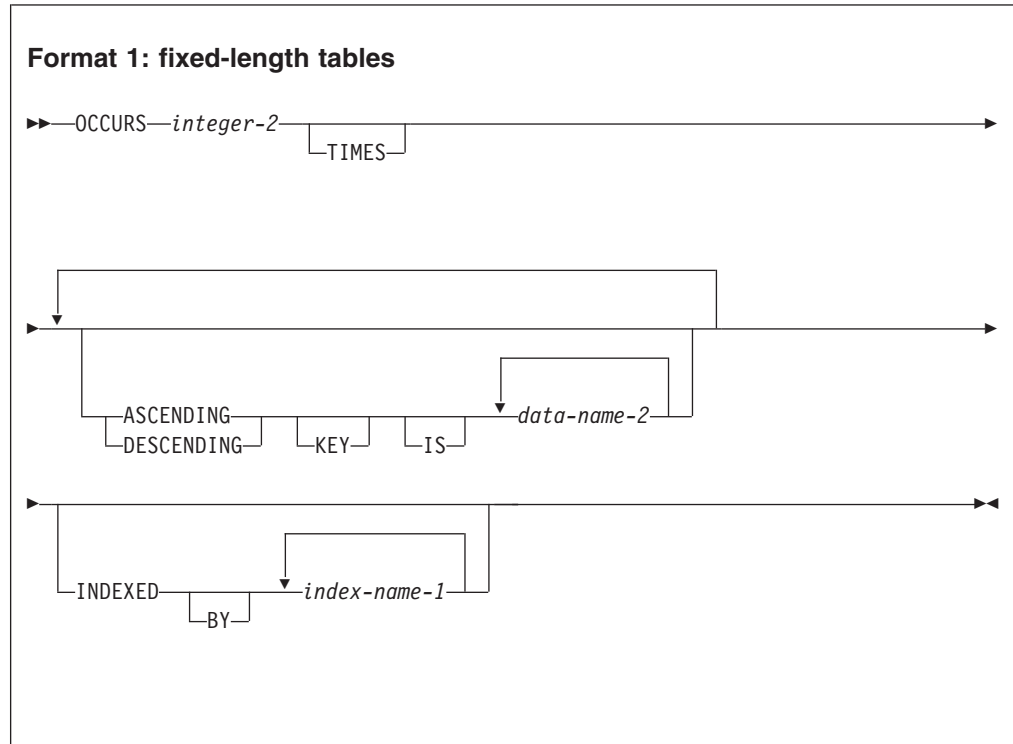
When subscripted or indexed, the subject refers to one occurrence within the table. When not subscripted or indexed, the subject references the entire table.

The OCCURS clause cannot be specified in a data description entry that:

- Has a level number of 01, 66, 77, or 88.
- Describes a redefined data item. (However, a redefined item can be subordinate to an item that contains an OCCURS clause.) See “REDEFINES clause” on page 216.

Fixed-length tables

Fixed-length tables are specified using the OCCURS clause. Because seven subscripts or indexes are allowed, six nested levels and one outermost level of the format-1 OCCURS clause are allowed. The format-1 OCCURS clause can be specified as subordinate to the OCCURS DEPENDING ON clause. In this way, a table of up to seven dimensions can be specified.



integer-2

The exact number of occurrences. *integer-2* must be greater than zero.

ASCENDING KEY and DESCENDING KEY phrases

Data is arranged in ascending or descending order, depending on the keyword specified, according to the values contained in *data-name-2*. The data-names are listed in their descending order of significance.

The order is determined by the rules for comparison of operands (see “Relation conditions” on page 266). The ASCENDING KEY and DESCENDING KEY data items are used in OCCURS clauses and the SEARCH ALL statement for a binary search of the table element.

data-name-2

Must be the name of the subject entry or the name of an entry subordinate to the subject entry. *data-name-2* cannot be a windowed date field. *data-name-2* can be qualified.

If *data-name-2* names the subject entry, that entire entry becomes the ASCENDING KEY or DESCENDING KEY and is the only key that can be specified for this table element.

If *data-name-2* does not name the subject entry, then *data-name-2*:

- Must be subordinate to the subject of the table entry itself
- Must *not* be subordinate to, or follow, any other entry that contains an OCCURS clause
- Must not contain an OCCURS clause

data-name-2 must not have subordinate items that contain OCCURS DEPENDING ON clauses.

When the ASCENDING KEY or DESCENDING KEY phrase is specified, the following rules apply:

- Keys must be listed in decreasing order of significance.
- The total number of keys for a given table element must not exceed 12.
- The data in the table must be arranged in ascending or descending sequence according to the collating sequence in use.
- The key must be described with one of the following usages:
 - BINARY
 - DISPLAY
 - DISPLAY-1
 - NATIONAL
 - PACKED-DECIMAL
 - COMPUTATIONAL
 - COMPUTATIONAL-1
 - COMPUTATIONAL-2
 - COMPUTATIONAL-3
 - COMPUTATIONAL-4
 - COMPUTATIONAL-5
- A key described with usage NATIONAL can have one of the following categories: national, national-edited, numeric-edited, numeric, or external floating-point.
- The sum of the lengths of all the keys associated with one table element must not exceed 256.
- If a key is specified without qualifiers and it is not a unique name, the key will be implicitly qualified with the subject of the OCCURS clause and all qualifiers of the OCCURS clause subject.

The following example illustrates the specification of ASCENDING KEY data items:

```
WORKING-STORAGE SECTION.
01 TABLE-RECORD.
   05 EMPLOYEE-TABLE OCCURS 100 TIMES
      ASCENDING KEY IS WAGE-RATE EMPLOYEE-NO
      INDEXED BY A, B.
      10 EMPLOYEE-NAME                PIC X(20).
      10 EMPLOYEE-NO                  PIC 9(6).
      10 WAGE-RATE                    PIC 9999V99.
      10 WEEK-RECORD OCCURS 52 TIMES
         ASCENDING KEY IS WEEK-NO INDEXED BY C.
         15 WEEK-NO                  PIC 99.
         15 AUTHORIZED-ABSENCES      PIC 9.
         15 UNAUTHORIZED-ABSENCES    PIC 9.
         15 LATE-ARRIVALS            PIC 9.
```

The keys for EMPLOYEE-TABLE are subordinate to that entry, and the key for WEEK-RECORD is subordinate to that subordinate entry.

In the preceding example, records in EMPLOYEE-TABLE must be arranged in ascending order of WAGE-RATE, and in ascending order of EMPLOYEE-NO within WAGE-RATE. Records in WEEK-RECORD must be arranged in ascending order of WEEK-NO. If they are not, results of any SEARCH ALL statement are unpredictable.

INDEXED BY phrase

The INDEXED BY phrase specifies the indexes that can be used with a table. A table without an INDEXED BY phrase can be referred to through indexing by using an index-name associated with another table. See “Subscripting using index-names (indexing)” on page 67.

Indexes normally are allocated in static memory associated with the program that contains the table. Thus indexes are in the last-used state when a program is reentered. However, in the following cases, indexes are allocated on a per-invocation basis. Thus you must set the value of the index on every entry for indexes on tables in the following sections:

- The local-storage section
- The working-storage section of a class definition (object instance variables)
- The linkage section of:
 - Methods
 - Programs compiled with the RECURSIVE clause
 - Programs compiled with the THREAD option

Indexes specified in an external data record do not possess the external attribute.

index-name-1

Each index-name specifies an index to be created by the compiler for use by the program. These index-names are *not* data-names and are not identified elsewhere in the COBOL program; instead, they can be regarded as private special registers for the use of this object program only. They are not data and are not part of any data hierarchy.

Unreferenced index names need not be uniquely defined.

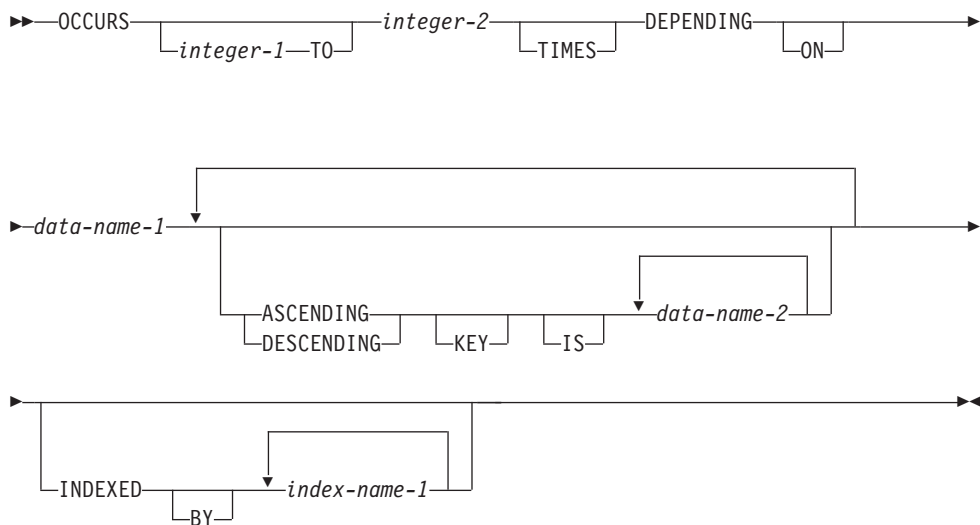
In one table entry, up to 12 index-names can be specified.

If a data item that possesses the global attribute includes a table accessed with an index, that index also possesses the global attribute. Therefore, the scope of an index-name is the same as that of the data-name that names the table in which the index is defined.

Variable-length tables

Variable-length tables are specified using the OCCURS DEPENDING ON clause.

Format 2: variable-length tables



integer-1

The minimum number of occurrences.

The value of *integer-1* must be greater than or equal to zero; it must also be less than the value of *integer-2*.

If *integer-1* is omitted, a value of 1 is assumed and the keyword TO must also be omitted.

integer-2

The maximum number of occurrences.

integer-2 must be greater than *integer-1*.

The *length* of the subject item is fixed. Only the *number of repetitions* of the subject item is variable.

OCCURS DEPENDING ON clause

The OCCURS DEPENDING ON clause specifies variable-length tables.

data-name-1

Identifies the *object* of the OCCURS DEPENDING ON clause; that is, the data item whose current value represents the current number of occurrences of the *subject* item. The contents of items whose occurrence numbers exceed the value of the object are undefined.

The object of the OCCURS DEPENDING ON clause (*data-name-1*) must describe an integer data item. The object cannot be a windowed date field.

The object of the OCCURS DEPENDING ON clause must not occupy any storage position within the range of the table (that is, any storage position from the first character position in the table through the last character position in the table).

The object of the OCCURS DEPENDING ON clause cannot be variably located; the object cannot follow an item that contains an OCCURS DEPENDING ON clause.

If the OCCURS clause is specified in a data description entry included in a record description entry that contains the EXTERNAL clause, *data-name-1*, if specified, must reference a data item that possesses the external attribute. *data-name-1* must be described in the same data division as the subject of the entry.

If the OCCURS clause is specified in a data description entry subordinate to one that contains the GLOBAL clause, *data-name-1*, if specified, must be a global name. *data-name-1* must be described in the same data division as the subject of the entry.

All data-names used in the OCCURS clause can be qualified; they cannot be subscripted or indexed.

At the time that the group item, or any data item that contains a subordinate OCCURS DEPENDING ON item or that follows but is not subordinate to the OCCURS DEPENDING ON item, is referenced, the value of the object of the OCCURS DEPENDING ON clause must fall within the range *integer-1* through *integer-2*.

When a group item that contains a subordinate OCCURS DEPENDING ON item is referred to, the part of the table area used in the operation is determined as follows:

- If the object is outside the group, only that part of the table area that is specified by the object at the start of the operation is used.
- If the object is included in the same group and the group data item is referenced as a sending item, only that part of the table area that is specified by the value of the object at the start of the operation is used in the operation.
- If the object is included in the same group and the group data item is referenced as a receiving item, the maximum length of the group item is used in the operation.

The following statements are affected by the maximum length rule:

- ACCEPT *identifier* (format 1 and 2)
- CALL ... USING BY REFERENCE *identifier*
- INVOKE ... USING BY REFERENCE *identifier*
- MOVE ... TO *identifier*
- READ ... INTO *identifier*
- RELEASE *identifier* FROM ...
- RETURN ... INTO *identifier*
- REWRITE *identifier* FROM ...
- STRING ... INTO *identifier*
- UNSTRING ... INTO *identifier* DELIMITER IN *identifier*
- WRITE *identifier* FROM ...

If a variable-length group item is not followed by a nonsubordinate item, the maximum length of the group is used when it appears as the identifier in CALL ...

USING BY REFERENCE *identifier*. Therefore, the object of the OCCURS DEPENDING ON clause does not need to be set unless the group is variably located.

If the group item is followed by a nonsubordinate item, the actual length, rather than the maximum length, is used. At the time the subject of entry is referenced, or any data item subordinate or superordinate to the subject of entry is referenced, the object of the OCCURS DEPENDING ON clause must fall within the range *integer-1* through *integer-2*.

Certain uses of the OCCURS DEPENDING ON clause result in *complex OCCURS DEPENDING ON* (ODO) items. The following constitute complex ODO items:

- A data item described with an OCCURS DEPENDING ON clause that is followed by a nonsubordinate elementary data item, described with or without an OCCURS clause
- A data item described with an OCCURS DEPENDING ON clause that is followed by a nonsubordinate group item
- A group item that contains one or more subordinate items described with an OCCURS DEPENDING ON clause
- A data item described with an OCCURS clause or an OCCURS DEPENDING ON clause that contains a subordinate data item described with an OCCURS DEPENDING ON clause (a table that contains variable-length elements)
- An index-name associated with a table that contains variable-length elements

The object of an OCCURS DEPENDING ON clause cannot be a nonsubordinate item that follows a complex ODO item.

Any nonsubordinate item that follows an item described with an OCCURS DEPENDING ON clause is a *variably located item*. That is, its location is affected by the value of the OCCURS DEPENDING ON object.

When implicit redefinition is used in a File Description (FD) entry, subordinate level items can contain OCCURS DEPENDING ON clauses.

The INDEXED BY phrase can be specified for a table that has a subordinate item that contains an OCCURS DEPENDING ON clause.

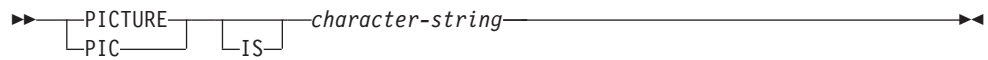
For more information about complex OCCURS DEPENDING ON, see *Appendix B. Complex OCCURS DEPENDING ON* in the *COBOL for AIX Programming Guide*.

The ASCENDING KEY phrase, the DESCENDING KEY phrase, and the INDEXED BY clause are described under “Fixed-length tables” on page 193.

PICTURE clause

The PICTURE clause specifies the general characteristics and editing requirements of an elementary item.

Format



PICTURE or PIC

The PICTURE clause must be specified for every elementary item except the following:

- Index data items
- The subject of the RENAMEs clause
- Items described with USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE
- Internal floating-point data items

In these cases, use of the PICTURE clause is prohibited.

The PICTURE clause can be specified only at the elementary level.

PIC is an abbreviation for PICTURE and has the same meaning.

character-string

character-string is made up of certain COBOL characters used as picture symbols. The allowable combinations determine the category of the elementary data item.

character-string can contain a maximum of 50 characters.

Symbols used in the PICTURE clause

Any punctuation character that appears within the PICTURE character-string is not considered a punctuation character, but rather is a PICTURE character-string symbol.

When specified in the SPECIAL-NAMES paragraph, DECIMAL-POINT IS COMMA exchanges the functions of the period and the comma in PICTURE character-strings and in numeric literals.

The lowercase letters that correspond to the uppercase letters that represent the following PICTURE symbols are equivalent to their uppercase representations in a PICTURE character-string:

A, B, E, G, N, P, S, V, X, Z, CR, DB

All other lowercase letters are not equivalent to their corresponding uppercase representations.

Table 15 on page 200 defines the meaning of each PICTURE clause symbol. The heading *Size* indicates how the item is counted in determining the number of character positions in the item. The type of the character positions depends on the USAGE clause specified for the item, as follows:

Usage	Type of character positions	Number of bytes per character
DISPLAY	Alphanumeric	1
DISPLAY-1	DBCS	2

Usage	Type of character positions	Number of bytes per character
NATIONAL	National	2
All others	Conceptual	Not applicable

Table 15. PICTURE clause symbol meanings

Symbol	Meaning	Size
A	A character position that can contain only a letter of the Latin alphabet or a space.	Each 'A' is counted as one character position in the size of the data item.
B	For usage DISPLAY, a character position into which an alphanumeric space is inserted. For usage DISPLAY-1, a character position into which a DBCS space is inserted. For usage NATIONAL, a character position into which a national space is inserted.	Each 'B' is counted as one character position in the size of the data item.
E	Marks the start of the exponent in an external floating-point item. For additional details of external floating-point items, see “Data categories and PICTURE rules” on page 204.	Each 'E' is counted as one character position in the size of the data item.
G	A DBCS character position.	Each 'G' is counted as one character position in the size of the data item.
N	A DBCS character position when specified with usage DISPLAY-1 or when usage is unspecified and the NSYMBOL(DBCS) compiler option is in effect. For category national, a national character position when specified with usage NATIONAL or when usage is unspecified and the NSYMBOL(NATIONAL) compiler option is in effect. For category national-edited, a national character position.	Each 'N' is counted as one character position in the size of the data item.
P	An assumed decimal scaling position. Used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. See “P symbol” on page 202 for further details.	Not counted in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions in numeric-edited items or in items that are used as arithmetic operands. The size of the value is the number of digit positions represented by the PICTURE character-string.
S	An indicator of the presence (but not the representation, and not necessarily the position) of an operational sign. An operational sign indicates whether the value of an item involved in an operation is positive or negative.	Not counted in the size of the elementary item, unless an associated SIGN clause specifies the SEPARATE CHARACTER phrase (which would be counted as one character position).
V	An indicator of the location of the assumed decimal point. Does not represent a character position. When the assumed decimal point is to the right of the rightmost symbol in the string, the V is redundant.	Not counted in the size of the elementary item.

Table 15. **PICTURE** clause symbol meanings (continued)

Symbol	Meaning	Size
X	A character position that can contain any allowable character from the alphanumeric character set of the computer.	Each 'X' is counted as one character position in the size of the data item.
Z	A leading numeric character position. When that position contains a zero, a space character replaces the zero.	Each 'Z' is counted as one character position in the size of the data item.
9	A character position that contains a numeral.	Each nine specifies one decimal digit in the value of the item. For usages DISPLAY and NATIONAL, each nine is counted as one character position in the size of the data item.
0	A character position into which the numeral zero is inserted.	Each zero is counted as one character position in the size of the data item.
/	A character position into which the slash character is inserted.	Each slash character is counted as one character position in the size of the data item.
,	A character position into which a comma is inserted.	Each comma is counted as one character position in the size of the data item.
.	An editing symbol that represents the decimal point for alignment purposes. In addition, it represents a character position into which a period is inserted.	Each period is counted as one character position in the size of the data item.
+ - CR DB	Editing sign control symbols. Each represents the character position into which the editing sign control symbol is placed.	Each character used in the editing sign symbol is counted as one character position in the size of the data item.
*	A check protect symbol: a leading numeric character position into which an asterisk is placed when that position contains a zero.	Each asterisk is counted as one character position in the size of the item.
cs	cs can be any valid currency symbol. A currency symbol represents a character position into which a currency sign value is placed. The default currency symbol is the character assigned the value X'24' in the code page in effect at compile time. In this document, the default currency symbol is represented by the dollar sign (\$) and cs stands for any valid currency symbol. For details, see "Currency symbol" on page 203.	The first occurrence of a currency symbol adds the number of characters in the currency sign value to the size of the data item. Each subsequent occurrence adds one character position to the size of the data item.

The following figure shows the sequences in which picture symbols can be specified to form picture character-strings. More detailed explanations of PICTURE clause symbols follow the figure.

FIRST SYMBOL SECOND SYMBOL	Non-Floating Insertion Symbols										Floating Insertion Symbols						Other Symbols								
	B	0	/	,	.	{ + }	{ - }	{ CR DB }	CS	E	{ Z }	{ * }	{ + }	{ - }	CS	CS	9	A X	S	V	P	P	G	N	
NON-FLOATING INSERTION SYMBOLS	B	•	•	•	•	•	•			•		•	•	•	•	•	•	•		•		•	•	•	•
	0	•	•	•	•	•	•			•		•	•	•	•	•	•	•		•		•	•	•	•
	/	•	•	•	•	•	•			•		•	•	•	•	•	•	•		•		•	•	•	•
	,	•	•	•	•	•	•			•		•	•	•	•	•	•	•		•		•	•	•	•
	.	•	•	•	•	•	•			•		•	•		•	•	•	•				•	•	•	•
	{ + }																								
	{ - }	•	•	•	•	•				•	•	•	•			•	•	•		•	•	•			
	{ CR DB }	•	•	•	•	•				•		•	•			•	•	•		•	•	•			
	CS						•																		
	E				•	•												•			•				
FLOATING INSERTION SYMBOLS	{ Z }	•	•	•	•	•				•		•													
	{ * }	•	•	•	•	•	•			•		•								•		•			
	{ + }	•	•	•	•	•				•				•											
	{ - }	•	•	•	•	•				•				•	•					•					
	CS	•	•	•	•	•	•									•									
	CS	•	•	•	•	•	•									•	•			•					
OTHER SYMBOLS	9	•	•	•	•	•	•			•		•		•			•	•	•	•		•			
	A X	•	•	•													•	•							
	S																								
	V	•	•	•	•	•	•			•		•		•			•		•		•				
	P	•	•	•	•	•	•			•		•		•			•		•		•				
	P					•				•									•	•		•			
	G	•																				•			
	N	•	•	•																				•	

Legend:

- Closed circle indicates that the symbol(s) at the top of the column can, in a given character-string, appear anywhere to the left of the symbol(s) at the left of the row.
- { } Braces indicate items that are mutually exclusive.

Symbols that appear twice

Nonfloating insertion symbols + and -, floating insertion symbols Z, *, +, -, and cs, and the symbol P appear twice. The leftmost column and uppermost row for each symbol represents its use to the left of the decimal point position. The second appearance of the symbol in the table represents its use to the right of the decimal point position.

P symbol

The symbol P specifies a scaling position and implies an assumed decimal point (to the left of the Ps if the Ps are leftmost PICTURE characters; to the right of the Ps if the Ps are rightmost PICTURE characters). The assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.

The symbol P can be specified only as a continuous string of Ps in the leftmost or rightmost digit positions within a PICTURE character-string.

In certain operations that reference a data item whose PICTURE character-string contains the symbol P, the algebraic value of the data item is used rather than the

actual character representation of the data item. This algebraic value assumes the decimal point in the prescribed location and zero in place of the digit position specified by the symbol P. The size of the value is the number of digit positions represented by the PICTURE character-string. These operations are any of the following:

- Any operation that requires a numeric sending operand
- A MOVE statement where the sending operand is numeric and its PICTURE character-string contains the symbol P
- A MOVE statement where the sending operand is numeric-edited and its PICTURE character-string contains the symbol P, and the receiving operand is numeric or numeric-edited
- A comparison operation where both operands are numeric

In all other operations, the digit positions specified with the symbol P are ignored and are not counted in the size of the operand.

Currency symbol

The currency symbol in a picture character-string is represented by the default currency symbol \$ or by a single character specified either in the CURRENCY compiler option or in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the environment division.

If the CURRENCY SIGN clause is specified, the CURRENCY and NOCURRENCY compiler options are ignored. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency sign value and currency symbol. For more information about the CURRENCY SIGN clause, see “CURRENCY SIGN clause” on page 117. For more information about the CURRENCY and NOCURRENCY compiler options, see *CURRENCY* in the *COBOL for AIX Programming Guide*.

A currency symbol can be repeated within the PICTURE character-string to specify floating insertion. Different currency symbols must not be used in the same PICTURE character-string.

Unlike all other picture symbols, currency symbols are case sensitive. For example, 'D' and 'd' specify different currency symbols.

A currency symbol can be used only to define a numeric-edited item with USAGE DISPLAY.

Character-string representation

Symbols that can appear more than once

The following symbols can appear more than once in one PICTURE character-string:

A B G N P X Z 9 0 / , + - * cs

At least one of the symbols A, G, N, X, Z, 9, or *, or at least two of the symbols +, -, or cs must be present in a PICTURE string.

An unsigned nonzero integer enclosed in parentheses immediately following any of these symbols specifies the number of consecutive occurrences of that symbol.

Example: The following two PICTURE clause specifications are equivalent:

```
PICTURE IS $99999.99CR
PICTURE IS $9(5).9(2)CR
```

Symbols that can appear only once

The following symbols can appear only once in one PICTURE character-string:

E S V . CR DB

Except for the PICTURE symbol V, each occurrence of any of the above symbols in a given PICTURE character-string represents an occurrence of that character or set of allowable characters in the data item.

Data categories and PICTURE rules

The allowable combinations of PICTURE symbols determine the data category of the item:

- Alphabetic
- Numeric
- Numeric-edited
- Alphanumeric
- Alphanumeric-edited
- DBCS
- External floating-point
- National
- National-edited

Note: Category internal floating point is defined by a USAGE clause that specifies the COMP-1 or COMP-2 phrase.

Alphabetic items

The PICTURE character-string can contain only the symbol A.

The content of the item must consist only of letters of the Latin alphabet and the space character.

Other clauses:

USAGE DISPLAY must be specified or implied.

Any associated VALUE clause must specify an alphanumeric literal containing only alphabetic characters, SPACE, or a symbolic-character as the value of a figurative constant.

The runtime locale in effect must indicate a code page that includes DBCS characters. For information about locales, see Appendix G, “Locale considerations,” on page 609.

Do not include a single byte character in a DBCS data item.

When padding is required for a DBCS data item, the following rules apply:

- Padding is done using double-byte space characters until the data area is filled (based on the number of double-byte character positions allocated for the data item).

- Padding is done using single-byte space characters when the padding needed is not an even number of bytes (for example, when an alphanumeric group item is moved to a DBCS data item).

Numeric items

Types of numeric items are:

- Binary
- Packed decimal (internal decimal)
- Zoned decimal (external decimal)
- National decimal (external decimal)

The type of a numeric item is defined by the usage clause as shown in the table below.

Table 16. Numeric types

Type	USAGE clause
Binary	BINARY, COMP, COMP-4, or COMP-5
Internal decimal	PACKED-DECIMAL, COMP-3
Zoned decimal (external decimal)	DISPLAY
National decimal (external decimal)	NATIONAL

For numeric date fields, the PICTURE character-string can contain only the symbols 9 and S. For all other numeric fields, the PICTURE character-string can contain only the symbols 9, P, S, and V.

The symbol S can be written only as the leftmost character in the PICTURE character-string.

The symbol V can be written only once in a given PICTURE character-string.

For binary items, the number of digit positions must range from 1 through 18 inclusive. For packed decimal and zoned decimal items the number of digit positions must range from 1 through 18, inclusive, when the ARITH(COMPAT) compiler option is in effect, or from 1 through 31, inclusive, when the ARITH(EXTEND) compiler option is in effect.

For numeric date fields, the number of digit positions must match the number of characters specified by the DATE FORMAT clause.

If unsigned, the contents of the item in standard data format must contain a combination of the Arabic numerals 0-9. If signed, it can also contain a +, -, or other representation of the operational sign.

Examples of valid ranges:

PICTURE	Valid range of values
9999	0 through 9999
S99	-99 through +99
S999V9	-999.9 through +999.9
PPP999	0 through .000999
S999PPP	-1000 through -999000 and +1000 through +999000 or zero

Other clauses:

The USAGE of the item can be DISPLAY, NATIONAL, BINARY, COMPUTATIONAL, PACKED-DECIMAL, COMPUTATIONAL-3, COMPUTATIONAL-4, or COMPUTATIONAL-5.

For signed numeric items described with usage NATIONAL, the SIGN IS SEPARATE clause must be specified or implied.

The TRUNC compiler option can affect the use of numeric data items. For details, see *TRUNC* in the *COBOL for AIX Programming Guide*.

Numeric-edited items

The PICTURE character-string can contain the following symbols:

B P V Z 9 0 / , . + - CR DB * cs

The combinations of symbols allowed are determined from the PICTURE clause symbol order allowed (see the figure in “Symbols used in the PICTURE clause” on page 199), and the editing rules (see “PICTURE clause editing” on page 211).

The following rules apply:

- Either the BLANK WHEN ZERO clause must be specified for the item, or the string must contain at least one of the following symbols:
B / Z 0 , . * + - CR DB cs
- Only one of the following symbols can be written in a given PICTURE character-string:
+ - CR DB
- If the ARITH(COMPAT) compiler option is in effect, then the number of digit positions represented in the character-string must be in the range 1 through 18, inclusive. If the ARITH(EXTEND) compiler option is in effect, then the number of digit positions represented in the character-string must be in the range 1 through 31, inclusive.
- The total number of character positions in the string (including editing-character positions) must not exceed 249.
- The contents of those character positions representing digits in standard data format must be one of the 10 Arabic numerals.

Other clauses:

USAGE DISPLAY or NATIONAL must be specified or implied.

If the usage of the item is DISPLAY, any associated VALUE clause must specify an alphanumeric literal or a figurative constant. The value is assigned without editing.

If the usage of the item is NATIONAL, any associated VALUE clause must specify an alphanumeric literal, a national literal, or a figurative constant. The value is assigned without editing.

Alphanumeric items

The PICTURE character-string must consist of either of the following:

- One or more occurrences of the symbol X.
- Combinations of the symbols A, X, and 9. (A character-string containing all As or all 9s does not define an alphanumeric item.)

The item is treated as if the character-string contained only the symbol X.

The contents of the item in standard data format can be any allowable characters from the character set of the computer.

Other clauses:

USAGE DISPLAY must be specified or implied.

Any associated VALUE clause must specify an alphanumeric literal or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- *symbolic-character*
- *ALL alphanumeric-literal*

Alphanumeric-edited items

The PICTURE character-string can contain the following symbols:

A X 9 B 0 /

The string must contain at least one A or X, and at least one B or 0 (zero) or /.

The contents of the item in standard data format must be two or more characters from the character set of the computer.

Other clauses:

USAGE DISPLAY must be specified or implied.

Any associated VALUE clause must specify an alphanumeric literal or or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- *symbolic-character*
- *ALL alphanumeric-literal*

The literal is treated exactly as specified; no editing is done.

DBCS items

The PICTURE character-string can contain the symbols G, G and B, or N. Each G, B, or N represents a single DBCS character position.

Any associated VALUE clause must contain a DBCS literal, the figurative constant SPACE, or the figurative constant *ALL DBCS-literal*.

Other clauses:

The runtime locale in effect must indicate a code page that includes DBCS characters. For information about locales, see Appendix G, “Locale considerations,” on page 609.

Do not include a single byte character in a DBCS data item.

When padding is required for a DBCS data item, the following rules apply:

- Padding is done using double-byte space characters until the data area is filled (based on the number of double-byte character positions allocated for the data item).
- Padding is done using single-byte space characters when the padding needed is not an even number of bytes (for example, when an alphanumeric group item is moved to a DBCS data item).

When PICTURE symbol G is used, USAGE DISPLAY-1 must be specified. When PICTURE symbol N is used and the NSYMBOL(DBCS) compiler option is in effect, USAGE DISPLAY-1 is implied if the USAGE clause is omitted.

National items

The PICTURE character-string can contain one or more occurrences of the picture symbol N.

These rules apply when the NSYMBOL(NATIONAL) compiler option is in effect or the USAGE NATIONAL clause is specified. In the absence of a USAGE NATIONAL clause, if the NSYMBOL(DBCS) compiler option is in effect, picture symbol N represents a DBCS character and the rules of the PICTURE clause for a DBCS item apply.

Each N represents a single national character position.

Any associated VALUE clause must specify an alphanumeric literal, a national literal, or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- *symbolic-character*
- *ALL alphanumeric-literal*
- *ALL national-literal*

Other clauses:

Only the NATIONAL phrase can be specified in the USAGE clause. When PICTURE symbol N is used and the NSYMBOL(NATIONAL) compiler option is in effect, USAGE NATIONAL is implied if the usage clause is omitted.

The following clauses can be used:

- JUSTIFIED
- EXTERNAL

- GLOBAL
- OCCURS
- REDEFINES
- RENAMES
- SYNCHRONIZED

The following clauses cannot be used:

- BLANK WHEN ZERO
- SIGN
- DATE FORMAT

National-edited items

The PICTURE character-string must contain the following:

- At least one symbol N, and
- At least one of the symbols B 0 (zero) or / (slash)

Each symbol represents a single national character position.

Any associated VALUE clause must specify an alphanumeric literal, a national literal, or one of the following figurative constants:

- ZERO
- SPACE
- QUOTE
- HIGH-VALUE
- LOW-VALUE
- *symbolic-character*
- *ALL alphanumeric-literal*
- *ALL national-literal*

The literal is treated exactly as specified; no editing is done.

The NSYMBOL(NATIONAL) compiler option has no effect on the definition of a data item of category national-edited.

Other clauses:

USAGE NATIONAL must be specified or implied.

The following clauses can be used:

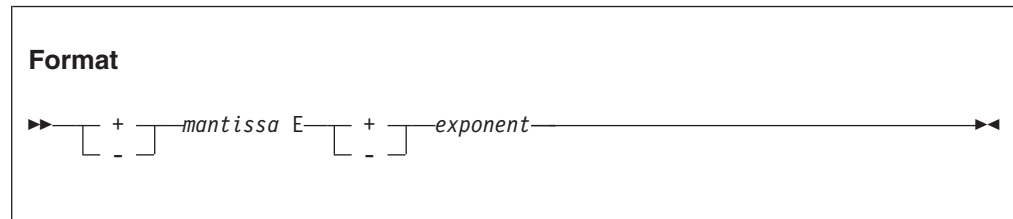
- JUSTIFIED
- EXTERNAL
- GLOBAL
- OCCURS
- REDEFINES
- RENAMES
- SYNCHRONIZED

The following clauses cannot be used:

- BLANK WHEN ZERO

- SIGN
- DATE FORMAT

External floating-point items



+ or - A sign character must immediately precede both the mantissa and the exponent.

A + sign indicates that a positive sign will be used in the output to represent positive values and that a negative sign will represent negative values.

A - sign indicates that a blank will be used in the output to represent positive values and that a negative sign will represent negative values.

Each sign position occupies one byte of storage.

mantissa

The mantissa can contain the symbols:

9 . V

An actual decimal point can be represented with a period (.) while an assumed decimal point is represented by a V.

Either an actual or an assumed decimal point must be present in the mantissa; the decimal point can be leading, embedded, or trailing.

The mantissa can contain from 1 to 16 numeric characters.

E Indicates the exponent.

exponent

The exponent must consist of the symbol 99.

Example: Pic -9v9(9)E-99

The DISPLAY phrase of the USAGE clause and a floating-point picture character-string define the item as a *display floating-point data item*.

The NATIONAL phrase of the USAGE clause and a floating-point picture character-string define the item as a *national floating-point data item*.

For items defined with usage DISPLAY, each picture symbol except V defines one alphanumeric character position in the item.

For items defined with usage NATIONAL, each picture symbol except V defines one national character position in the item.

Other clauses:

The DISPLAY phrase or the NATIONAL phrase of the USAGE clause must be specified or implied.

The OCCURS, REDEFINES, and RENAMES clauses can be associated with external floating-point items.

The SIGN clause is accepted as documentation and has no effect on the representation of the sign.

The SYNCHRONIZED clause is treated as documentation.

The following clauses are invalid with external floating-point items:

- BLANK WHEN ZERO
- JUSTIFIED
- VALUE

PICTURE clause editing

There are two general methods of editing in a PICTURE clause:

- Insertion editing:
 - Simple insertion
 - Special insertion
 - Fixed insertion
 - Floating insertion
- Suppression and replacement editing:
 - Zero suppression and replacement with asterisks
 - Zero suppression and replacement with spaces

The type of editing allowed for an item depends on its *data category*. The type of editing that is valid for each category is shown in the following table. *cs* indicates any valid currency symbol.

Table 17. Data categories

Data category	Type of editing	Insertion symbol
Alphabetic	None	None
Numeric	None	None
Numeric-edited	Simple insertion	B 0 / ,
	Special insertion	.
	Fixed insertion	<i>cs</i> + - CR DB
	Floating insertion	<i>cs</i> + -
	Zero suppression	Z *
	Replacement	Z * + - <i>cs</i>
Alphanumeric	None	None
Alphanumeric-edited	Simple insertion	B 0 /
DBCS	Simple insertion	B
External floating-point	Special insertion	.
National	None	None

Table 17. Data categories (continued)

Data category	Type of editing	Insertion symbol
National-edited	Simple insertion	B 0 /

Types of editing are described in the following sections:

- “Simple insertion editing”
- “Special insertion editing”
- “Fixed insertion editing” on page 213
- “Floating insertion editing” on page 214
- “Zero suppression and replacement editing” on page 215

Simple insertion editing

This type of editing is valid for alphanumeric-edited, numeric-edited, and DBCS items.

Each insertion symbol is counted in the size of the item, and represents the position within the item where the equivalent character is to be inserted. For edited DBCS items, each insertion symbol (B) is counted in the size of the item and represents the position within the item where the DBCS space is to be inserted.

For example:

PICTURE	Value of data	Edited result
X(10)/XX	ALPHANUMER01	ALPHANUMER/01
X(5)BX(7)	ALPHANUMERIC	ALPHA NUMERIC
99,B999,B000	1234	01, <i>b</i> 234, <i>b</i> 000 ¹
99,999	12345	12, <i>b</i> 345
GGBBGG	D1D2D3D4	D1D2 <i>bbbb</i> D3D4 ¹
Notes: 1. The symbol <i>b</i> represents a space.		

Special insertion editing

This type of editing is valid for either numeric-edited items or external floating-point items.

The period (.) is the special insertion symbol; it also represents the actual decimal point for alignment purposes.

The period insertion symbol is counted in the size of the item, and represents the position within the item where the actual decimal point is inserted.

Either the actual decimal point or the symbol V as the assumed decimal point, but not both, must be specified in one PICTURE character-string.

For example:

PICTURE	Value of data	Edited result
999.99	1.234	001.23
999.99	12.34	012.34
999.99	123.45	123.45
999.99	1234.5	234.50
+999.99E+99	12345	+123.45E+02

Fixed insertion editing

This type of editing is valid only for numeric-edited items. The following insertion symbols are used:

- *cs*
- + - CR DB (editing-sign control symbols)

In fixed insertion editing, only one currency symbol and one editing-sign control symbol can be specified in a PICTURE character-string.

Unless it is preceded by a + or - symbol, the currency symbol must be the first character in the character-string.

When either + or - is used as a symbol, it must be the first or last character in the character-string.

When CR or DB is used as a symbol, it must occupy the rightmost two character positions in the character-string. If these two character positions contain the symbols CR or DB, the uppercase letters are the insertion characters.

Editing sign control symbols produce results that depend on the value of the data item, as shown below:

Editing symbol in PICTURE character-string	Result: data item positive or zero	Result: data item negative
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

For example:

PICTURE	Value of data	Edited result
999.99+	+6555.556	555.55+
+9999.99	-6555.555	-6555.55
9999.99	+1234.56	1234.56
\$999.99	-123.45	\$123.45
-\$999.99	-123.456	-\$123.45
-\$999.99	+123.456	\$123.45
\$9999.99CR	+123.45	\$0123.45
\$9999.99CR	-123.45	\$0123.45CR

Floating insertion editing

This type of editing is valid only for numeric-edited items.

The following symbols are used:

cs + -

Within one PICTURE character-string, these symbols are mutually exclusive as floating insertion characters.

Floating insertion editing is specified by using a string of at least two of the allowable floating insertion symbols to represent leftmost character positions into which the actual characters can be inserted.

The leftmost floating insertion symbol in the character-string represents the leftmost limit at which the actual character can appear in the data item. The rightmost floating insertion symbol represents the rightmost limit at which the actual character can appear.

The second leftmost floating insertion symbol in the character-string represents the leftmost limit at which numeric data can appear within the data item. Nonzero numeric data can replace all characters at or to the right of this limit.

Any simple-insertion symbols (B 0 / ,) within or to the immediate right of the string of floating insertion symbols are considered part of the floating character-string. If the period (.) special-insertion symbol is included within the floating string, it is considered to be part of the character-string.

To avoid truncation, the minimum size of the PICTURE character-string must be:

- The number of character positions in the sending item, plus
- The number of nonfloating insertion symbols in the receiving item, plus
- One character position for the floating insertion symbol

Representing floating insertion editing

In a PICTURE character-string, there are two ways to represent floating insertion editing and thus two ways in which editing is performed:

1. Any or all leading numeric character positions to the left of the decimal point are represented by the floating insertion symbol. When editing is performed, a single floating insertion character is placed to the immediate left of the first nonzero digit in the data, or of the decimal point, whichever is farther to the left. The character positions to the left of the inserted character are filled with spaces.

If all numeric character positions in the PICTURE character-string are represented by the insertion character, then at least one of the insertion characters must be to the left of the decimal point.

2. All the numeric character positions are represented by the floating insertion symbol. When editing is performed, then:
 - If the value of the data is zero, the entire data item will contain spaces.
 - If the value of the data is nonzero, the result is the same as in rule 1.

For example:

PICTURE	Value of data	Edited result
\$\$\$\$.99	.123	\$.12
\$\$\$9.99	.12	\$0.12
,\$\$\$,999.99	-1234.56	\$1,234.56
+,+++,999.99	-123456.789	-123,456.78
\$\$,\$\$\$,\$\$\$\$.99CR	-1234567	\$1,234,567.00CR
++,+++,+++.+++	0000.00	

Zero suppression and replacement editing

This type of editing is valid only for numeric-edited items.

In zero suppression editing, the symbols Z and * are used. These symbols are mutually exclusive in one PICTURE character-string.

The following symbols are mutually exclusive as floating replacement symbols in one PICTURE character-string:

*Z * + - cs*

Specify zero suppression and replacement editing with a string of one or more of the allowable symbols to represent leftmost character positions in which zero suppression and replacement editing can be performed.

Any simple insertion symbols (B 0 / ,) within or to the immediate right of the string of floating editing symbols are considered part of the string. If the period (.) special insertion symbol is included within the floating editing string, it is considered to be part of the character-string.

Representing zero suppression

In a PICTURE character-string, there are two ways to represent zero suppression, and two ways in which editing is performed:

- Any or all of the leading numeric character positions to the left of the decimal point are represented by suppression symbols. When editing is performed, the replacement character replaces any leading zero in the data that appears in the same character position as a suppression symbol. Suppression stops at the leftmost character:
 - That does not correspond to a suppression symbol
 - That contains nonzero data
 - That is the decimal point
- All the numeric character positions in the PICTURE character-string are represented by the suppression symbols. When editing is performed and the value of the data is nonzero, the result is the same as in the preceding rule. If the value of the data is zero, then:
 - If Z has been specified, the entire data item will contain spaces.
 - If * has been specified, the entire data item except the actual decimal point will contain asterisks.

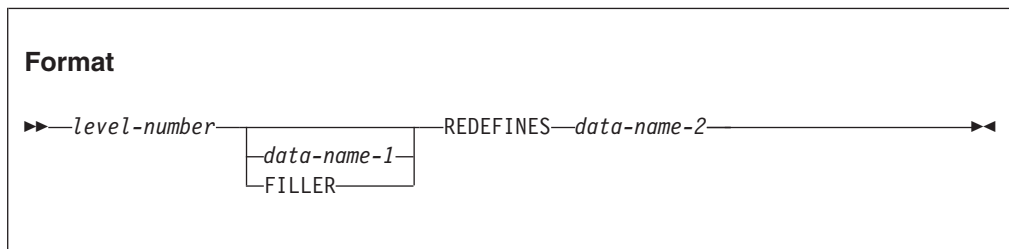
For example:

PICTURE	Value of data	Edited result
****. **	0000.00	****. **
ZZZZ.ZZ	0000.00	
ZZZZ.99	0000.00	.00
****.99	0000.00	****.00
ZZ99.99	0000.00	00.00
Z,ZZZ.ZZ+	+123.456	123.45+
*,***.***	-123.45	**123.45-
,,***,***.***	+12345678.9	12,345,678.90+
\$Z,ZZZ,ZZZ.ZZCR	+12345.67	\$ 12,345.67
\$B*,***,***.**BBDB	-12345.67	\$ ***12,345.67 DB

Do not specify both the asterisk (*) as a suppression symbol and the BLANK WHEN ZERO clause for the same entry.

REDEFINES clause

The REDEFINES clause allows you to use different data description entries to describe the same computer storage area.



(*level-number*, *data-name-1*, and FILLER are not part of the REDEFINES clause, and are included in the format only for clarity.)

When specified, the REDEFINES clause must be the first entry following *data-name-1* or FILLER. If *data-name-1* or FILLER is not specified, the REDEFINES clause must be the first entry following the *level-number*.

data-name-1, FILLER

Identifies an alternate description for the data area identified by *data-name-2*; *data-name-1* is the *redefining* item or the REDEFINES *subject*.

Neither *data-name-1* nor any of its subordinate entries can contain a VALUE clause.

data-name-2

Identifies the *redefined* item or the REDEFINES *object*.

The data description entry for *data-name-2* can contain a REDEFINES clause.

The data description entry for *data-name-2* cannot contain an OCCURS clause. However, *data-name-2* can be subordinate to an item whose data

description entry contains an OCCURS clause; in this case, the reference to *data-name-2* in the REDEFINES clause must not be subscripted.

Neither *data-name-1* nor *data-name-2* can contain an OCCURS DEPENDING ON clause.

data-name-1 and *data-name-2* must have the same level in the hierarchy; however, the level numbers need not be the same. Neither *data-name-1* nor *data-name-2* can be defined with level number 66 or 88.

data-name-1 and *data-name-2* can each be described with any usage.

Redefinition begins at *data-name-1* and ends when a level-number less than or equal to that of *data-name-1* is encountered. No entry that has a level-number numerically lower than those of *data-name-1* and *data-name-2* can occur between these entries. In the following example:

```
05  A PICTURE X(6).  
05  B REDEFINES A.  
    10 B-1          PICTURE X(2).  
    10 B-2          PICTURE 9(4).  
05  C              PICTURE 99V99.
```

A is the redefined item, and B is the redefining item. Redefinition begins with B and includes the two subordinate items B-1 and B-2. Redefinition ends when the level-05 item C is encountered.

If the GLOBAL clause is used in the data description entry that contains the REDEFINES clause, only *data-name-1* (the redefining item) possesses the global attribute. For example, in the following description, only item B possesses the GLOBAL attribute:

```
05  A PICTURE X(6).  
05  B REDEFINES A GLOBAL PICTURE X(4).
```

The EXTERNAL clause must not be specified in the same data description entry as a REDEFINES clause.

If the redefined data item (*data-name-2*) is declared to be an external data record, the size of the redefining data item (*data-name-1*) must not be greater than the size of the redefined data item. If the redefined data item is not declared to be an external data record, there is no such constraint.

The following example shows that the redefining item, B, can occupy more storage than the redefined item, A. The size of storage for the REDEFINED clause is determined in number of bytes. Item A occupies 6 bytes of storage and item B, a data item of category national, occupies 8 bytes of storage.

```
05  A PICTURE X(6).  
05  B REDEFINES A GLOBAL PICTURE N(4).
```

One or more redefinitions of the same storage area are permitted. The entries that give the new descriptions of the storage area must immediately follow the description of the redefined area without intervening entries that define new character positions. Multiple redefinitions can, but need not, all use the data-name of the original entry that defined this storage area. For example:

```
05  A          PICTURE 9999.  
05  B REDEFINES A PICTURE 9V999.  
05  C REDEFINES A PICTURE 99V99.
```

Also, multiple redefinitions can use the name of the preceding definition as shown in the following example:

```
05  A                PICTURE 9999.
05  B REDEFINES A    PICTURE 9V999.
05  C REDEFINES B    PICTURE 99V99.
```

When more than one level-01 entry is written subordinate to an FD entry, a condition known as *implicit redefinition* occurs. That is, the second level-01 entry implicitly redefines the storage allotted for the first entry. In such level-01 entries, the REDEFINES clause must not be specified.

When the data item implicitly redefines multiple 01-level records in a file description (FD) entry, items subordinate to the redefining or redefined item can contain an OCCURS DEPENDING ON clause.

REDEFINES clause considerations

When an area is redefined, all descriptions of the area are always in effect; that is, redefinition does not supersede a previous description. Thus, if B REDEFINES C has been specified, either of the two procedural statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, the area described as B would receive the value and format of X. In the second case, the same physical area (described now as C) would receive the value and format of Y. Note that if the second statement is executed immediately after the first, the value of Y replaces the value of X in the one storage area.

The usage of a redefining data item need not be the same as that of a redefined item. This does not, however, cause any change in the format or content of existing data. For example:

```
05  B                PICTURE 99 USAGE DISPLAY VALUE 8.
05  C REDEFINES B    PICTURE S99 USAGE COMPUTATIONAL-4.
05  A                PICTURE S99 USAGE COMPUTATIONAL-4.
```

Redefining B does not change the bit configuration of the data in the storage area. Therefore, the following two statements produce different results:

```
ADD B TO A
ADD C TO A
```

In the first case, the value 8 is added to A (because B has USAGE DISPLAY). In the second statement, the value -3848 is added to A (because C has USAGE COMPUTATIONAL-4), and the bit configuration of the storage area has the binary value -3848. This example demonstrates how the improper use of redefinition can give unexpected or incorrect results.

REDEFINES clause examples

The REDEFINES clause can be specified for an item within the scope of (subordinate to) an area that is redefined. In the following example, WEEKLY-PAY redefines SEMI-MONTHLY-PAY (which is within the scope of REGULAR-EMPLOYEE, while REGULAR-EMPLOYEE is redefined by TEMPORARY-EMPLOYEE).

```
05  REGULAR-EMPLOYEE.
    10  LOCATION                PICTURE A(8).
    10  GRADE                   PICTURE X(4).
    10  SEMI-MONTHLY-PAY        PICTURE 9999V99.
    10  WEEKLY-PAY REDEFINES SEMI-MONTHLY-PAY
                                   PICTURE 999V999.
```

```

05  TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
    10  LOCATION                PICTURE A(8).
    10  FILLER                  PICTURE X(6).
    10  HOURLY-PAY              PICTURE 99V99.

```

The REDEFINES clause can also be specified for an item subordinate to a redefining item, as shown for CODE-H REDEFINES HOURLY-PAY in the following example:

```

05  REGULAR-EMPLOYEE.
    10  LOCATION                PICTURE A(8).
    10  GRADE                   PICTURE X(4).
    10  SEMI-MONTHLY-PAY       PICTURE 999V999.
05  TEMPORARY-EMPLOYEE REDEFINES REGULAR-EMPLOYEE.
    10  LOCATION                PICTURE A(8).
    10  FILLER                  PICTURE X(6).
    10  HOURLY-PAY              PICTURE 99V99.
    10  CODE-H REDEFINES HOURLY-PAY PICTURE 9999.

```

Data items within an area can be redefined without changing their lengths. For example:

```

05  NAME-2.
    10  SALARY                  PICTURE XXX.
    10  SO-SEC-NO              PICTURE X(9).
    10  MONTH                   PICTURE XX.
05  NAME-1 REDEFINES NAME-2.
    10  WAGE                    PICTURE XXX.
    10  EMP-NO                  PICTURE X(9).
    10  YEAR                    PICTURE XX.

```

Data item lengths and types can also be respecified within an area. For example:

```

05  NAME-2.
    10  SALARY                  PICTURE XXX.
    10  SO-SEC-NO              PICTURE X(9).
    10  MONTH                   PICTURE XX.
05  NAME-1 REDEFINES NAME-2.
    10  WAGE                    PICTURE 999V999.
    10  EMP-NO                  PICTURE X(6).
    10  YEAR                    PICTURE XX.

```

Data items can also be respecified with a length that is greater than the length of the redefined item. For example:

```

05  NAME-2.
    10  SALARY                  PICTURE XXX.
    10  SO-SEC-NO              PICTURE X(9).
    10  MONTH                   PICTURE XX.
05  NAME-1 REDEFINES NAME-2.
    10  WAGE                    PICTURE 999V999.
    10  EMP-NO                  PICTURE X(6).
    10  YEAR                    PICTURE X(4).

```

This does not change the length of the redefined item NAME-2.

Undefined results

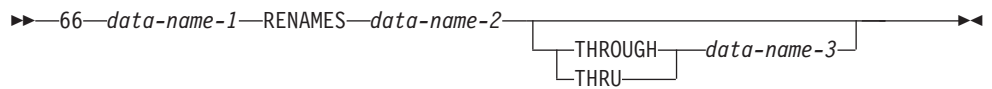
Undefined results can occur when:

- A redefining item is moved to a redefined item (that is, if B REDEFINES C and the statement MOVE B TO C is executed).
- A redefined item is moved to a redefining item (that is, if B REDEFINES C and the statement MOVE C TO B is executed).

RENAMES clause

The RENAMES clause specifies alternative and possibly overlapping groupings of elementary data items.

Format



The special level-number 66 must be specified for data description entries that contain the RENAMES clause. (Level-number 66 and *data-name-1* are not part of the RENAMES clause, and are included in the format only for clarity.)

One or more RENAMES entries can be written for a logical record. All RENAMES entries associated with one logical record must immediately follow the last data description entry of that record.

data-name-1

Identifies an alternative grouping of data items.

A level-66 entry cannot rename a level-01, level-77, level-88, or another level-66 entry.

data-name-1 cannot be used as a qualifier; it can be qualified only by the names of level indicator entries or level-01 entries.

data-name-2, data-name-3

Identify the original grouping of elementary data items; that is, they must name elementary or group items within the associated level-01 entry and must not be the same data-name. Both data-names can be qualified.

data-name-2 and *data-name-3* can each reference any of the following items:

- An elementary data item
- An alphanumeric group item
- A national group item

When *data-name-2* or *data-name-3* references a national group item, the referenced item is processed as a group (not as an elementary data item of category national).

The OCCURS clause must not be specified in the data entries for *data-name-2* and *data-name-3*, or for any group entry to which they are subordinate. In addition, the OCCURS DEPENDING clause must not be specified for any item defined between *data-name-2* and *data-name-3*.

The keywords THROUGH and THRU are equivalent.

When the THROUGH phrase is specified:

- *data-name-1* defines an alphanumeric group item that includes all the elementary items that:

- Start with *data-name-2* if it is an elementary item, or the first elementary item within *data-name-2* if it is a group item
- End with *data-name-3* if it is an elementary item, or the last elementary item within *data-name-3* if it is an alphanumeric group item or national group item
- The storage area occupied by the starting item through the ending item becomes the storage area occupied by *data-name-1*.

Usage note: The group defined with the THROUGH phrase can include data items of usage NATIONAL.

The leftmost character position in *data-name-3* must not precede the leftmost character position in *data-name-2*, and the rightmost character position in *data-name-3* must not precede the rightmost character position in *data-name-2*. This means that *data-name-3* cannot be totally subordinate to *data-name-2*.

When the THROUGH phrase is not specified:

- The storage area occupied by *data-name-2* becomes the storage area occupied by *data-name-1*.
- All of the data attributes of *data-name-2* become the data attributes for *data-name-1*. That is:
 - When *data-name-2* is an alphanumeric group item, *data-name-1* is an alphanumeric group item.
 - When *data-name-2* is a national group item, *data-name-1* is a national group item.
 - When *data-name-2* is an elementary item, *data-name-1* is an elementary item.

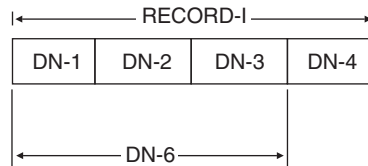
The following figure illustrates valid and invalid RENAMEs clause specifications.

COBOL Specifications

Storage Layouts

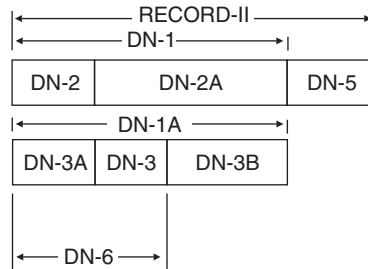
Example 1 (Valid)

```
01 RECORD-I.
   05 DN-1... .
   05 DN-2... .
   05 DN-3... .
   05 DN-4... .
66 DN-6 RENAMES DN-1 THROUGH DN-3.
```



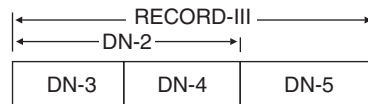
Example 2 (Valid)

```
01 RECORD-II.
   05 DN-1.
      10 DN-2... .
      10 DN-2A... .
   05 DN-1A REDEFINES DN-1.
      10 DN-3A... .
      10 DN-3... .
      10 DN-3B... .
   05 DN-5... .
66 DN-6 RENAMES DN-2 THROUGH DN-3.
```



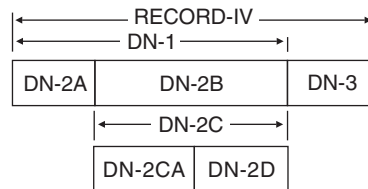
Example 3 (Invalid)

```
01 RECORD-III.
   05 DN-2.
      10 DN-3... .
      10 DN-4... .
   05 DN-5... .
66 DN-6 RENAMES DN-2 THROUGH DN-3. DN-6 is indeterminate
```



Example 4 (Invalid)

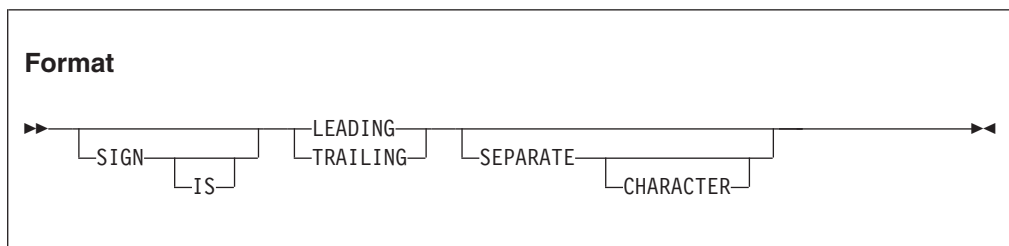
```
01 RECORD-IV.
   05 DN-1.
      10 DN-2A... .
      10 DN-2B... .
      10 DN-2C REDEFINES DN-2B.
         15 DN-2CA... .
      15 DN-2D... .
      05 DN-3... .
66 DN-4 RENAMES DN-1 THROUGH DN-2CA. DN-4 is indeterminate
```



SIGN clause

The SIGN clause specifies the position and mode of representation of the operational sign for the signed numeric item to which it applies.

The SIGN clause is required only when an explicit description of the properties or position of the operational sign is necessary.



The SIGN clause can be specified only for the following items:

- An elementary numeric data item of usage DISPLAY or NATIONAL that is described with an S in its picture character string, or
- A group item that contains at least one such elementary entry as a subordinate item

When the SIGN clause is specified at the group level, that SIGN clause applies only to subordinate signed numeric elementary data items of usage DISPLAY or NATIONAL. Such a group can also contain items that are not affected by the SIGN clause. If the SIGN clause is specified for a group or elementary entry that is subordinate to a group item that has a SIGN clause, the SIGN clause for the subordinate entry takes precedence for that subordinate entry.

The SIGN clause is treated as documentation for external floating-point items.

When the SIGN clause is specified without the SEPARATE phrase, USAGE DISPLAY must be specified explicitly or implicitly. When SIGN IS SEPARATE is specified, either USAGE DISPLAY or USAGE NATIONAL can be specified.

If you specify the CODE-SET clause in an FD entry, any signed numeric data description entries associated with that file description entry must be described with the SIGN IS SEPARATE clause.

If the SEPARATE CHARACTER phrase is not specified, then:

- The operational sign is presumed to be associated with the LEADING or TRAILING digit position, whichever is specified, of the elementary numeric data item. (In this instance, specification of SIGN IS TRAILING is the equivalent of the standard action of the compiler.)
- The character S in the PICTURE character string is not counted in determining the size of the item (in terms of standard data format characters).

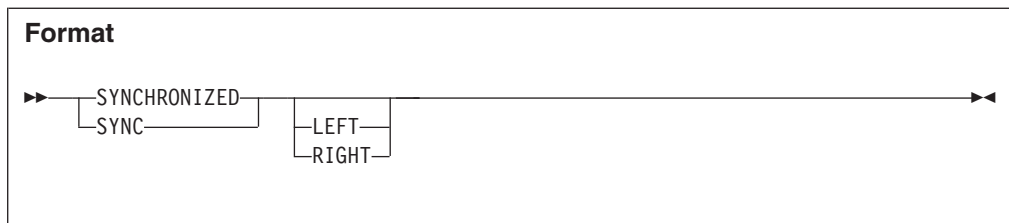
If the SEPARATE CHARACTER phrase is specified, then:

- The operational sign is presumed to be the LEADING or TRAILING character position, whichever is specified, of the elementary numeric data item. This character position is not a digit position.
- The character S in the PICTURE character string is counted in determining the size of the data item (in terms of standard data format characters).
- + is the character used for the positive operational sign.
- - is the character used for the negative operational sign.

The SEPARATE CHARACTER phrase cannot be specified for a date field.

SYNCHRONIZED clause

The SYNCHRONIZED clause specifies the alignment of an elementary item on a natural boundary in storage.



SYNC is an abbreviation for SYNCHRONIZED and has the same meaning.

The SYNCHRONIZED clause is never required, but can improve performance on some systems for binary items used in arithmetic.

The SYNCHRONIZED clause can be specified for elementary items and for level-01 group items, in which case every elementary item within the group item is synchronized.

LEFT Specifies that the elementary item is to be positioned so that it will begin at the left character position of the natural boundary in which the elementary item is placed.

RIGHT Specifies that the elementary item is to be positioned such that it will terminate on the right character position of the natural boundary in which it has been placed.

When specified, the LEFT and the RIGHT phrases are syntax checked but have no effect on the execution of the program.

The length of an elementary item is not affected by the SYNCHRONIZED clause.

The following table lists the effect of the SYNCHRONIZE clause on other language elements.

Table 18. SYNCHRONIZE clause effect on other language elements

Language element	Comments
OCCURS clause	When specified for an item within the scope of an OCCURS clause, each occurrence of the item is synchronized.
USAGE DISPLAY or PACKED-DECIMAL	Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.
USAGE NATIONAL	Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.

Table 18. **SYNCHRONIZE** clause effect on other language elements (continued)

Language element	Comments
USAGE BINARY or COMPUTATIONAL	<p>When the item is the first elementary item subordinate to an item that contains a REDEFINES clause, the item must not require the addition of unused character positions.</p> <p>When the synchronized clause is not specified for a subordinate data item (one with a level number of 02 through 49):</p> <ul style="list-style-type: none"> • The item is aligned at a displacement that is a multiple of 2 relative to the beginning of the record if its USAGE is BINARY and its PICTURE is in the range of S9 through S9(4). • The item is aligned at a displacement that is a multiple of 4 relative to the beginning of the record if its USAGE is BINARY and its PICTURE is in the range of S9(5) through S9(18), or its USAGE is INDEX. <p>When SYNCHRONIZED is not specified for binary items, no space is reserved for slack bytes.</p>
USAGE POINTER, PROCEDURE-POINTER, FUNCTION-POINTER, OBJECT REFERENCE	<p>The data is aligned on a fullword boundary when the ADDR(32) compiler option is specified.</p> <p>The data is aligned on a doubleword boundary when the ADDR(64) compiler option is specified.</p>
USAGE COMPUTATIONAL-1	The data is aligned on a fullword boundary.
USAGE COMPUTATIONAL-2	The data is aligned on a doubleword boundary.
USAGE COMPUTATIONAL-3	The data is treated the same as the SYNCHRONIZED clause for a PACKED-DECIMAL item.
USAGE COMPUTATIONAL-4	The data is treated the same as the SYNCHRONIZED clause for a COMPUTATIONAL item.
USAGE COMPUTATIONAL-5	The data is treated the same as the SYNCHRONIZED clause for a COMPUTATIONAL item.
DBCS and external floating-point items	Each item is syntax checked, but the SYNCHRONIZED clause has no effect on execution.
REDEFINES clause	<p>For an item that contains a REDEFINES clause, the data item that is redefined must have the proper boundary alignment for the data item that redefines it. For example, if you write the following, be sure that data item A begins on a fullword boundary:</p> <pre>02 A PICTURE X(4). 02 B REDEFINES A PICTURE S9(9) BINARY SYNC.</pre>

In the file section, the compiler assumes that all level-01 records that contain SYNCHRONIZED items are aligned on doubleword boundaries in the buffer. You must provide the necessary slack bytes between records to ensure alignment when there are multiple records in a block.

In the working-storage section, the compiler aligns all level-01 entries on a doubleword boundary.

For the purposes of aligning binary items in the linkage section, all level-01 items are assumed to begin on doubleword boundaries. Therefore, if you issue a CALL statement, such operands of any USING phrase within it must be aligned correspondingly.

Slack bytes

There are two types of slack bytes:

- Slack bytes *within* records: unused character positions that precede each synchronized item in the record
- Slack bytes *between* records: unused character positions added between blocked logical records

Slack bytes within records

For any data description that has binary items that are not on their natural boundaries, the compiler inserts slack bytes within a record to ensure that all SYNCHRONIZED items are on their proper boundaries.

Because it is important that you know the length of the records in a file, you need to determine whether slack bytes are required and, if so, how many bytes the compiler will add. The algorithm that the compiler uses is as follows:

- The total number of bytes occupied by all elementary data items that precede the binary item are added together, including any slack bytes that are previously added.
- This sum is divided by m , where:
 - $m = 2$ for binary items of four-digit length or less
 - $m = 4$ for binary items of five-digit length or more and for COMPUTATIONAL-1 data items
 - $m = 4$ or 8 for data items described with USAGE INDEX, USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE OBJECT REFERENCE, or USAGE FUNCTION-POINTER
 - $m = 8$ for COMPUTATIONAL-2 data items
- If the remainder (r) of this division is equal to zero, no slack bytes are required. If the remainder is not equal to zero, the number of slack bytes that must be added is equal to $m - r$.

These slack bytes are added to each record immediately following the elementary data item that precedes the binary item. They are defined as if they constitute an item with a level-number equal to that of the elementary item that immediately precedes the SYNCHRONIZED binary item, and are included in the size of the group that contains them.

For example:

```
01 FIELD-A.
   05 FIELD-B          PICTURE X(5).
   05 FIELD-C.
      10 FIELD-D        PICTURE XX.
      [10 SLACK-BYTES    PICTURE X.  INSERTED BY COMPILER]
      10 FIELD-E COMPUTATIONAL PICTURE S9(6) SYNC.
01 FIELD-L.
   05 FIELD-M          PICTURE X(5).
   05 FIELD-N          PICTURE XX.
```

```

[05 SLACK-BYTES                PICTURE X.  INSERTED BY COMPILER]
05 FIELD-0.
    10 FIELD-P COMPUTATIONAL    PICTURE S9(6) SYNC.

```

Slack bytes can also be added by the compiler when a group item is defined with an OCCURS clause and contains within it a SYNCHRONIZED binary data item. To determine whether slack bytes are to be added, the following action is taken:

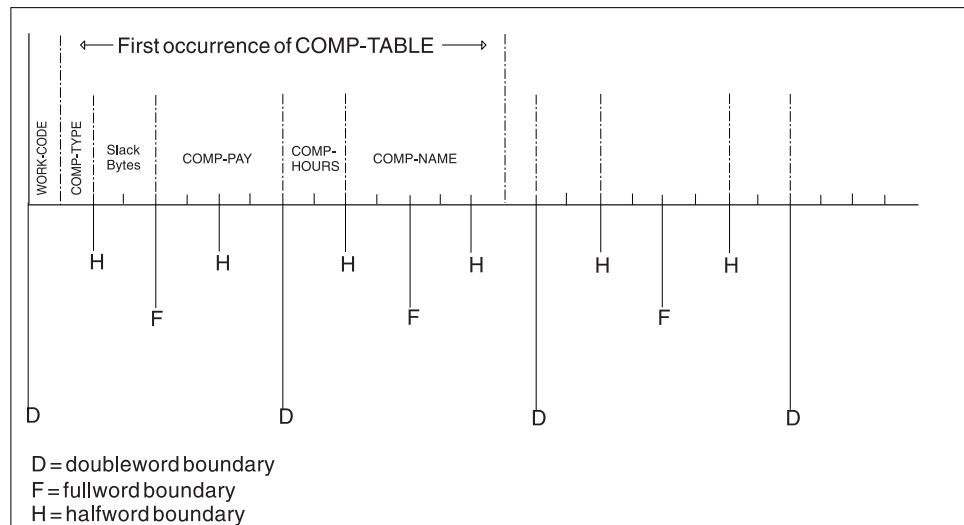
- The compiler calculates the size of the group, including all the necessary slack bytes within a record.
- This sum is divided by the largest m required by any elementary item within the group.
- If r is equal to zero, no slack bytes are required. If r is not equal to zero, $m - r$ slack bytes must be added.

The slack bytes are inserted at the end of each occurrence of the group item that contains the OCCURS clause. For example, a record defined as follows appears in storage, as shown, in the figure after the record:

```

01 WORK-RECORD.
    05 WORK-CODE                PICTURE X.
    05 COMP-TABLE OCCURS 10 TIMES.
        10 COMP-TYPE            PICTURE X.
        [10 SLACK-BYTES          PIC XX.  INSERTED BY COMPILER]
        10 COMP-PAY              PICTURE S9(4)V99 COMP SYNC.
        10 COMP-HOURS            PICTURE S9(3) COMP SYNC.
        10 COMP-NAME             PICTURE X(5).

```



In order to align COMP-PAY and COMP-HOURS on their proper boundaries, the compiler added 2 slack bytes within the record.

In the previous example, without further adjustment, the second occurrence of COMP-TABLE would begin 1 byte before a doubleword boundary, and the alignment of COMP-PAY and COMP-HOURS would not be valid for any occurrence of the table after the first. Therefore, the compiler must add slack bytes at the end of the group, as though the record had been written as follows:

```

01 WORK-RECORD.
    05 WORK-CODE                PICTURE X.
    05 COMP-TABLE OCCURS 10 TIMES.
        10 COMP-TYPE            PICTURE X.
        [10 SLACK-BYTES          PIC XX.  INSERTED BY COMPILER]
        10 COMP-PAY              PICTURE S9(4)V99 COMP SYNC.

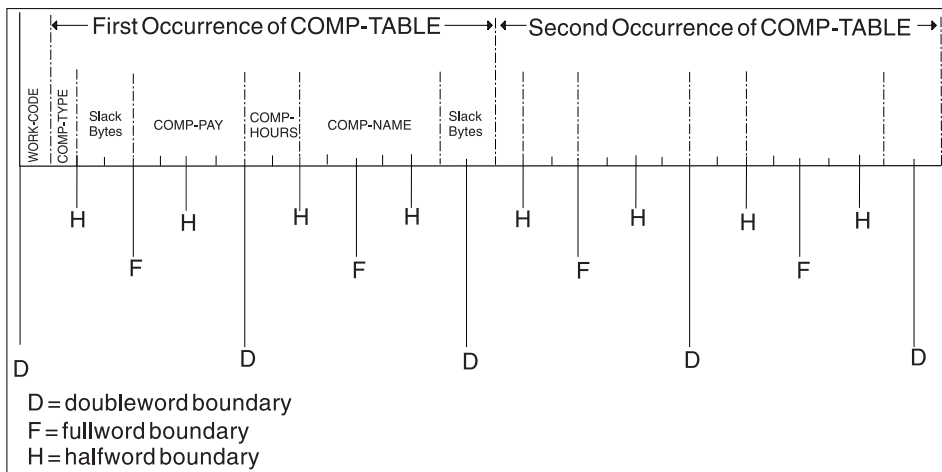
```

```

10 COMP-HOURS      PICTURE S9(3)  COMP SYNC.
10 COMP-NAME       PICTURE X(5).
[10 SLACK-BYTES     PIC XX.  INSERTED BY COMPILER]

```

In this example, the second and each succeeding occurrence of COMP-TABLE begins 1 byte beyond a doubleword boundary. The storage layout for the first occurrence of COMP-TABLE now appears as shown in the following figure:



Each succeeding occurrence within the table will now begin at the same relative position as the first.

Slack bytes between records

The lengths of all the elementary data items in the record, including all slack bytes, are added. The total is then divided by the highest value of m for any one of the elementary items in the record.

If r (the remainder) is equal to zero, no slack bytes are required. If r is not equal to zero, $m - r$ slack bytes are required. These slack bytes can be specified by writing a level-02 FILLER at the end of the record.

Consider the following record description:

```

01 COMP-RECORD.
  05 A-1  PICTURE X(5).
  05 A-2  PICTURE X(3).
  05 A-3  PICTURE X(3).
  05 B-1  PICTURE S9999  USAGE COMP SYNCHRONIZED.
  05 B-2  PICTURE S99999 USAGE COMP SYNCHRONIZED.
  05 B-3  PICTURE S9999  USAGE COMP SYNCHRONIZED.

```

The number of bytes in A-1, A-2, and A-3 totals 11. B-1 is a four-digit COMPUTATIONAL item and 1 slack byte must therefore be added before B-1. With this byte added, the number of bytes that precede B-2 totals 14. Because B-2 is a COMPUTATIONAL item of five digits in length, 2 slack bytes must be added before it. No slack bytes are needed before B-3.

The revised record description entry now appears as:

```

01 COMP-RECORD.
  05 A-1  PICTURE X(5).
  05 A-2  PICTURE X(3).
  05 A-3  PICTURE X(3).

```

```

[05 SLACK-BYTE-1  PICTURE X.  INSERTED BY COMPILER]
05 B-1           PICTURE S9999 USAGE COMP SYNCHRONIZED.
[05 SLACK-BYTE-2  PICTURE XX. INSERTED BY COMPILER]
05 B-2           PICTURE S99999 USAGE COMP SYNCHRONIZED.
05 B-3           PICTURE S9999  USAGE COMP SYNCHRONIZED.

```

There is a total of 22 bytes in COMP-RECORD, but from the rules above, it appears that $m = 4$ and $r = 2$. Therefore, to attain proper alignment for blocked records, you must add 2 slack bytes at the end of the record.

The final record description entry appears as:

```

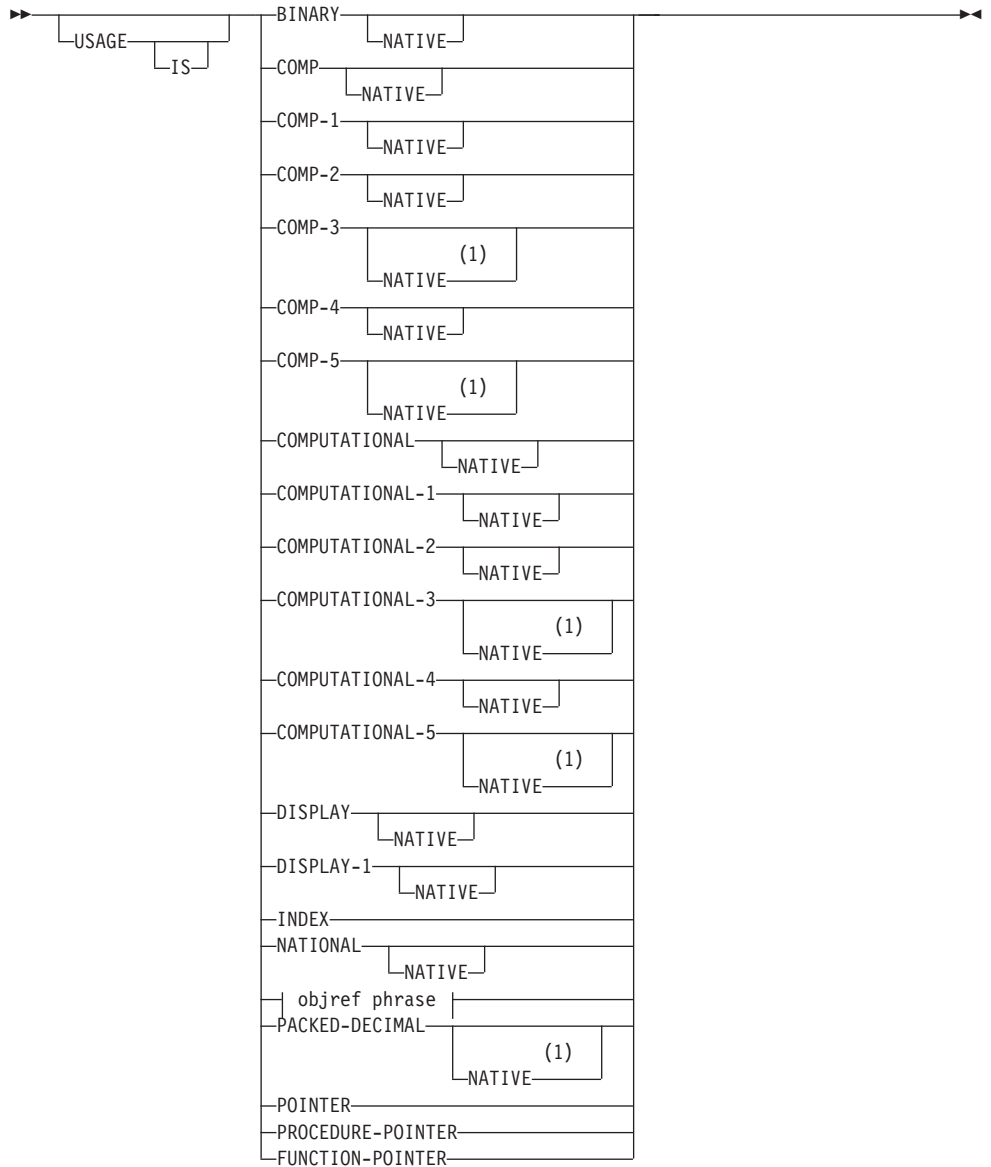
01 COMP-RECORD.
05 A-1          PICTURE X(5).
05 A-2          PICTURE X(3).
05 A-3          PICTURE X(3).
[05 SLACK-BYTE-1 PICTURE X.  INSERTED BY COMPILER]
05 B-1          PICTURE S9999 USAGE COMP SYNCHRONIZED.
[05 SLACK-BYTE-2 PICTURE XX. INSERTED BY COMPILER]
05 B-2          PICTURE S99999 USAGE COMP SYNCHRONIZED.
05 B-3          PICTURE S9999  USAGE COMP SYNCHRONIZED.
05 FILLER       PICTURE XX.  [SLACK BYTES YOU ADD]

```

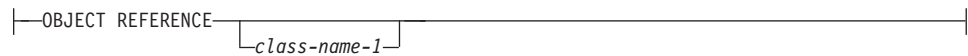
USAGE clause

The USAGE clause specifies the format in which data is represented in storage.

Format 1



objref phrase:



Notes:

- 1 The NATIVE phrase is treated as a comment for COMP-3, COMPUTATIONAL-3, COMP-5, COMPUTATIONAL-5, NATIONAL, and PACKED-DECIMAL data items.

The USAGE clause can be specified for a data description entry with any level-number other than 66 or 88.

When specified at the group level, the USAGE clause applies to each elementary item in the group. The usage of elementary items must not contradict the usage of a group to which the elementary items belongs.

A USAGE clause must not be specified in a group level entry for which a GROUP-USAGE NATIONAL clause is specified.

When a GROUP-USAGE NATIONAL clause is specified or implied for a group level entry, USAGE NATIONAL must be specified or implied for every elementary item within the group. For details, see “GROUP-USAGE clause” on page 191.

When the USAGE clause is not specified at either the group or elementary level, a usage clause is implied with:

- Usage DISPLAY when the PICTURE clause contains only symbols other than G or N
- Usage NATIONAL when the PICTURE clause contains only one or more of the symbol N and the NSYMBOL(NATIONAL) compiler option is in effect
- Usage DISPLAY-1 when the PICTURE clause contains one or more of the symbol N and the NSYMBOL(DBCS) compiler option is in effect

For data items defined with the DATE FORMAT clause, only usage DISPLAY and COMP-3 (or its equivalents, COMPUTATIONAL-3 and PACKED-DECIMAL) are allowed. For details, see “Combining the DATE FORMAT clause with other clauses” on page 186.

Computational items

A computational item is a value used in arithmetic operations. It must be numeric. If a group item is described with a computational usage, the elementary items within the group have that usage.

The maximum length of a computational item is 18 decimal digits, except for a PACKED-DECIMAL item. If the ARITH(COMPAT) compiler option is in effect, then the maximum length of a PACKED-DECIMAL item is 18 decimal digits. If the ARITH(EXTEND) compiler option is in effect, then the maximum length of a PACKED-DECIMAL item is 31 decimal digits.

The PICTURE of a computational item can contain only:

- 9** One or more numeric character positions
- S** One operational sign
- V** One implied decimal point
- P** One or more decimal scaling positions

COMPUTATIONAL-1 and COMPUTATIONAL-2 items (internal floating-point) cannot have PICTURE strings.

BINARY

Specified for binary data items. Such items have a decimal equivalent consisting of the decimal digits 0 through 9, plus a sign. Negative numbers are represented as the two's complement of the positive number with the same absolute value.

The amount of storage occupied by a binary item depends on the number of decimal digits defined in its PICTURE clause:

Digits in PICTURE clause	Storage occupied
1 through 4	2 bytes (halfword)
5 through 9	4 bytes (fullword)
10 through 18	8 bytes (doubleword)

Binary data is *big-endian*: the operational sign is contained in the leftmost bit.

BINARY, COMPUTATIONAL, and COMPUTATIONAL-4 data items can be affected by the TRUNC compiler option. For information about the effect of this compiler option, see *TRUNC* in the *COBOL for AIX Programming Guide*.

PACKED-DECIMAL

Specified for internal decimal items. Such an item appears in storage in packed decimal format. There are two digits for each character position, except for the trailing character position, which is occupied by the low-order digit and the sign. Such an item can contain any of the digits 0 through 9, plus a sign, representing a value not exceeding 18 decimal digits.

The sign representation uses the same bit configuration as the 4-bit sign representation in zoned decimal fields. For details, see *Sign representation of zoned and packed-decimal data* in the *COBOL for AIX Programming Guide*.

COMPUTATIONAL or COMP (binary)

This is the equivalent of BINARY. The COMPUTATIONAL phrase is synonymous with BINARY.

COMPUTATIONAL-1 or COMP-1 (floating-point)

Specified for internal floating-point items (single precision). COMP-1 items are 4 bytes long.

COMP-1 data items are affected by the FLOAT(NATIVE|HEX) compiler option. For details, see *FLOAT* in the *COBOL for AIX Programming Guide*.

COMPUTATIONAL-2 or COMP-2 (long floating-point)

Specified for internal floating-point items (double precision). COMP-2 items are 8 bytes long.

COMP-2 data items are affected by the FLOAT(NATIVE|HEX) compiler option. For details, see *FLOAT* in the *COBOL for AIX Programming Guide*.

COMPUTATIONAL-3 or COMP-3 (internal decimal)

This is the equivalent of PACKED-DECIMAL.

COMPUTATIONAL-4 or COMP-4 (binary)

This is the equivalent of BINARY.

COMPUTATIONAL-5 or COMP-5 (native binary)

These data items are represented in storage as binary data. The data items can contain values up to the capacity of the native binary representation (2, 4, or 8 bytes), rather than being limited to the value implied by the number of nines in the picture for the item (as is the case for USAGE BINARY data). When numeric data is moved or stored into a COMP-5 item, truncation occurs at the binary field size rather than at the COBOL picture size limit. When a COMP-5 item is referenced, the full binary field size is used in the operation.

The TRUNC(BIN) compiler option causes all binary data items (USAGE BINARY, COMP, COMP-4) to be handled as if they were declared USAGE COMP-5.

The following table shows several picture character strings, the resulting storage representation, and the range of values for data items described with USAGE COMP-5.

Picture	Storage representation	Numeric values
S9(1) through S9(4)	Binary halfword (2 bytes)	-32768 through +32767
S9(5) through S9(9)	Binary fullword (4 bytes)	-2,147,483,648 through +2,147,483,647
S9(10) through S9(18)	Binary doubleword (8 bytes)	-9,223,372,036,854,775,808 through +9,223,372,036,854,775,807
9(1) through 9(4)	Binary halfword (2 bytes)	0 through 65535
9(5) through 9(9)	Binary fullword (4 bytes)	0 through 4,294,967,295
9(10) through 9(18)	Binary doubleword (8 bytes)	0 through 18,446,744,073,709,551,615

The picture for a COMP-5 data item can specify a scaling factor (that is, decimal positions or implied integer positions). In this case, the maximal capacities listed in the table above must be scaled appropriately. For example, a data item described with PICTURE S99V99 COMP-5 is represented in storage as a binary halfword, and supports a range of values from -327.68 to +327.67.

USAGE NOTE: When the ON SIZE ERROR phrase is used on an arithmetic statement and a receiver is defined with USAGE COMP-5, the maximum value that the receiver can contain is the value implied by the item's decimal PICTURE character-string. Any attempt to store a value larger than this maximum will result in a size error condition.

DISPLAY phrase

The data item is stored in character form, one character for each 8-bit byte. This corresponds to the format used for printed output. DISPLAY can be explicit or implicit.

USAGE IS DISPLAY is valid for the following types of items:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- Numeric-edited
- External floating-point
- External decimal

Alphabetic, alphanumeric, alphanumeric-edited, and numeric-edited items are discussed in "Data categories and PICTURE rules" on page 204.

External decimal items with USAGE DISPLAY are sometimes referred to as *zoned decimal* items. Each digit of a number is represented by a single byte. The 4

high-order bits of each byte are zone bits; the 4 high-order bits of the low-order byte represent the sign of the item. The 4 low-order bits of each byte contain the value of the digit.

If the ARITH(COMPAT) compiler option is in effect, then the maximum length of an external decimal item is 18 digits. If the ARITH(EXTEND) compiler option is in effect, then the maximum length of an external decimal item is 31 digits.

The PICTURE character-string of an external decimal item can contain only:

- One or more of the symbol 9
- The operational-sign, S
- The assumed decimal point, V
- One or more of the symbol P

Effect of CHAR(EBCDIC) compiler option

Data items defined with the DISPLAY or DISPLAY-1 phrase are treated as EBCDIC when the CHAR(EBCDIC) compiler option is used, unless the character data is defined with the NATIVE phrase.

DISPLAY-1 phrase

The DISPLAY-1 phrase defines an item as DBCS. The data item is stored in character form, with each character occupying 2 bytes of storage.

FUNCTION-POINTER phrase

The FUNCTION-POINTER phrase defines an item as a *function-pointer data item*. A function-pointer data item can contain the address of a procedure entry point.

A function-pointer is a 4-byte elementary item or an 8-byte elementary item depending on whether the ADDR(32) or ADDR(64) compiler option is in effect. Function-pointers have the same capabilities as procedure-pointers. Function-pointers are easily interoperable with C function pointers.

A function-pointer can contain one of the following addresses or can contain NULL:

- The primary entry point of a COBOL program, defined by the PROGRAM-ID paragraph of the outermost program
- An alternate entry point of a COBOL program, defined by a COBOL ENTRY statement
- An entry point in a non-COBOL program

A VALUE clause for a function-pointer data item can contain only NULL or NULLS.

A function-pointer can be used in the same contexts as a procedure-pointer, as defined in “PROCEDURE-POINTER phrase” on page 237.

INDEX phrase

A data item defined with the INDEX phrase is an *index data item*.

An *index data item* is a 4-byte elementary item or an 8-byte elementary item depending on whether the ADDR(32) or ADDR(64) compiler option is in effect, that can be used to save index-name values for future reference. An *index data item* is not necessarily connected with any specific table. Through a SET statement, an index data item can be assigned an index-name value. Such a value corresponds to the occurrence number in a table.

Direct references to an index data item can be made only in a SEARCH statement, a SET statement, a relation condition, the USING phrase of the procedure division header, or the USING phrase of the CALL or ENTRY statement.

An index data item can be part of an alphanumeric group item that is referenced in a MOVE statement or an input/output statement.

An index data item saves values that represent table occurrences, yet is not necessarily defined as part of any table. There is no conversion of values when an index data item is referenced in the following circumstances:

- directly in a SEARCH or SET statement
- indirectly in a MOVE statement
- indirectly in an input or output statement

An index data item cannot be a conditional variable.

The DATE FORMAT, JUSTIFIED, PICTURE, BLANK WHEN ZERO, or VALUE clauses cannot be used to describe a group item or elementary items described with the USAGE IS INDEX clause.

SYNCHRONIZED can be used with USAGE IS INDEX to obtain efficient use of the index data item.

NATIONAL phrase

The NATIONAL phrase defines an item whose content is represented in storage in UTF-16 (CCSID 1200). The class and category of the data item depend on the picture symbols that are specified in the associated PICTURE clause.

OBJECT REFERENCE phrase

A data item defined with the OBJECT REFERENCE phrase is an *object reference*. An object reference data item is a 4-byte elementary item or an 8-byte elementary item depending on whether the ADDR(32) or ADDR(64) compiler option is in effect.

class-name-1

An optional class name.

You must declare *class-name-1* in the REPOSITORY paragraph in the configuration section of the containing class or outermost program.

If specified, *class-name-1* indicates that *data-name-1* always refers to an object-instance of class *class-name-1* or a class derived from *class-name-1*.

Important: The programmer must ensure that the referenced object meets this requirement; violations are not diagnosed.

If *class-name-1* is not specified, the object reference can refer to an object of any class. In this case, *data-name-1* is a *universal* object reference.

You can specify *data-name-1* within an alphanumeric group item without affecting the semantics of the group item. There is no conversion of values or other special handling of the object references when statements are executed that operate on the group. The group continues to behave as an alphanumeric group item.

An object reference can be defined in any section of the data division of a factory definition, object definition, method, or program. An object-reference data item can be used in only:

- A SET statement (format 7 only)
- A relation condition
- An INVOKE statement
- The USING or RETURNING phrase of an INVOKE statement
- The USING or RETURNING phrase of a CALL statement
- A program procedure division or ENTRY statement USING or RETURNING phrase
- A method procedure division USING or RETURNING phrase

Object-reference data items:

- Are ignored in CORRESPONDING operations
- Are unaffected by INITIALIZE statements
- Can be the subject or object of a REDEFINES clause
- Cannot be a conditional variable
- Can be written to a file (but upon subsequent reading of the record the content of the object reference is undefined)

A VALUE clause for an object-reference data item can contain only NULL or NULLS.

You can use the SYNCHRONIZED clause with the USAGE OBJECT REFERENCE clause to obtain efficient alignment of the object-reference data item.

The DATE FORMAT, JUSTIFIED, PICTURE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items defined with the USAGE OBJECT REFERENCE clause.

POINTER phrase

A data item defined with USAGE IS POINTER is a *pointer data item*. A pointer data item is a 4-byte elementary item or an 8-byte elementary item depending on whether the ADDR(32) or ADDR(64) compiler option is in effect.

You can use pointer data items to accomplish limited base addressing. Pointer data items can be compared for equality or moved to other pointer items.

A pointer data item can be used only:

- In a SET statement (format 5 only)
- In a relation condition
- In the USING phrase of a CALL statement, an ENTRY statement, or the procedure division header

Pointer data items can be part of an alphanumeric group that is referred to in a MOVE statement or an input/output statement. However, if a pointer data item is part of a group, there is no conversion of values when the statement is executed.

A pointer data item can be the subject or object of a REDEFINES clause.

SYNCHRONIZED can be used with USAGE IS POINTER to obtain efficient use of the pointer data item.

A VALUE clause for a pointer data item can contain only NULL or NULLS.

A pointer data item cannot be a conditional variable.

A pointer data item does not belong to any class or category.

The DATE FORMAT, JUSTIFIED, PICTURE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items defined with the USAGE IS POINTER clause.

Pointer data items are ignored in the processing of a CORRESPONDING phrase.

A pointer data item can be written to a file, but upon subsequent reading of the record that contains the pointer, the address contained might no longer represent a valid pointer.

USAGE IS POINTER is implicitly specified for the ADDRESS OF special register. For more information, see *Using tables (arrays) and pointers* in the *COBOL for AIX Programming Guide*.

PROCEDURE-POINTER phrase

The PROCEDURE-POINTER phrase defines an item as a *procedure-pointer data item*.

A procedure-pointer data item is a 4-byte elementary item or an 8-byte elementary item depending on whether the ADDR(32) or ADDR(64) compiler option is in effect.

A procedure-pointer can contain one of the following addresses or can contain NULL:

- The primary entry point of a COBOL program as defined by the program-ID paragraph of the outermost program of a compilation unit
- An alternate entry point of a COBOL program as defined by a COBOL ENTRY statement
- An entry point in a non-COBOL program

A procedure-pointer data item can be used only:

- In a SET statement (format 6 only)
- In a CALL statement
- In a relation condition
- In the USING phrase of an ENTRY statement or the procedure division header

Procedure-pointer data items can be compared for equality or moved to other procedure-pointer data items.

Procedure-pointer data items can be part of a group that is referred to in a MOVE statement or an input/output statement. However, there is no conversion of values when the statement is executed. If a procedure-pointer data item is written to a file, subsequent reading of the record that contains the procedure-pointer can result in an invalid value in the procedure-pointer.

A procedure-pointer data item can be the subject or object of a REDEFINES clause.

SYNCHRONIZED can be used with USAGE IS PROCEDURE-POINTER to obtain efficient alignment of the procedure-pointer data item.

The GLOBAL, EXTERNAL, and OCCURS clause can be used with USAGE IS PROCEDURE-POINTER.

A VALUE clause for a procedure-pointer data item can contain only NULL or NULLS.

The DATE FORMAT, JUSTIFIED, PICTURE, and BLANK WHEN ZERO clauses cannot be used to describe group or elementary items defined with the USAGE IS PROCEDURE-POINTER clause.

A procedure-pointer data item cannot be a conditional variable.

A procedure-pointer data item does not belong to any class or category.

Procedure-pointer data items are ignored in CORRESPONDING operations.

NATIVE phrase

By using the NATIVE phrase, you can mix characters and floating-point data as represented on the z/OS[®] or OS/390[®] and AIX platforms. The NATIVE phrase overrides the CHAR(EBCDIC) and FLOAT(HEX) compiler options, which indicate host data type usages.

The use of both host and native data types within a program (ASCII and EBCDIC, and Hex Floating-point and IEEE floating-point) is valid only for those data items specifically defined with the NATIVE phrase.

Specifying NATIVE does not change the class or the category of the data item.

Numeric data items are processed in arithmetic operations (numeric comparisons, arithmetic expressions, assignment to numeric targets, arithmetic statements) based on their logical numeric values, regardless of their internal representations.

Characters are converted to the representation of the target item prior to an assignment.

Comparisons are done based on the collating sequence rules applicable to the operands. If native and non-native alphanumeric or DBCS characters are compared, the comparison is based on the COLLSEQ option in effect.

VALUE clause

The VALUE clause specifies the initial contents of a data item or the values associated with a condition-name. The use of the VALUE clause differs depending on the data division section in which it is specified.

A VALUE clause that is used in the file section or the linkage section in an entry other than a condition-name entry is syntax checked, but has no effect on the execution of the program.

In the working-storage section and the local-storage section, the VALUE clause can be used in condition-name entries or in specifying the initial value of any data item. The data item assumes the specified value at the beginning of program execution. If the initial value is not explicitly specified, the value is unpredictable.

Format 1

Format 1: literal value



Format 1 specifies the initial value of a data item. Initialization is independent of any BLANK WHEN ZERO or JUSTIFIED clause that is specified.

A format-1 VALUE clause specified in a data description entry that contains or is subordinate to an OCCURS clause causes every occurrence of the associated data item to be assigned the specified value. Each structure that contains the DEPENDING ON phrase of the OCCURS clause is assumed to contain the maximum number of occurrences for the purposes of VALUE initialization.

The VALUE clause must not be specified for a data description entry that contains or is subordinate to an entry that contains either an EXTERNAL or a REDEFINES clause. This rule does not apply to condition-name entries.

A format-1 VALUE clause can be specified for an elementary data item or for a group item. When the VALUE clause is specified at the group level, the group area is initialized without consideration for the subordinate entries within the group. In addition, a VALUE clause must not be specified for subordinate entries within the group.

For group items, the VALUE clause must not be specified if any subordinate entries contain a JUSTIFIED or SYNCHRONIZED clause.

If the VALUE clause is specified for an alphanumeric group, all subordinate items must be explicitly or implicitly described with USAGE DISPLAY.

The VALUE clause must not conflict with other clauses in the data description entry or in the data description of that entry's hierarchy.

The functions of the editing characters in a PICTURE clause are ignored in determining the initial value of the item described. However, editing characters are included in determining the size of the item. Therefore, any editing characters must be included in the literal. For example, if the item is defined as PICTURE +999.99 and the value is to be +12.34, then the VALUE clause should be specified as VALUE "+012.34".

A VALUE clause cannot be specified for external floating-point items.

A data item cannot contain a VALUE clause if the prior data item contains an OCCURS clause with the DEPENDING ON phrase.

Rules for literal values

- Wherever a literal is specified, a figurative constant can be substituted, in accordance with the rules specified in “Figurative constants” on page 13.
- If the item is class numeric, the VALUE clause literal must be numeric. If the literal defines the value of a working-storage item or local-storage item, the literal is aligned according to the rules for numeric moves, with one additional restriction: The literal must not have a value that requires truncation of nonzero digits. If the literal is signed, the associated PICTURE character-string must contain a sign symbol.
- With some exceptions, numeric literals in a VALUE clause must have a value within the range of values indicated by the PICTURE clause for the item. For example, for PICTURE 99PPP, the literal must be zero or within the range 1000 through 99000. For PICTURE PPP99, the literal must be within the range 0.00000 through 0.00099.

The exceptions are the following:

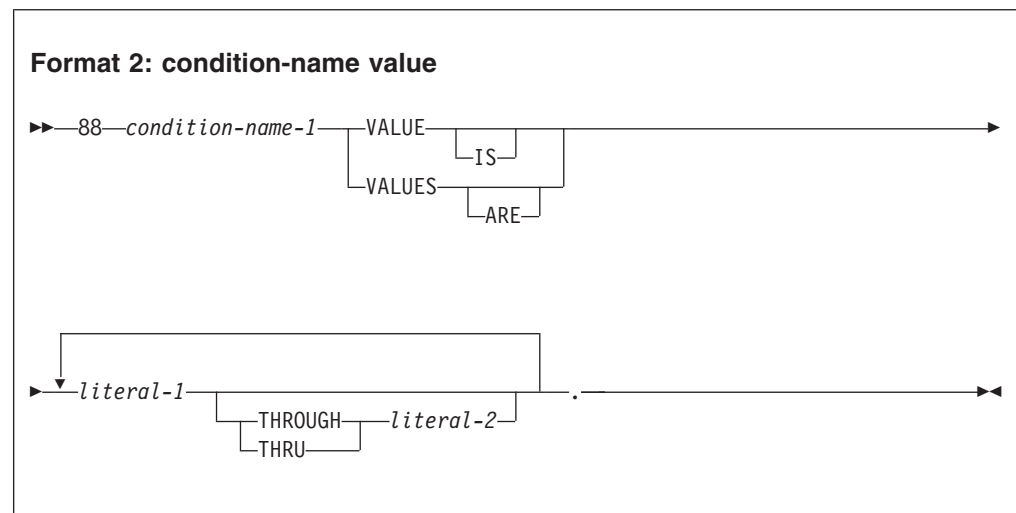
- Data items described with usage COMP-5 that do not have a picture symbol P in their PICTURE clause
- When the TRUNC(BIN) compiler option is in effect, data items described with usage BINARY, COMP, or COMP-4 that do not have a picture symbol P in their PICTURE clause

A VALUE clause for these items can have a value up to the capacity of the native binary representation.

- If the VALUE clause is specified for an elementary alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited item described with usage DISPLAY, the VALUE clause literal must be an alphanumeric literal or a figurative constant. The literal is aligned according to the alphanumeric alignment rules, with one additional restriction: the number of characters in the literal must not exceed the size of the item.
- If the VALUE clause is specified for an elementary national, national-edited, or numeric-edited item described with usage NATIONAL, the VALUE clause literal must be a national or alphanumeric literal or a figurative constant as specified in “Figurative constants” on page 13. The value of an alphanumeric literal is converted from its source code representation to UTF-16 representation. The literal is aligned according to the national alignment rules, with one additional restriction: the number of characters in the literal must not exceed the size, in character positions, of the item.
- If the VALUE clause is specified at the group level for an alphanumeric group, the literal must be an alphanumeric literal or a figurative constant as specified in “Figurative constants” on page 13, other than ALL *national-literal*. The size of the literal must not exceed the size of the group item.

- If the VALUE clause is specified at the group level for a national group, the literal can be an alphanumeric literal, a national literal, or one of the figurative constants ZERO, SPACE, QUOTES, HIGH-VALUE, LOW-VALUE, *symbolic character*, ALL *national-literal*, or ALL *-literal*. The value of an alphanumeric literal is converted from its source code representation to UTF-16 representation. Each figurative constant represents a national character value. The size of the literal must not exceed the size of the group item.
- A VALUE clause associated with a DBCS item must contain a DBCS literal, the figurative constant SPACE, or the figurative constant ALL *DBCS-literal*. The length of the literal must not exceed the size indicated by the data item's PICTURE clause.
- A VALUE clause that specifies a national literal can be associated only with a data item of class national.
- A VALUE clause that specifies a DBCS literal can be associated only with a data item of class DBCS.
- A VALUE clause associated with a COMPUTATIONAL-1 or COMPUTATIONAL-2 (internal floating-point) item must specify a floating-point literal. In addition, the figurative constant ZERO and both integer and decimal forms of the zero literal can be specified in a floating-point VALUE clause.
For information about floating-point literal values, see “Rules for floating-point literal values” on page 32.

Format 2



This format associates a value, values, or ranges of values with a condition-name. Each such condition-name requires a separate level-88 entry. Level-number 88 and the condition-name are not part of the format-2 VALUE clause itself. They are included in the format only for clarity.

condition-name-1

A user-specified name that associates a value with a conditional variable. If the associated conditional variable requires subscripts or indexes, each procedural reference to the condition-name must be subscripted or indexed as required for the conditional variable.

Condition-names are tested procedurally in condition-name conditions (see “Conditional expressions” on page 262).

literal-1

Associates the condition-name with a single value.

The class of *literal-1* must be a valid class for assignment to the associated conditional variable.

literal-1* THROUGH *literal-2

Associates the condition-name with at least one range of values. When the THROUGH phrase is used, *literal-1* must be less than *literal-2*, unless the associated data item is a non-year-last windowed date field. For details, see "Rules for condition-name entries."

literal-1 and *literal-2* must be of the same class. The class of *literal-1* and *literal-2* must be a valid class for assignment to the associated conditional variable.

The range of alphanumeric literals, national literals, or DBCS literals specified for the THROUGH phrase is based on the collating sequence in effect for the associated conditional variable. For more information about collating sequences, see Appendix G, "Locale considerations," on page 609.

If the associated conditional variable is of class DBCS, *literal-1* and *literal-2* must be DBCS literals. The figurative constant SPACE or the figurative constant ALL *DBCS-literal* can be specified.

If the associated conditional variable is of class national, *literal-1* and *literal-2* must be either both national literals or both alphanumeric literals for a given condition-name. The figurative constants ZERO, SPACE, QUOTE, HIGH-VALUE, LOW-VALUE, *symbolic-character*, ALL *national-literal*, or ALL *literal* can be specified.

Rules for condition-name entries

- The VALUE clause is required in a condition-name entry, and must be the only clause in the entry. Each condition-name entry is associated with a preceding conditional variable. Thus every level-88 entry must always be preceded either by the entry for the conditional variable or by another level-88 entry when several condition-names apply to one conditional variable. Each such level-88 entry implicitly has the PICTURE characteristics of the conditional variable.
- A space, a separator comma, or a separator semicolon must separate successive operands.

Each entry must end with a separator period.
- The keywords THROUGH and THRU are equivalent.
- The condition-name entries associated with a particular conditional variable must immediately follow the conditional variable entry. The conditional variable can be any elementary data description entry except the following:
 - Another condition-name
 - A RENAME clause (level-66 item)
 - An item described with USAGE IS INDEX
 - An item described with USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, or USAGE OBJECT REFERENCE
- Condition-names can be specified both at the group level and at subordinate levels within an alphanumeric group or national group.
- When the condition-name is specified for an alphanumeric group data description entry:
 - The value of *literal-1* (or *literal-1* and *literal-2*) must be specified as an alphanumeric literal or figurative constant.

- The group can contain items of any usage.
- When the condition-name is specified for a national group data description entry:
 - The value of *literal-1* (or *literal-1* and *literal-2*) must be specified as an alphanumeric literal, a national literal, or a figurative constant.
 - The group can contain only items of usage national, as specified for the “GROUP-USAGE clause” on page 191.
- When the condition-name is associated with an alphanumeric group data description entry or a national group data description entry:
 - The size of each literal value must not exceed the sum of the sizes of all the elementary items within the group.
 - No element within the group can contain a JUSTIFIED or SYNCHRONIZED clause.
- Relation tests implied by the definition of a condition-name are performed in accordance with the rules referenced in the table below.

Table 19. Relation test references for condition-names

Type of conditional variable	Relation condition rules
Alphanumeric group item	“Group comparisons” on page 275
National group item (treated as elementary data item of class national)	“National comparisons” on page 273
Elementary data item of class alphanumeric	“Alphanumeric comparisons” on page 271
Elementary data item of class national	“National comparisons” on page 273
Elementary data item of class numeric	“Numeric comparisons” on page 274
Elementary data item of class DBCS	“DBCS comparisons” on page 273

- A VALUE clause that specifies a national literal can be associated with a condition-name defined only for a data item of class national.
- A VALUE clause that specifies a DBCS literal can be associated with a condition-name defined only for a data item of class DBCS.
- The literals in a condition-name entry for an elementary data item of class national or a national group item must be either national literals or alphanumeric literals, and *literal-1* and *literal-2* must be of the same class. For alphanumeric groups or elementary data items of other classes, the type of literal must be consistent with the data type of the conditional variable. In the following example:
 - CITY-COUNTY-INFO, COUNTY-NO, and CITY are conditional variables.
The PICTURE associated with COUNTY-NO limits the condition-name value to a two-digit numeric literal.
The PICTURE associated with CITY limits the condition-name value to a three-character alphanumeric literal.
 - The associated condition-names are level-88 entries.
Any values for the condition-names associated with CITY-COUNTY-INFO cannot exceed five characters.
Because this is an alphanumeric group item, the literal must be alphanumeric.

```

05 CITY-COUNTY-INFO.
   88 BRONX           VALUE "03NYC".
   88 BROOKLYN        VALUE "24NYC".
   88 MANHATTAN        VALUE "31NYC".
   88 QUEENS          VALUE "41NYC".

```

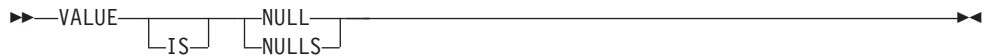
88	STATEN-ISLAND	VALUE "43NYC".
10	COUNTY-NO	PICTURE 99.
88	DUTCHESS	VALUE 14.
88	KINGS	VALUE 24.
88	NEW-YORK	VALUE 31.
88	RICHMOND	VALUE 43.
10	CITY	PICTURE X(3).
88	BUFFALO	VALUE "BUF".
88	NEW-YORK-CITY	VALUE "NYC".
88	POUGHKEEPSIE	VALUE "POK".
05	POPULATION...	

- If the item is a windowed date field, the following restrictions apply:
 - For alphanumeric conditional variables:
 - Both *literal-1* and *literal-2* (if specified) must be alphanumeric literals of the same length as the conditional variable.
 - The literals must not be specified as figurative constants.
 - If *literal-2* is specified, both literals must contain only decimal digits.
 - If the YEARWINDOW compiler option is specified as a negative integer, *literal-2* must not be specified.
 - If *literal-2* is specified, *literal-1* must be less than *literal-2* after applying the century window specified by the YEARWINDOW compiler option. That is, the expanded date value of *literal-1* must be less than the expanded date value of *literal-2*.

For more information about using condition-names with windowed date fields, see “Condition-name conditions and windowed date field comparisons” on page 266.

Format 3

Format 3: NULL value



This format assigns an invalid address as the initial value of an item defined as USAGE POINTER, USAGE PROCEDURE POINTER, or USAGE FUNCTION-POINTER. It also assigns an invalid object reference as the initial value of an item defined as USAGE OBJECT REFERENCE.

VALUE IS NULL can be specified only for elementary items described implicitly or explicitly as USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, or USAGE OBJECT REFERENCE.

Part 6. Procedure division

Chapter 20. Procedure division structure	249
Requirements for a method procedure division	250
The procedure division header	251
The USING phrase	252
RETURNING phrase	254
References to items in the linkage section	254
Declaratives	255
Procedures	256
Arithmetic expressions	257
Arithmetic operators	258
Arithmetic with date fields	260
Addition that involves date fields	260
Subtraction that involves date fields	261
Storing arithmetic results that involve date fields	261
Conditional expressions	262
Simple conditions	263
Class condition	263
Condition-name condition	265
Condition-name conditions and windowed date field comparisons	266
Relation conditions	266
General relation conditions	267
Alphanumeric comparisons	271
DBCS comparisons	273
National comparisons	273
Numeric comparisons	274
Group comparisons	275
Comparison of index-names and index data items	275
Comparison of date fields	276
Data pointer relation conditions	277
Procedure-pointer and function-pointer relation conditions	278
Object-reference relation conditions	278
Sign condition	279
Date fields in sign conditions	279
Switch-status condition	280
Complex conditions	280
Negated simple conditions	281
Combined conditions	282
Order of evaluation of conditions	283
Order of evaluation:	283
Abbreviated combined relation conditions	284
Using parentheses	284
Statement categories	286
Imperative statements	286
Arithmetic	287
Data movement	287
Ending	287
Input-output	287
Ordering	288
Procedure-branching	288
Program or method linkage	288
Table-handling	288
Conditional statements	288
Arithmetic	288
Data movement	289
Decision	289
Input-output	289
Ordering	289
Program or method linkage	289
Table-handling	289
Delimited scope statements	289
Explicit scope terminators	290
Implicit scope terminators	290
Compiler-directing statements	290
Statement operations	291
CORRESPONDING phrase	291
GIVING phrase	292
ROUNDED phrase	292
SIZE ERROR phrases	293
Arithmetic statements	294
Arithmetic statement operands	294
Size of operands	294
Overlapping operands	295
Multiple results	295
Data manipulation statements	296
Overlapping operands	296
Input-output statements	296
Common processing facilities	296
File status key	297
Invalid key condition	300
INTO and FROM phrases	301
File position indicator	302
Chapter 21. Procedure division statements	303
ACCEPT statement	304
Data transfer	304
System date-related information transfer	305
DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, and TIME	306
ADD statement	308
ROUNDED phrase	310
SIZE ERROR phrases	310
CORRESPONDING phrase (format 3)	310
END-ADD phrase	311
ALTER statement	312
Segmentation considerations	312
CALL statement	314
USING phrase	316
BY REFERENCE phrase	317
BY CONTENT phrase	317
BY VALUE phrase	318
RETURNING phrase	319
ON EXCEPTION phrase	320
NOT ON EXCEPTION phrase	320
ON OVERFLOW phrase	320
END-CALL phrase	321
CANCEL statement	322
CLOSE statement	324
Effect of CLOSE statement on file types	326

COMPUTE statement	328	COLLATING SEQUENCE phrase	376
ROUNDED phrase	329	USING phrase	377
SIZE ERROR phrases	329	GIVING phrase	377
END-COMPUTE phrase	329	OUTPUT PROCEDURE phrase	378
CONTINUE statement	330	MERGE special registers	378
DELETE statement	331	Segmentation considerations	379
Sequential access mode	331	MOVE statement	380
Random or dynamic access mode	332	Elementary moves	381
END-DELETE phrase	332	Elementary move rules	382
DISPLAY statement	333	Valid and invalid elementary moves	384
DIVIDE statement	335	Moves involving date fields	385
ROUNDED phrase	338	Moves involving file record areas	386
REMAINDER phrase	338	Group moves	386
SIZE ERROR phrases	338	MULTIPLY statement	388
END-DIVIDE phrase	339	ROUNDED phrase	390
ENTRY statement	340	SIZE ERROR phrases	390
USING phrase	340	END-MULTIPLY phrase	390
EVALUATE statement	341	OPEN statement	391
END-EVALUATE phrase	342	General rules	393
Determining values	342	Label records	393
Comparing selection subjects and objects	343	OPEN statement notes	393
Executing the EVALUATE statement	344	PERFORM statement	396
EXIT statement	345	Basic PERFORM statement	396
EXIT METHOD statement	346	END-PERFORM	398
EXIT PROGRAM statement	347	PERFORM with TIMES phrase	398
GOBACK statement	348	PERFORM with UNTIL phrase	399
GO TO statement	349	PERFORM with VARYING phrase	400
Unconditional GO TO	349	Varying identifiers	401
Conditional GO TO	349	Varying two identifiers	402
Altered GO TO	350	Varying three identifiers	404
MORE-LABELS GO TO	350	Varying more than three identifiers	404
IF statement	351	Varying phrase rules	405
END-IF phrase	351	READ statement	406
Transferring control	352	KEY IS phrase	408
Nested IF statements	352	AT END phrases	408
INITIALIZE statement	353	INVALID KEY phrases	408
REPLACING phrase	354	END-READ phrase	408
INITIALIZE statement rules	354	Multiple record processing	408
INSPECT statement	356	Sequential access mode	409
TALLYING phrase (formats 1 and 3)	359	Sequential files	409
REPLACING phrase (formats 2 and 3)	360	Indexed or relative files	410
BEFORE and AFTER phrases (all formats)	361	Random access mode	411
CONVERTING phrase (format 4)	362	Indexed files	411
Data flow	363	Relative files	411
Comparison cycle	364	Dynamic access mode	412
Example of the INSPECT statement	364	READ statement notes:	412
INVOKE statement	366	RELEASE statement	413
USING phrase	368	RETURN statement	415
BY VALUE phrase	368	AT END phrases	416
Conformance requirements for arguments	368	END-RETURN phrase	416
RETURNING phrase	369	REWRITE statement	417
Conformance requirements for the		INVALID KEY phrases	418
RETURNING item	370	END-REWRITE phrase	418
ON EXCEPTION phrase	370	Reusing a logical record	418
NOT ON EXCEPTION phrase	370	Sequential files	418
END-INVOKE phrase	371	Indexed files	418
Interoperable data types for COBOL and Java	371	Relative files	419
Miscellaneous argument types for COBOL and		SEARCH statement	420
Java	373	Serial search	421
MERGE statement	374	VARYING phrase	422
ASCENDING/DESCENDING KEY phrase	374	WHEN phrase (serial search)	423

Binary search	423	ADVANCING phrase rules	465
WHEN phrase (binary search).	424	LINAGE-COUNTER rules	465
Search statement considerations	426	END-OF-PAGE phrases	466
AT END and WHEN phrases	426	INVALID KEY phrases	466
NEXT SENTENCE.	426	END-WRITE phrase	467
END-SEARCH phrase	426	WRITE for sequential files	467
SET statement	427	WRITE for indexed files	468
Format 1: SET for basic table handling	427	WRITE for relative files	468
Format 2: SET for adjusting indexes	428	XML GENERATE statement	470
Format 3: SET for external switches	429	Nested XML GENERATE or XML PARSE	
Format 4: SET for condition-names	430	statements	475
Format 5: SET for USAGE IS POINTER data		Operation of XML GENERATE	476
items	430	Format conversion of elementary data	477
Format 6: SET for procedure-pointer and		Trimming of generated XML data	478
function-pointer data items.	431	XML element name and attribute name	
Format 7: SET for USAGE OBJECT REFERENCE		formation.	479
data items	433	XML PARSE statement	480
SORT statement	434	Nested XML GENERATE or XML PARSE	
ASCENDING KEY and DESCENDING KEY		statements	483
phrases	435	Control flow.	484
DUPLICATES phrase.	436		
COLLATING SEQUENCE phrase.	436		
USING phrase	437		
INPUT PROCEDURE phrase	437		
GIVING phrase.	438		
OUTPUT PROCEDURE phrase	439		
SORT special registers	439		
Segmentation considerations	440		
START statement	441		
KEY phrase	441		
INVALID KEY phrases	442		
END-START phrase	442		
Indexed files	442		
Relative files.	443		
STOP statement	444		
STRING statement.	445		
ON OVERFLOW phrases	447		
END-STRING phrase.	447		
Data flow	448		
SUBTRACT statement	450		
ROUNDED phrase	452		
SIZE ERROR phrases.	452		
CORRESPONDING phrase (format 3)	452		
END-SUBTRACT phrase	453		
UNSTRING statement	454		
DELIMITED BY phrase	456		
Delimiter with two or more characters	456		
Two or more delimiters	456		
INTO phrase	457		
POINTER phrase	457		
TALLYING IN phrase	457		
ON OVERFLOW phrases	457		
When an overflow condition occurs	458		
When an overflow condition does not occur	458		
END-UNSTRING phrase	458		
Data flow	458		
Values at the end of execution of the			
UNSTRING statement	460		
Example of the UNSTRING statement	460		
WRITE statement	462		
ADVANCING phrase.	465		

Chapter 20. Procedure division structure

The procedure division is an optional division.

Program procedure division

The program procedure division consists of optional declaratives, and procedures that contain sections, paragraphs, sentences, and statements.

Factory procedure division

The factory procedure division contains only factory method definitions.

Object procedure division

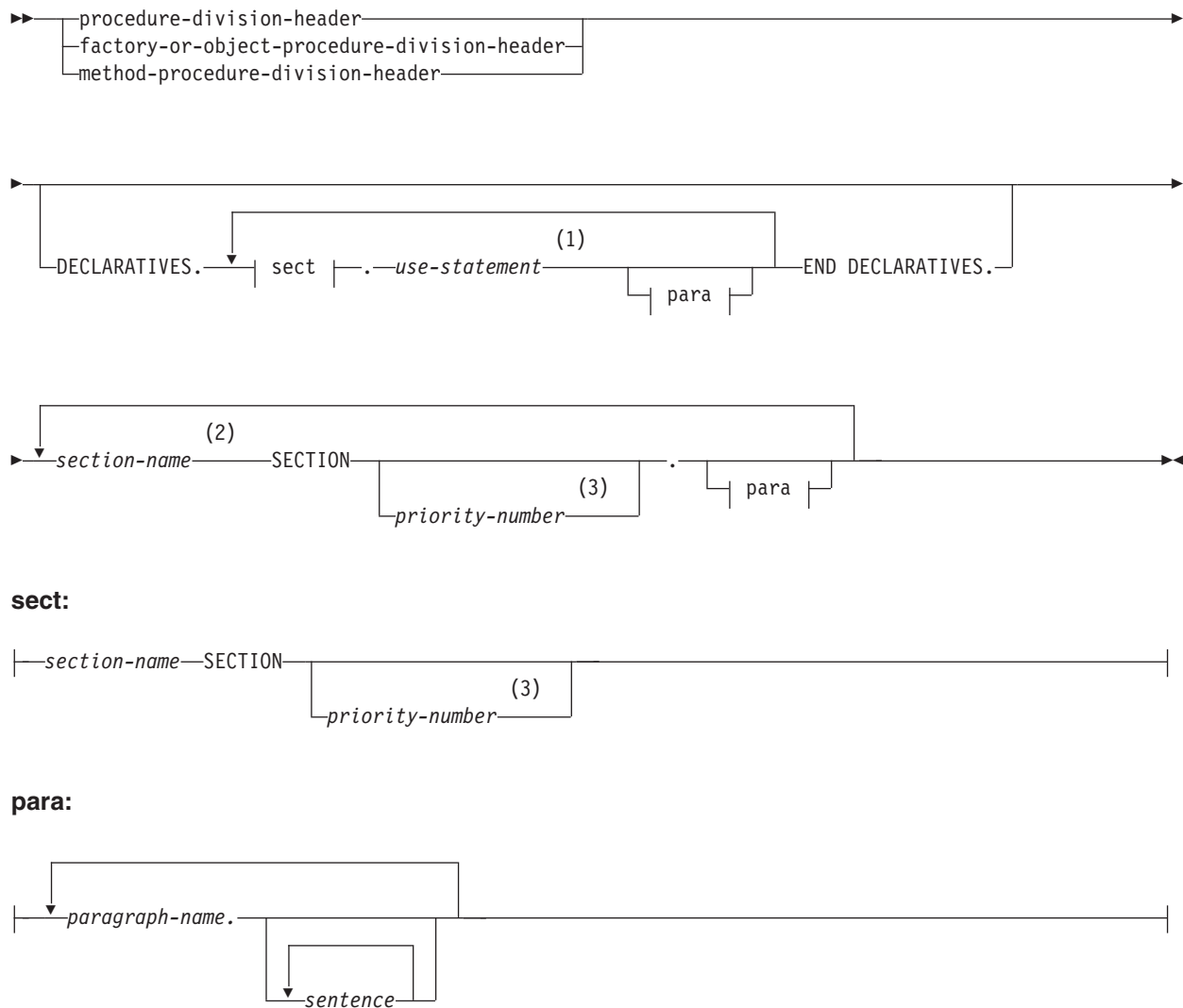
The object procedure division contains only object method definitions.

Method procedure division

A method procedure division consists of optional declaratives, and procedures that contain sections, paragraphs, sentences, and statements. A method can INVOKE other methods, be recursively invoked, and issue a CALL to a program. A method procedure division cannot contain nested programs or methods.

For additional details on a method procedure division, see “Requirements for a method procedure division” on page 250.

Format: procedure division



Notes:

- 1 The USE statement is described under "USE statement" on page 556.
- 2 *Section-name* can be omitted. If you omit *section-name*, *paragraph-name* can be omitted.
- 3 Priority-numbers are not valid for methods, recursive programs, or programs compiled with the THREAD option.

Requirements for a method procedure division

When using a method procedure division:

- You can use the EXIT METHOD statement or the GOBACK statement to return control to the invoking method or program. An implicit EXIT METHOD statement is generated as the last statement of every method procedure division. For details on the EXIT METHOD statement, see "EXIT METHOD statement" on page 346.

- You can use the STOP RUN statement (which terminates the run unit) in a method.
- You can use the RETURN-CODE special register within a method procedure division to access return codes from subprograms that are called with the CALL statement, but the RETURN-CODE value is not returned to the invoker of the current method. Use the procedure division RETURNING data name to return a value to the invoker of the current method. For details, see the discussion of RETURNING *data-name-2* under “The procedure division header.”

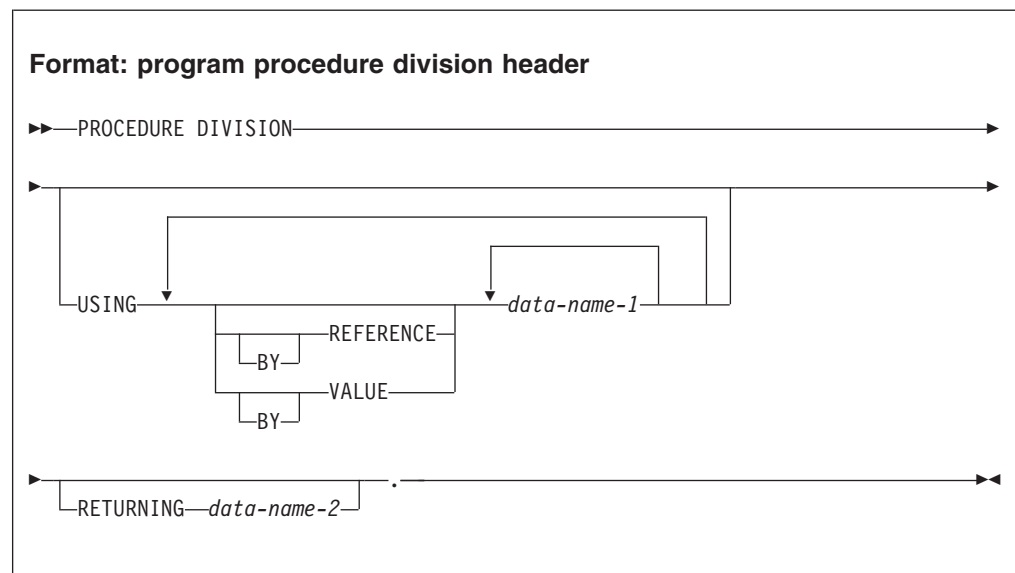
You cannot specify the following statements or clauses in a method procedure division:

- ALTER
- ENTRY
- EXIT PROGRAM
- GO TO without a specified procedure name
- SEGMENT-LIMIT
- USE FOR DEBUGGING

The procedure division header

The procedure division, if specified, is identified by one of the following headers, depending on whether you are specifying a program, a factory definition, an object definition, or a method definition.

The following is the format for a procedure division header in a program.



The following is the format for a procedure division header in a factory paragraph or object paragraph.

Format: factory and object procedure division header

►►—PROCEDURE DIVISION.—►►

The following is the format for a procedure division header in a method.

Format: method procedure division header

►►—PROCEDURE DIVISION—►►

—USING—►
—BY—
—VALUE—*data-name-1*—►

—RETURNING—*data-name-2*—►►

The USING phrase

The USING phrase specifies the parameters that a program or method receives when the program is called or the method is invoked.

The USING phrase is valid in the procedure division header of a called subprogram or invoked method entered at the beginning of the nondeclaratives portion. Each USING identifier must be defined as a level-01 or level-77 item in the linkage section of the called subprogram or invoked method.

In a called subprogram entered at the first executable statement following an ENTRY statement, the USING phrase is valid in the ENTRY statement. Each USING identifier must be defined as a level-01 or level-77 item in the linkage section of the called subprogram.

However, a data item specified in the USING phrase of the CALL statement can be a data item of any level in the data division of the calling COBOL program or method. A data item specified in the USING phrase of an INVOKE statement can be a data item of any level in the data division of the invoking COBOL program or method.

A data item in the USING phrase of the procedure division header can have a REDEFINES clause in its data description entry.

It is possible to call COBOL programs from non-COBOL programs or to pass user parameters from a system command to a COBOL main program. COBOL methods can be invoked only from Java or COBOL.

Command-line arguments are always passed in as native data types. If you specify the host data type compiler options CHAR(EBCDIC) or FLOAT(HEX), you must specify the NATIVE phrase in the description of arguments with data types that are affected by those compiler options.

The order of appearance of USING identifiers in both calling and called subprograms, or invoking methods or programs and invoked methods, determines the correspondence of single sets of data available to both. The correspondence is positional and not by name. For calling and called subprograms, corresponding identifiers must contain the same number of bytes although their data descriptions need not be the same.

For index-names, no correspondence is established. Index-names in calling and called programs, or invoking method or program and invoked methods, always refer to separate indexes.

The identifiers specified in a CALL USING or INVOKE USING statement name the data items available to the calling program or invoking method or program that can be referred to in the called program or invoked method. These items can be defined in any data division section.

A given identifier can appear more than once in a procedure division USING phrase. The last value passed to it by a CALL or INVOKE statement is used.

The BY REFERENCE or BY VALUE phrase applies to all parameters that follow until overridden by another BY REFERENCE or BY VALUE phrase.

BY REFERENCE *(for programs only)*

When an argument is passed BY CONTENT or BY REFERENCE, BY REFERENCE must be specified or implied for the corresponding formal parameter on the PROCEDURE or ENTRY USING phrase.

BY REFERENCE is the default if neither BY REFERENCE nor BY VALUE is specified.

If the reference to the corresponding data item in the CALL statement declares the parameter to be passed BY REFERENCE (explicit or implicit), the program executes as if each reference to a USING identifier in the called subprogram procedure division is replaced by a reference to the corresponding USING identifier in the calling program.

If the reference to the corresponding data item in the CALL statement declares the parameter to be passed BY CONTENT, the value of the item is moved when the CALL statement is executed and placed into a system-defined storage item that possesses the attributes declared in the linkage section for *data-name-1*. The data description of each parameter in the BY CONTENT phrase of the CALL statement must be the same, meaning no conversion or extension or truncation, as the data description of the corresponding parameter in the USING phrase of the procedure division header.

BY VALUE

When an argument is passed BY VALUE, the value of the argument is passed, not a reference to the sending data item. The receiving subprogram or method has access only to a temporary copy of the sending data item. Any modifications made to the formal parameters that correspond to an argument passed BY VALUE do not affect the argument.

Parameters specified in the USING phrase of a method procedure division header must be passed to the method BY VALUE. Method arguments are always passed as native data types. If you specify the host data type compiler option FLOAT(HEX), you must specify the NATIVE phrase in the description of arguments that are affected by those compiler options. See *Passing data in the COBOL for AIX Programming Guide* for examples that illustrate these concepts.

data-name-1

data-name-1 must be a level-01 or level-77 item in the linkage section.

When *data-name-1* is an object reference in a method procedure division header, an explicit class-name must be specified in the data description entry for that object reference; that is, *data-name-1* must not be a universal object reference.

For methods, the parameter data types are restricted to the data types that are interoperable between COBOL and Java, as listed in “Interoperable data types for COBOL and Java” on page 371.

RETURNING phrase

The RETURNING phrase specifies a data item that is to receive the program or method result.

data-name-2

data-name-2 is the RETURNING data item. *data-name-2* must be a level-01 or level-77 item in the linkage section.

In a method procedure division header, the data type of *data-name-2* must be one of the types supported for Java interoperation, as listed in “Interoperable data types for COBOL and Java” on page 371.

The RETURNING data item is an output-only parameter. On entry to the method, the initial state of the RETURNING data item has an undefined and unpredictable value. You must initialize the PROCEDURE DIVISION RETURNING data item before you reference its value. The value that is returned to the invoking routine is the value that the data item has at the point of exit from the method. See “RETURNING phrase” on page 369 for further details on conformance requirements for the INVOKE RETURNING identifier and the method RETURNING data item.

Do not use the procedure division RETURNING phrase in:

- Programs that contain the ENTRY statement.
- Nested programs.
- Main programs: Results of specifying procedure division RETURNING on a main program are undefined. You should specify the procedure division RETURNING phrase only on called subprograms. For main programs, use the RETURN-CODE special register to return a value to the operating environment.

References to items in the linkage section

Data items defined in the linkage section of the called program or invoked method can be referenced within the procedure division of that program if and only if they satisfy one of the following conditions:

- They are operands of the USING phrase of the procedure division header or the ENTRY statement.

- They are operands of SET ADDRESS OF, CALL ... BY REFERENCE ADDRESS OF, or INVOKE ... BY REFERENCE ADDRESS OF.
- They are defined with a REDEFINES or RENAMES clause, the object of which satisfies the above conditions.
- They are items subordinate to any item that satisfies the condition in the rules above.
- They are condition-names or index-names associated with data items that satisfy any of the above conditions.

Declaratives

Declaratives provide one or more special-purpose sections that are executed when an exceptional condition occurs.

When declarative sections are specified, they must be grouped at the beginning of the procedure division and the entire procedure division must be divided into sections.

Each declarative section starts with a USE statement that identifies the section's function. The series of procedures that follow specify the actions that are to be taken when the exceptional condition occurs. Each declarative section ends with another section-name followed by a USE statement, or with the keywords END DECLARATIVES.

The entire group of declarative sections is preceded by the keyword DECLARATIVES written on the line after the procedure division header. The group is followed by the keywords END DECLARATIVES. The keywords DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a separator period. No other text can appear on the same line.

In the declaratives part of the procedure division, each section header must be followed by a separator period, and must be followed by a USE statement followed by a separator period. No other text can appear on the same line.

The USE statement has three formats, discussed in these sections:

- "EXCEPTION/ERROR declarative" on page 556
- "DEBUGGING declarative" on page 558
- "LABEL declarative" on page 558

The USE statement itself is never executed; instead, the USE statement defines the conditions that execute the succeeding procedural paragraphs, which specify the actions to be taken. After the procedure is executed, control is returned to the routine that activated it.

A declarative procedure can be performed from a nondeclarative procedure.

You can include a statement that executes a previously called USE procedure that is still in control. However, to avoid an infinite loop, you must be sure there is an eventual exit at the bottom.

The declarative procedure is exited when the last statement in the procedure is executed.

Procedures

Within the procedure division, a *procedure* consists of:

- A *section* or a group of sections
- A *paragraph* or group of paragraphs

A *procedure-name* is a user-defined name that identifies a section or a paragraph.

Section

A *section-header* optionally followed by one or more paragraphs.

Section-header

A *section-name* followed by the keyword SECTION, optionally followed by a *priority-number*, followed by a separator period.

Section-headers are optional after the keywords END DECLARATIVES or if there are no declaratives.

Section-name

A user-defined word that identifies a section. A referenced section-name, because it cannot be qualified, must be unique within the program in which it is defined.

Priority-number

An integer or a positive signed numeric literal ranging in value from 0 through 99. *Priority-number* identifies a fixed segment or an independent segment that is to contain the section.

Sections in the declaratives portion must contain priority numbers in the range of 0 through 49.

You cannot specify priority-numbers:

- In a method definition
- In a program that is declared with the RECURSIVE attribute
- In a program compiled with the THREAD compiler option

A section ends immediately before the next section header, or at the end of the procedure division, or, in the declaratives portion, at the keywords END DECLARATIVES.

Segments

A segment consists of all sections in a program that have the same priority-number. Priority-number determines whether a section is stored in a fixed segment or an independent segment at run time.

Segments with a priority-number of 0 through 49 are fixed segments. Segments with a priority-number of 50 through 99 are independent segments.

The type of segment (fixed or independent) controls the segmentation feature.

In fixed segments, procedures are always in last-used state. In independent segments, procedures are in initial state each time the segment receives control from a segment with a different priority-number, except when the transfer of control results from the execution of a GOBACK or EXIT PROGRAM statement. Restrictions on the use of ALTER, SORT, and MERGE statements in independent segments are described under those statements.

COBOL for AIX does not support the overlay feature of the Standard COBOL 85 segmentation module.

Paragraph

A *paragraph-name* followed by a separator period, optionally followed by one or more sentences.

Paragraphs must be preceded by a period because paragraphs always follow either the identification division header, a section, or another paragraph, all of which must end with a period.

Paragraph-name

A user-defined word that identifies a paragraph. A *paragraph-name*, because it can be qualified, need not be unique.

If there are no declaratives (format 2), a *paragraph-name* is not required in the procedure division.

A paragraph ends immediately before the next *paragraph-name* or section header, or at the end of the procedure division, or, in the declaratives portion, at the keywords END DECLARATIVES.

Paragraphs need not all be contained within sections, even if one or more paragraphs are so contained.

Sentence

One or more *statements* terminated by a separator period.

Statement

A syntactically valid combination of *identifiers* and symbols (literals, relational-operators, and so forth) beginning with a COBOL verb.

Identifier

The word or words necessary to make unique reference to a data item, optionally including qualification, subscripting, indexing, and reference-modification. In any procedure division reference (except the class test), the contents of an identifier must be compatible with the class specified through its PICTURE clause, otherwise results are unpredictable.

Execution begins with the first statement in the procedure division, excluding declaratives. Statements are executed in the order in which they are presented for compilation, unless the statement rules dictate some other order of execution.

The end of the procedure division is indicated by one of the following:

- An identification division header that indicates the start of a nested source program
- An END PROGRAM, END METHOD, END FACTORY, or END OBJECT marker
- The physical end of a program; that is, the physical position in a source program after which no further source program lines occur

Arithmetic expressions

Arithmetic expressions are used as operands of certain conditional and arithmetic statements.

An arithmetic expression can consist of any of the following:

1. An identifier described as a numeric elementary item (including numeric functions)
2. A numeric literal

3. The figurative constant ZERO
4. Identifiers and literals, as defined in items 1, 2, and 3, separated by arithmetic operators
5. Two arithmetic expressions, as defined in items 1, 2, 3, or 4, separated by an arithmetic operator
6. An arithmetic expression, as defined in items 1, 2, 3, 4, or 5, enclosed in parentheses

Any arithmetic expression can be preceded by a unary operator.

Identifiers and literals that appear in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic can be performed.

If an exponential expression is evaluated as both a positive and a negative number, the result is always the positive number. For example, the square root of 4:

```
4 ** 0.5
```

is evaluated as +2 and -2. COBOL for AIX always returns +2.

If the value of an expression to be raised to a power is zero, the exponent must have a value greater than zero. Otherwise, the size error condition exists. In any case where no real number exists as the result of an evaluation, the size error condition exists.

Arithmetic operators

Five binary arithmetic operators and two unary arithmetic operators (as shown in Table 20) can be used in arithmetic expressions. These operators are represented by specific characters that must be preceded and followed by a space.

Table 20. Binary and unary operators

Binary operator	Meaning	Unary operator	Meaning
+	Addition	+	Multiplication by +1
-	Subtraction	-	Multiplication by -1
*	Multiplication		
/	Division		
**	Exponentiation		

Limitation: Exponents in fixed-point exponential expressions cannot contain more than nine digits. The compiler will truncate any exponent with more than nine digits. In the case of truncation, the compiler will issue a diagnostic message if the exponent is a literal or constant; if the exponent is a variable or data-name, a diagnostic is issued at run time.

Parentheses can be used in arithmetic expressions to specify the order in which elements are to be evaluated.

Expressions within parentheses are evaluated first. When expressions are contained within nested parentheses, evaluation proceeds from the least inclusive to the most inclusive set.

When parentheses are not used, or parenthesized expressions are at the same level of inclusiveness, the following hierarchic order is implied:

1. Unary operator
2. Exponentiation
3. Multiplication and division
4. Addition and subtraction

Parentheses either eliminate ambiguities in logic where consecutive operations appear at the same hierarchic level, or modify the normal hierarchic sequence of execution when this is necessary. When the order of consecutive operations at the same hierarchic level is not completely specified by parentheses, the order is from left to right.

An arithmetic expression can begin only with a left parenthesis, a unary operator, or an operand (that is, an identifier or a literal). It can end only with a right parenthesis or an operand. An arithmetic expression must contain at least one reference to an identifier or a literal.

There must be a one-to-one correspondence between left and right parentheses in an arithmetic expression, with each left parenthesis placed to the left of its corresponding right parenthesis.

If the first operator in an arithmetic expression is a unary operator, it must be immediately preceded by a left parenthesis if that arithmetic expression immediately follows an identifier or another arithmetic expression.

The following table shows permissible arithmetic symbol pairs. An arithmetic symbol pair is the combination of two such symbols in sequence. In the table:

Yes Indicates a permissible pairing.

No Indicates that the pairing is not permitted.

Table 21. Valid arithmetic symbol pairs

	Identifier or literal second symbol	* / ** + - second symbol	Unary + or unary - second symbol	(second symbol) second symbol
Identifier or literal first symbol	No	Yes	No	No	Yes
* / ** + - first symbol	Yes	No	Yes	Yes	No
Unary + or unary - first symbol	Yes	No	No	Yes	No
(first symbol	Yes	No	Yes	Yes	No
) first symbol	No	Yes	No	No	Yes

Arithmetic with date fields

Arithmetic operations that include a date field are restricted to:

- Adding a nondate to a date field
- Subtracting a nondate from a date field
- Subtracting a date field from a compatible date field

Date field operands are compatible if they have the same date format except for the year part, which can be windowed or expanded.

The following operations are not allowed:

- Any operation between incompatible dates
- Adding two date fields
- Subtracting a date field from a nondate
- Unary minus applied to a date field
- Division, exponentiation, or multiplication of or by a date field
- Arithmetic expressions that specify a year-last date field
- Arithmetic statements that specify a year-last date field, except as a receiving data item when the sending field is a nondate

The sections below describe the result of using date fields in the supported addition and subtraction operations.

For more information about using date fields in arithmetic operations, see:

- “ADD statement” on page 308
- “COMPUTE statement” on page 328
- “SUBTRACT statement” on page 450

Usage notes

- Arithmetic operations treat date fields as numeric items; they do not recognize any date-specific internal structure. For example, adding 1 to a windowed date field that contains the value 991231 (which might be used in an application to represent December 31, 1999) results in the value 991232, not 000101.
- When used as operands in arithmetic expressions or arithmetic statements, windowed date fields are automatically expanded according to the century window specified by the YEARWINDOW compiler option. For details of windowed expansion, see “Semantics of windowed date fields” on page 185.

Addition that involves date fields

The following table shows the result of using a date field with a compatible operand in an addition.

Table 22. Results of using date fields in addition

	Nondate second operand	Date field second operand
Nondate first operand	Nondate	Date field
Date field first operand	Date field	Not allowed

For details on how a result is stored in a receiving field, see “Storing arithmetic results that involve date fields” on page 261.

Subtraction that involves date fields

The following table shows the result of using a date field with a compatible operand in the subtraction:

first operand - second operand

In a SUBTRACT statement, these operands appear in the reverse order:

SUBTRACT second operand FROM first operand

Table 23. Results of using date fields in subtraction

	Nondate second operand	Date field second operand
Nondate first operand	Nondate	Not allowed
Date field first operand	Date field	Nondate

Storing arithmetic results that involve date fields

The following statements perform arithmetic, then store the result, or sending field, into one or more receiving fields:

- ADD
- COMPUTE
- DIVIDE
- MULTIPLY
- SUBTRACT

In a MULTIPLY statement, only GIVING identifiers can be date fields. In a DIVIDE statement, only GIVING identifiers or the REMAINDER identifier can be date fields.

Any windowed date fields that are operands of the arithmetic expression or statement are treated as if they were expanded before use, as described under “Semantics of windowed date fields” on page 185.

If the sending field is a date field, then the receiving field must be a compatible date field. That is, both fields must have the same date format, except for the year part, which can be windowed or expanded.

If the ON SIZE ERROR clause is not specified on the statement, the store operation follows the existing COBOL rules for the statement and proceeds as if the receiving and sending fields (after any automatic expansion of windowed date field operands or result) were both nondates.

Table 24 on page 262 shows how these statements store the value of a sending field in a receiving field, where either field can be a date field. The section uses the following terms to describe how the store is performed:

Nonwindowed

The statement performs the store with no special date-sensitive size error processing, as described under “SIZE ERROR phrases” on page 293.

Windowed with nondate sending field

The nondate sending field is treated as a windowed date field compatible with the windowed date receiving field, but with the year part

representing the number of years since 1900. (This representation is similar to a windowed date field with a base year of 1900, except that the year part is not limited to a positive number of at most two digits.) The store proceeds as if this assumed year part of the sending field were expanded by adding 1900 to it.

Windowed with date sending field

The store proceeds as if all windowed date field operands had been expanded as necessary, so that the sending field is a compatible expanded date field.

Size error processing: For both kinds of sending field, if the assumed or actual year part of the sending field falls within the century window, the sending field is stored in the receiving field after removing the century component of the year part. That is, the low-order or rightmost two digits of the expanded year part are retained and the high-order or leftmost two digits are discarded.

If the year part does not fall within the century window, then the receiving field is unmodified, and the size error imperative statement is executed when any remaining arithmetic operations are complete.

For example:

```
77 DUE-DATE PICTURE 9(5) DATE FORMAT YYXX.  
77 IN-DATE PICTURE 9(8) DATE FORMAT YYYYXX VALUE 1995001.  
...  
  COMPUTE DUE-DATE = IN-DATE + 10000  
    ON SIZE ERROR imperative-statement  
  END-COMPUTE
```

The sending field is an expanded date field representing January 1, 2005. Assuming that 2005 falls within the century window, the value stored in DUE-DATE is 05001; that is, the sending value of 2005001 without the century component 20.

Table 24. Storing arithmetic results that involve date fields when ON SIZE ERROR is specified

	Nondate sending field	Date field sending field
Nondate receiving field	Nonwindowed	Not allowed
Windowed date field receiving field	Windowed	Windowed
Expanded date field receiving field	Nonwindowed	Nonwindowed

Conditional expressions

A conditional expression causes the object program to select alternative paths of control, depending on the truth value of a test. Conditional expressions are specified in EVALUATE, IF, PERFORM, and SEARCH statements.

A conditional expression can be specified in either simple conditions or complex conditions. Both simple and complex conditions can be enclosed within any number of paired parentheses; the parentheses do not change whether the condition is simple or complex.

Simple conditions

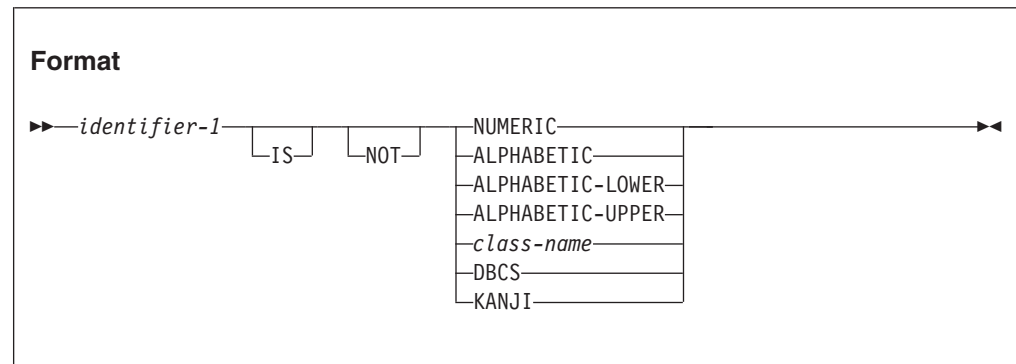
There are five simple conditions:

- Class condition
- Condition-name condition
- Relation condition
- Sign condition
- Switch-status condition

A simple condition has a truth value of either true or false.

Class condition

The class condition determines whether the content of a data item is alphabetic, alphabetic-lower, alphabetic-upper, numeric, DBCS, KANJI, or contains only the characters in the set of characters specified by the CLASS clause as defined in the SPECIAL-NAMES paragraph of the environment division.



identifier-1

Must reference a data item described with one of the following usages:

- DISPLAY, NATIONAL, COMPUTATIONAL-3, or PACKED-DECIMAL when NUMERIC is specified
- DISPLAY-1 when DBCS or KANJI is specified
- DISPLAY or NATIONAL when ALPHABETIC, ALPHABETIC-UPPER, or ALPHABETIC-LOWER is specified
- DISPLAY when class-name is specified

Must not be of class alphabetic when NUMERIC is specified.

Must not be of class numeric when ALPHABETIC, ALPHABETIC-UPPER, or ALPHABETIC-LOWER is specified.

Table 25 on page 265 lists the forms of class condition that are valid for each type of identifier.

If *identifier-1* is a function-identifier, it must reference an alphanumeric or national function.

An alphanumeric group item can be used in a class condition where an elementary alphanumeric item can be used, *except* that the NUMERIC class condition cannot be used if the group contains one or more signed elementary items.

When *identifier-1* is described with usage NATIONAL, the class-condition tests for the national character representation of the characters associated with the specified character class. For example, specifying a class condition of the form IF national-item IS ALPHABETIC is a test for the lowercase and uppercase letters Latin capital letter A through Latin capital letter Z and the space, as represented in national characters. Specifying IF national-item IS NUMERIC is a test for the characters 0 through 9.

NOT When used, NOT and the next keyword define the class test to be executed for truth value. For example, NOT NUMERIC is a truth test for determining that the result of a NUMERIC class test is false (in other words, the item contains data that is nonnumeric).

NUMERIC

identifier-1 consists entirely of the characters 0 through 9, with or without an operational sign.

If its PICTURE does not contain an operational sign, the identifier being tested is determined to be numeric only if the contents are numeric and an operational sign is not present.

If its PICTURE does contain an operational sign, the identifier being tested is determined to be numeric only if the item is an elementary item, the contents are numeric, and a valid operational sign is present.

ALPHABETIC

identifier-1 consists entirely of any combination of the lowercase or uppercase Latin alphabetic characters A through Z and the space.

ALPHABETIC-LOWER

identifier-1 consists entirely of any combination of the lowercase Latin alphabetic characters a through z and the space.

ALPHABETIC-UPPER

identifier-1 consists entirely of any combination of the uppercase Latin alphabetic characters A through Z and the space.

class-name

identifier-1 consists entirely of the characters listed in the definition of class-name in the SPECIAL-NAMES paragraph.

DBCS *identifier-1* consists entirely of DBCS characters. *identifier-1* contains DBCS characters that correspond to valid EBCDIC DBCS characters.

A range check is performed on the item for valid character representation. The valid range is X'41' through X'FE' for both bytes of each DBCS character and X'4040' for the DBCS blank. (These ranges are for the equivalent DBCS character representation for Enterprise COBOL for z/OS, not the actual DBCS character value ranges of the workstation DBCS characters.)

KANJI

identifier-1 contains DBCS characters that correspond to valid EBCDIC DBCS characters.

A range check is performed on the item for valid character representation. The valid range is from X'41' through X'7E' for the first byte, from X'41' through X'FE' for the second byte, and X'4040' for the DBCS blank. (These ranges are for the equivalent DBCS character representation for Enterprise COBOL for z/OS, not the actual DBCS character value ranges of the workstation DBCS characters.)

Table 25. Valid forms of the class condition for different types of data items

Type of data item referenced by <i>identifier-1</i>	Valid forms of the class condition	
Alphabetic	ALPHABETIC ALPHABETIC-LOWER ALPHABETIC-UPPER <i>class-name</i>	NOT ALPHABETIC NOT ALPHABETIC-LOWER NOT ALPHABETIC-UPPER NOT <i>class-name</i>
Alphanumeric, alphanumeric-edited, or numeric-edited	ALPHABETIC ALPHABETIC-LOWER ALPHABETIC-UPPER NUMERIC <i>class-name</i>	NOT ALPHABETIC NOT ALPHABETIC-LOWER NOT ALPHABETIC-UPPER NOT NUMERIC NOT <i>class-name</i>
External-decimal or internal-decimal	NUMERIC	NOT NUMERIC
DBCS	DBCS KANJI	NOT DBCS NOT KANJI
National	NUMERIC ALPHABETIC ALPHABETIC-LOWER ALPHABETIC-UPPER	NOT NUMERIC NOT ALPHABETIC NOT ALPHABETIC-LOWER NOT ALPHABETIC-UPPER
Numeric	NUMERIC <i>class-name</i>	NOT NUMERIC NOT <i>class-name</i>

Condition-name condition

A condition-name condition tests a conditional variable to determine whether its value is equal to any values that are associated with the condition-name.

Format

►► *condition-name-1* ◀◀

A condition-name is used in conditions as an abbreviation for the relation condition. The rules for comparing a conditional variable with a condition-name value are the same as those specified for relation conditions.

If *condition-name-1* has been associated with a range of values (or with several ranges of values), the conditional variable is tested to determine whether its value falls within the ranges, including the end values. The result of the test is true if one of the values that corresponds to the condition-name equals the value of its associated conditional variable.

Condition-names are allowed for alphanumeric, DBCS, national, and floating-point data items, as well as others, as defined for the condition-name format of the VALUE clause.

The following example illustrates the use of conditional variables and condition-names:

```

01 AGE-GROUP      PIC 99.
   88 INFANT      VALUE 0.
   88 BABY        VALUE 1, 2.
   88 CHILD       VALUE 3 THRU 12.
   88 TEENAGER    VALUE 13 THRU 19.

```

AGE-GROUP is the conditional variable; INFANT, BABY, CHILD, and TEENAGER are condition-names. For individual records in the file, only one of the values specified in the condition-name entries can be present.

The following IF statements can be added to the above example to determine the age group of a specific record:

```

IF INFANT...      (Tests for value 0)
IF BABY...        (Tests for values 1, 2)
IF CHILD...       (Tests for values 3 through 12)
IF TEENAGER...    (Tests for values 13 through 19)

```

Depending on the evaluation of the condition-name condition, alternative paths of execution are taken by the object program.

Condition-name conditions and windowed date field comparisons

If the conditional variable is a windowed date field, the values associated with its condition-names are treated like values of the windowed date field. That is, they are treated as if they were converted to expanded date format, as described under “Semantics of windowed date fields” on page 185.

For example, given YEARWINDOW(1945), a century window of 1945–2044, and the following definition:

```

05 DATE-FIELD    PIC 9(6) DATE FORMAT YYXXXX.
   88 DATE-TARGET VALUE 051220.

```

a value of 051220 in DATE-FIELD would cause the following condition to be true:

```
IF DATE-TARGET...
```

because the value associated with DATE-TARGET and the value of DATE-FIELD would both be treated as if they were prefixed by "20" before comparison.

However, the following condition would be false:

```
IF DATE-FIELD = 051220...
```

because in a comparison with a windowed date field, literals are treated as if they were prefixed by "19" regardless of the century window. So the above condition effectively becomes:

```
IF 20051220 = 19051220...
```

For more information about using windowed date fields in conditional expressions, see “Comparison of date fields” on page 276.

Relation conditions

A relation condition specifies the comparison of two operands. The relational operator that joins the two operands specifies the type of comparison. The relation condition is true if the specified relation exists between the two operands; the relation condition is false if the specified relation does not exist.

Comparisons are defined for the following:

- Two operands of class alphabetic
- Two operands of class alphanumeric
- Two operands of class DBCS
- Two operands of class national
- Two operands of class numeric
- Two operands of different classes where each operand is one of the classes alphabetic, alphanumeric, or national
- Two operands where one is a numeric integer and the other is class alphanumeric or national
- Two operands where one is class DBCS and the other is class national
- Comparisons involving indexes or index data items
- Two data pointer operands
- Two procedure pointer operands
- Two function pointer operands
- Two object reference operands
- An alphanumeric group and any operand that has usage DISPLAY, DISPLAY-1, or NATIONAL

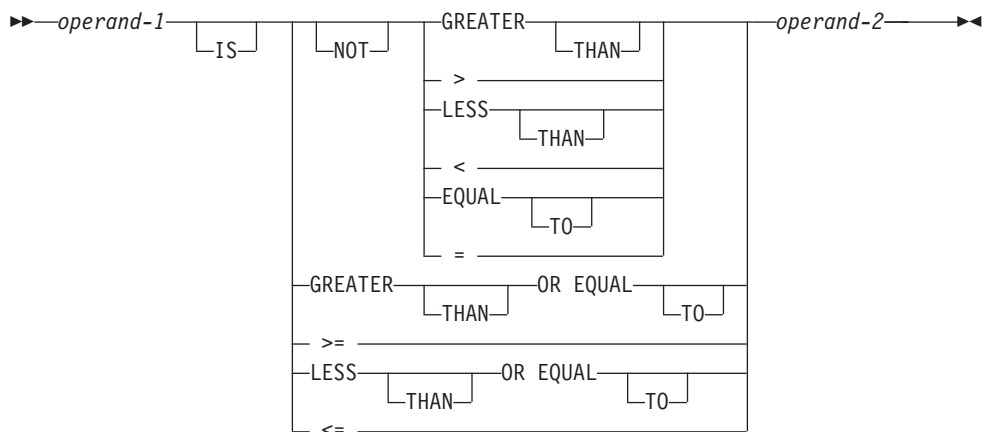
The following relation condition formats are defined:

- A general relation condition, for comparisons that involve only data items, literals, index-names, or index data items. For details, see “General relation conditions.”
- A data pointer relation condition. For details, see “Data pointer relation conditions” on page 277.
- A program pointer relation condition, for comparison of procedure pointers or function pointers. For details, see “Procedure-pointer and function-pointer relation conditions” on page 278.
- An object-reference relation condition. For details, see “Object-reference relation conditions” on page 278.

General relation conditions

A general relation condition compares two operands, either of which can be an identifier, literal, arithmetic expression, or index-name.

Format 1: general relation condition



operand-1

The subject of the relation condition. Can be an identifier, literal, function-identifier, arithmetic expression, or index-name.

operand-2

The object of the relation condition. Can be an identifier, literal, function-identifier, arithmetic expression, or index-name.

An alphanumeric literal can be enclosed in parentheses within a relation condition.

The relation condition must contain at least one reference to an identifier.

The relational operators, shown in Table 26, specify the type of comparison to be made. Each relational operator must be preceded and followed by a space. The two characters of the relational operators `>=` and `<=` must not have a space between them.

Table 26. Relational operators and their meanings

Relational operator	Can be written	Meaning
IS GREATER THAN	IS >	Greater than
IS NOT GREATER THAN	IS NOT >	Not greater than
IS LESS THAN	IS <	Less than
IS NOT LESS THAN	IS NOT <	Not less than
IS EQUAL TO	IS =	Equal to
IS NOT EQUAL TO	IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	IS >=	Is greater than or equal to
IS LESS THAN OR EQUAL TO	IS <=	Is less than or equal to

In a general relation condition, data items, literals, and figurative constants of class alphabetic, alphanumeric, DBCS, national, and numeric are compared using the following comparison types:

Comparison type	Meaning
Alphanumeric	Comparison of the alphanumeric character value of two operands
DBCS	Comparison of the DBCS character value of two operands
National	Comparison of the national character value of two operands
Numeric	Comparison of the algebraic value of two operands
Group	Comparison of the alphanumeric character value of two operands, where one or both operands is an alphanumeric group item

Table 27 on page 270 and Table 28 on page 271 show the permissible pairs for comparisons with different types of operands. The comparison type is indicated at the row and column intersection for permitted comparisons, using the following key:

- Alph** Comparison of alphanumeric characters (further described in “Alphanumeric comparisons” on page 271)
- DBCS** Comparison of DBCS characters (further described in “DBCS comparisons” on page 273)
- Nat** Comparison of national characters (further described in “National comparisons” on page 273)
- Num** Comparison of algebraic value (further described in “Numeric comparisons” on page 274)
- Group** Comparison of alphanumeric characters involving an alphanumeric group (further described in “Group comparisons” on page 275)
- (Int)** Integer items only (combined with comparison type Alph, Nat, Num, or Group)
- Blank** Comparison is not allowed

For rules and restrictions for comparisons involving year-last date fields, see “Comparison of date fields” on page 276.

For rules and restrictions for comparisons involving index-names and index data items, see “Comparison of index-names and index data items” on page 275.

Introduction to Table 27 on page 270: This table is organized in the following manner:

- In the first column, under “Type of data item or literal”, each row identifies a type of operand. In some cases, the type of operand references a grouping of operands that have common properties for comparison. For example, the row for “Alphanumeric character items” references all the types of operands that are listed in the cell, as follows:
 - Data items of category:
 - Alphanumeric
 - Alphanumeric- edited
 - Numeric-edited with usage DISPLAY
 - Alphanumeric functions
- Subsequent column headings refer to the a type of operand or a grouping of operands. For example, the column heading “Alphabetic and alphanumeric

character items" refers to the types of operands identified as "Alphabetic data items" and all the types of operands that are grouped under the operand titled "Alphanumeric character items".

- Literals are listed as a type of operand only in the first column. They do not appear as column headings because literals cannot be used as both operands of a relation condition.

Table 27. Comparisons involving data items and literals

Type of data item or literal	Alpha-numeric group items	Alphabetic and alphanumeric character items	Zoned decimal items	Native numeric items	Alpha-numeric floating-point items	National character items	National decimal items	National floating-point items	DBCS items
Alphanumeric group item	Group	Group	Group (Int)		Group	Group	Group (Int)	Group	Group
Alphabetic data items	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	
Alphanumeric character items: <ul style="list-style-type: none"> • Data items of category: <ul style="list-style-type: none"> – Alphanumeric – Alphanumeric- edited – Numeric-edited with usage DISPLAY • Alphanumeric functions 	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	
Alphanumeric literals	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	
Numeric literals	Group (Int)	Alph (Int)	Num	Num	Num	Nat (Int)	Num	Num	
Zoned decimal data items	Group (Int)	Alph (Int)	Num	Num	Num	Nat (Int)	Num	Num	
Native numeric items: <ul style="list-style-type: none"> • Binary • Arithmetic expression • Internal decimal • Internal floating-point Numeric and integer intrinsic functions			Num	Num	Num		Num	Num	
Display floating-point items	Group	Alph	Num	Num	Num	Nat	Num	Num	
Floating-point literals			Num	Num	Num		Num	Num	
National character items: <ul style="list-style-type: none"> • Data items of category: <ul style="list-style-type: none"> – National – National- edited – Numeric- edited with usage NATIONAL • National intrinsic functions • National groups (treated as elementary item) 	Group	Nat	Nat (Int)		Nat	Nat	Nat (Int)	Nat	Nat
National literals	Group	Nat	Nat (Int)		Nat	Nat	Nat (Int)	Nat	Nat
National decimal items	Group (Int)	Alph (Int)	Num	Num	Num	Nat (Int)	Num	Num	

Table 27. Comparisons involving data items and literals (continued)

Type of data item or literal	Alpha-numeric group items	Alpha-betic and alpha-numeric character items	Zoned decimal items	Native numeric items	Alpha-numeric floating-point items	National character items	National decimal items	National floating-point items	DBCS items
National floating-point items	Group	Nat	Num	Num	Num	Nat	Num	Num	
DBCS data items	Group					Nat			DBCS
DBCS literals	Group					Nat			DBCS

Table 28. Comparisons involving figurative constants

Figurative constant	Alpha-numeric group items	Alpha-betic and alpha-numeric character items	Zoned decimal items	Native numeric items	Alpha-numeric floating point items	National character items	National decimal items	National floating point items	DBCS items
ZERO	Group	Alph	Num	Num	Num	Nat	Num	Num	
SPACE	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	DBCS
HIGH-VALUE, LOW-VALUE QUOTE	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	
Symbolic character	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	
ALL alphanumeric literal	Group	Alph	Alph (Int)		Alph	Nat	Alph (Int)	Nat	
ALL national literal	Group	Nat	Nat (Int)		Nat	Nat	Nat (Int)	Nat	Nat
ALL DBCS literal	Group					Nat			DBCS

Alphanumeric comparisons

An alphanumeric comparison is a comparison of the single-byte character values of two operands.

When one of the operands is neither class alphanumeric nor class alphabetic, that operand is processed as follows:

- A display floating-point data item is treated as though it were a data item of category alphanumeric, rather than as a numeric value.
- A zoned decimal integer operand is treated as though it were moved to a temporary elementary data item of category alphanumeric with a length the same as the number of digits in the integer, according to the rules of the MOVE statement.

When the ZWB compiler option is in effect, the unsigned value of the integer operand is moved to the temporary data item. When the NOZWB compiler option is specified, the signed value is moved to the temporary data item. See ZWB in the *COBOL for AIX Programming Guide* for more details about the ZWB (NOZWB) compiler option.

Comparison then proceeds with the temporary data item of category alphanumeric.

Comparison of two alphanumeric operands:

The collating sequence used for comparison is determined by the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph and the setting of the COLLSEQ compiler option.

If the PROGRAM COLLATING SEQUENCE clause is specified with an alphabet-name of STANDARD-1, STANDARD-2, or EBCDIC, the COLLSEQ compiler option is ignored. The collating sequence is the one associated with the specified alphabet-name. The comparison proceeds as described below for “Standard comparison.”

If the PROGRAM COLLATING SEQUENCE clause is not specified or is specified with an alphabet-name of NATIVE, the method of comparison is determined by the COLLSEQ compiler option:

- If the COLLSEQ(BINARY) compiler option is in effect, the collating sequence is determined by the binary values of characters. The comparison proceeds as described below for “Standard comparison.”
- If the COLLSEQ(EBCDIC) compiler option is in effect, the EBCDIC collating sequence is used. The comparison proceeds as described below for “Standard comparison.”
- If the COLLSEQ(LOCALE) compiler option is in effect, the collating sequence is determined by the runtime locale. For purposes of comparison, trailing spaces are truncated from the operands except that an operand consisting of all spaces is truncated to a single space. Locale-based comparison is not necessarily a character-by-character comparison. If the shorter operand were extended with spaces as for a nonlocale-based comparison, the result might not be the culturally-expected result. For information about locales, see Appendix G, “Locale considerations,” on page 609.

If the PROGRAM COLLATING SEQUENCE clause is specified with an alphabet-name that references a collating sequence defined by literals, the collating sequence is determined by the order of the specified literals and the sequence indicated by the COLLSEQ compiler option, as described for the “ALPHABET clause” on page 113. The comparison proceeds as described below for “Standard comparison.”

Standard comparison:

A standard comparison is any comparison that is not based on a locale. The standard comparison method depends on whether the operands to be compared are of equal length or unequal length.

If the operands are of unequal length, the comparison proceeds as though the shorter operand were padded on the right with appropriate space characters to make the operands of equal length. The comparison then proceeds according to the rules for the comparison of operands of equal length.

If the operands are of equal length, the comparison proceeds by comparing corresponding character positions in the two operands, starting from the leftmost position, until either unequal characters are encountered or the rightmost character position is reached, whichever comes first. The operands are determined to be equal if all corresponding characters are equal.

The first-encountered unequal character in the operands is compared to determine the relation of the operands. The operand that contains the character with the higher collating value is the greater operand.

DBCS comparisons

A DBCS comparison is a comparison of two DBCS operands; the following rules apply:

- Comparisons of DBCS operands are based on a collating sequence specified by the COLLSEQ compiler option:
 - If the COLLSEQ(LOCALE) compiler option is in effect, the collating sequence is determined by the runtime locale. For information about the locale, see Appendix G, “Locale considerations,” on page 609.
 - Otherwise, the collating sequence is determined by the binary values of the DBCS characters.

National comparisons

A national comparison is a comparison of the national character value of two operands of class national.

When the relation condition specifies an operand that is not class national, that operand is converted to a data item of category national before the comparison. The following list describes the conversion of operands to category national.

DBCS A DBCS operand is treated as though it were moved to a temporary data item of category national of the same length as the DBCS operand. DBCS characters are converted to the corresponding national characters. The source code page used for the conversion depends on whether the DBCS operand is an EBCDIC or ASCII data item. If the DBCS operand is an EBCDIC data item (that is, the compiler option CHAR(EBCDIC) is in effect and the NATIVE phrase is not specified for the operand), the EBCDIC code page in effect for the operand is used; otherwise, the code page indicated by the locale in effect for the operand is used. For details, see Appendix G, “Locale considerations,” on page 609.

Alphabetic, alphanumeric, alphanumeric-edited, and numeric-edited with usage DISPLAY

The operand is treated as though it were moved to a temporary data item of category national of the length needed to represent the number of character positions in that operand. Alphanumeric characters are converted to the corresponding national characters. The source code page used for the conversion depends on whether the alphanumeric operand is an EBCDIC or ASCII data item. If the alphanumeric operand is an EBCDIC data item (that is, the compiler option CHAR(EBCDIC) is in effect and the NATIVE phrase is not specified for the operand), the EBCDIC code page in effect for the operand is used; otherwise, the code page indicated by the locale in effect for the operand is used. For details, see Appendix G, “Locale considerations,” on page 609.

Numeric integer

A numeric integer operand is treated as though it were moved to a temporary data item of category alphanumeric of a length the same as the number of digits in the integer. The unsigned value is used. The resulting temporary data item is then converted as an alphanumeric operand.

External floating-point

A display floating-point item is treated as though it were a data item of category alphanumeric, rather than as a numeric value, and then converted as an alphanumeric operand.

A national floating-point item is treated as though it were a data item of category national, rather than as a numeric value.

The implicit moves for the conversions are carried out in accordance with the rules of the MOVE statement.

The resulting category national data item is used in the comparison of two national operands.

Comparison of two national operands:

The method used for comparison is determined by the setting of the NCOLLSEQ compiler option.

If the NCOLLSEQ(BINARY) compiler option is in effect, the collating sequence is determined by the binary values of national characters. The comparison proceeds as follows:

- If the operands are of unequal length, the comparison proceeds as though the shorter operand were padded on the right with the default national space character (NX'0020') to make the operands of equal length. The comparison then proceeds according to the rules for the comparison of operands of equal length.
- If the operands are of equal length, the comparison proceeds by comparing corresponding national character positions in the two operands, starting from the leftmost position, until either unequal national characters are encountered or the rightmost national character position is reached, whichever comes first. The operands are determined to be equal if all corresponding national characters are equal.
- The first-encountered unequal national character in the operands is compared to determine the relation of the operands. The operand that contains the national character with the higher collating value is the greater operand.

If the NCOLLSEQ(LOCALE) compiler option is in effect, the collating sequence is determined by the runtime locale. For purposes of comparison, trailing spaces are truncated from the operands except that an operand consisting of all spaces is truncated to a single space. Locale-based comparison is not necessarily a character-by-character comparison. If the shorter operand were extended with spaces as for a nonlocale-based comparison, the result might not be the culturally-expected result. For information about locales, see Appendix G, "Locale considerations," on page 609.

The PROGRAM COLLATING SEQUENCE clause has no effect on comparisons of national operands.

Numeric comparisons

A numeric comparison is a comparison of the algebraic value of two operands of class numeric.

When the algebraic values of numeric operands are compared:

- The length (number of digits) of the operands is not significant.
- The usage of the operands is not significant.
- Unsigned numeric operands are considered positive.
- All zero values compare equal; the presence or absence of a sign does not affect the result.

Group comparisons

A group comparison is a comparison of the alphanumeric character values of two operands.

For the comparison operation, each operand is treated as though it were an elementary data item of category alphanumeric of the same size as the operand, in bytes. The comparison then proceeds as for two elementary operands of category alphanumeric, as described in “Alphanumeric comparisons” on page 271.

Usage note: There is no conversion of data for group comparisons. The comparison operates on bytes of data without regard to data representation. The result of comparing an elementary item or literal operand to an alphanumeric group item is predictable when that operand and the content of the group item have the same data representation.

Comparison of index-names and index data items

Comparisons involving index-names, index data items, or both conform to the following rules:

- The comparison of two index-names is actually the comparison of the corresponding occurrence numbers.
- In the comparison of an index-name with a data item (other than an index data item), or in the comparison of an index-name with a literal, the occurrence number that corresponds to the value of the index-name is compared with the data item or literal.
- In the comparison of an index-name with an arithmetic expression, the occurrence number that corresponds to the value of the index-name is compared with the arithmetic expression.

Because an integer function can be used wherever an arithmetic expression can be used, you can compare an index-name to an integer or numeric function.

- In the comparison of an index data item with an index-name or another index data item, the actual values are compared without conversion. Results of any other comparison involving an index data item are undefined.

Valid comparisons for index-names and index data items are shown in the following table.

Table 29. Comparisons for index-names and index data items

Operands compared	Index-name	Index data item	Data-name (numeric integer only)	Literal (numeric integer only)	Arithmetic Expression
Index-name	Compare occurrence number	Compare without conversion	Compare occurrence number with content of referenced data item	Compare occurrence number with literal	Compare occurrence number with arithmetic expression
Index data item	Compare without conversion	Compare without conversion	Invalid	Invalid	Invalid

Comparison of date fields

Date fields can be alphanumeric category, zoned decimal, or internal decimal; the existing rules for the validity and comparison type (numeric or alphanumeric) apply. For example, an alphanumeric date field cannot be compared with an internal decimal date field. In addition to these rules, two date fields can be compared only if they are compatible; they must have the same date format except for the year part, which can be windowed or expanded.

For year-last date fields, the only comparisons that are supported are IS EQUAL TO and IS NOT EQUAL TO between two year-last date fields with identical date formats, or between a year-last date field and a nondate.

Table 30 shows supported comparisons for nonyear-last date fields. This table uses the following terms to describe how the comparisons are performed:

Nonwindowed

The comparison is performed with no windowing, as if the operands were both nondates.

Windowed

The comparison is performed as if:

1. Any windowed date field in the relation were expanded according to the century window specified by the YEARWINDOW compiler option, as described under “Semantics of windowed date fields” on page 185.
2. Any repetitive alphanumeric figurative constant were expanded to the size of the windowed date field with which it is compared, giving an alphanumeric nondate comparand. Repetitive alphanumeric figurative constants include ZERO (in an alphanumeric context), SPACE, LOW-VALUE, HIGH-VALUE, QUOTE and ALL *literal*.
3. Any nondate operands were treated as if they had the same date format as the date field, but with a base year of 1900.

The comparison is then performed according to normal COBOL rules. Alphanumeric comparisons are not changed to numeric comparisons by the prefixing of the century value.

Table 30. Comparisons with date fields

	Nondate second operand	Windowed date field second operand	Expanded date field second operand
Nondate first operand	Nonwindowed	Windowed ¹	Nonwindowed
Windowed date field first operand	Windowed ¹	Windowed	Windowed
Expanded date field first operand	Nonwindowed	Windowed	Nonwindowed
1. When compared with windowed date fields, nondates are assumed to contain a windowed year relative to 1900. For details, see item 3 under the definition of "Windowed" comparison.			

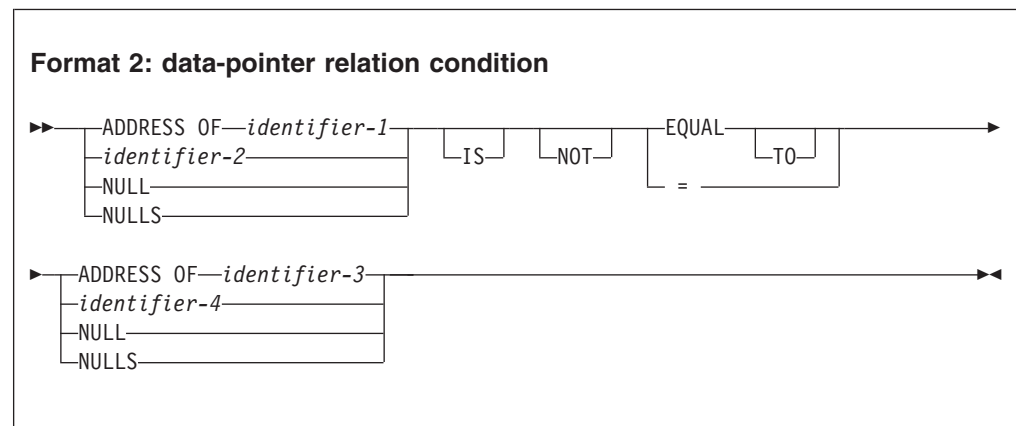
Relation conditions can contain arithmetic expressions. For information about the treatment of date fields in arithmetic expressions, see “Arithmetic with date fields” on page 260.

Data pointer relation conditions

Only EQUAL and NOT EQUAL are allowed as relational operators when specifying pointer data items. Pointer data items are items defined explicitly as USAGE POINTER, or are ADDRESS OF special registers, which are implicitly defined as USAGE POINTER.

The operands are equal if the two addresses used in the comparison would both result in the same storage location.

This relation condition is allowed in IF, PERFORM, EVALUATE, and SEARCH format-1 statements. It is not allowed in SEARCH format-2 (SEARCH ALL) statements because there is no meaningful ordering that can be applied to pointer data items.



identifier-1, identifier-3

Can specify any level item defined in the linkage section, except 66 and 88.

identifier-2, identifier-4

Must be described as USAGE POINTER.

NULL, NULLS

Can be used only if the other operand is defined as USAGE POINTER.
That is, NULL=NULL is not allowed.

The following table summarizes the permissible comparisons for USAGE POINTER, NULL, and ADDRESS OF.

Table 31. Permissible comparisons for USAGE POINTER, NULL, and ADDRESS OF

	USAGE POINTER second operand	ADDRESS OF second operand	NULL or NULLS second operand
USAGE POINTER first operand	Yes	Yes	Yes
ADDRESS OF first operand	Yes	Yes	Yes
NULL/NULS first operand	Yes	Yes	No
Yes Comparison allowed only for EQUAL, NOT EQUAL			
No No comparison allowed			

Procedure-pointer and function-pointer relation conditions

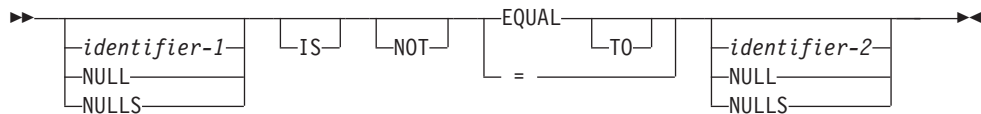
Only EQUAL and NOT EQUAL are allowed as relational operators when specifying procedure-pointer or function-pointer data items in a relation condition.

Procedure-pointer data items are defined explicitly as USAGE PROCEDURE-POINTER. Function-pointer data items are defined explicitly as USAGE FUNCTION-POINTER.

The operands are equal if the two addresses used in the comparison would both result in the same storage location.

This relation condition is allowed in IF, PERFORM, EVALUATE, and SEARCH format-1 statements. It is not allowed in SEARCH format-2 (SEARCH ALL) statements, because there is no meaningful ordering that can be applied to procedure-pointer data items.

Format 3: procedure-pointer and function-pointer relation condition



identifier-1, identifier-2

Must be described as USAGE PROCEDURE-POINTER or USAGE FUNCTION-POINTER. *identifier-1* and *identifier-2* need not be described the same.

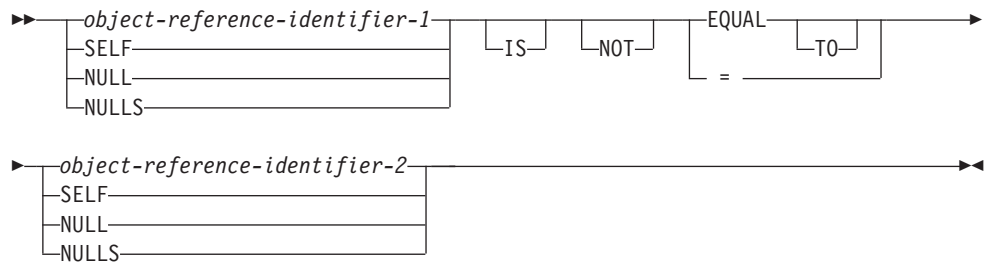
NULL, NULLS

Can be used only if the other operand is defined as USAGE FUNCTION-POINTER or USAGE PROCEDURE-POINTER. That is, NULL=NULL is not allowed.

Object-reference relation conditions

A data item of usage OBJECT REFERENCE can be compared for equality or inequality with another data item of usage OBJECT REFERENCE or with NULL, NULLS, or SELF.

Format 4: object-reference relation condition



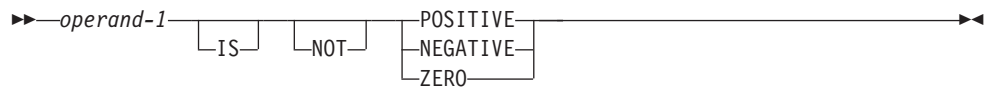
A comparison with SELF is allowed only in a method.

Two object-references compare equal only if the data items identify the same object.

Sign condition

The sign condition determines whether the algebraic value of a numeric operand is greater than, less than, or equal to zero.

Format: sign condition



operand-1

Must be defined as a numeric identifier, or as an arithmetic expression that contains at least one reference to a variable. *operand-1* can be defined as a floating-point identifier.

The operand is:

- POSITIVE if its value is greater than zero
- NEGATIVE if its value is less than zero
- ZERO if its value is equal to zero

An unsigned operand is either POSITIVE or ZERO.

NOT One algebraic test is executed for the truth value of the sign condition. For example, NOT ZERO is regarded as true when the operand tested is positive or negative in value.

Date fields in sign conditions

The operand in a sign condition can be a date field, but is treated as a nondate for the sign condition test. Thus if the operand is an identifier of a windowed date field, date windowing is not done, so the sign condition can be used to test a windowed date field for an all-zero value.

However, if the operand is an arithmetic expression, then any windowed date fields in the expression will be expanded during the computation of the arithmetic result prior to using the result for the sign condition test.

For example, given that:

- Identifier WIN-DATE is defined as a windowed date field and contains a value of zero
- Compiler option DATEPROC is in effect
- Compiler option YEARWINDOW (*starting-year*) is in effect, with a *starting-year* other than 1900

then this sign condition would evaluate to true:

WIN-DATE IS ZERO

whereas this sign condition would evaluate to false:

WIN-DATE + 0 IS ZERO

Switch-status condition

The switch-status condition determines the on or off status of an UPSI switch.

Format
►►— <i>condition-name</i> —————◄◄

condition-name

Must be defined in the special-names paragraph as associated with the on or off value of an UPSI switch. (See “SPECIAL-NAMES paragraph” on page 110.)

The switch-status condition tests the value associated with *condition-name*. (The value is considered to be alphanumeric.) The result of the test is true if the UPSI switch is set to the value (0 or 1) corresponding to *condition-name*. For details, see *UPSI* in the *COBOL for AIX Programming Guide*.

Complex conditions

A complex condition is formed by combining simple conditions, combined conditions, or complex conditions with logical operators, or negating those conditions with logical negation.

Each logical operator must be preceded and followed by a space. The following table shows the logical operators and their meanings.

Table 32. Logical operators and their meanings

Logical operator	Name	Meaning
AND	Logical conjunction	The truth value is true when both conditions are true.

Table 32. Logical operators and their meanings (continued)

Logical operator	Name	Meaning
OR	Logical inclusive OR	The truth value is true when either or both conditions are true.
NOT	Logical negation	Reversal of truth value (the truth value is true if the condition is false).

Unless modified by parentheses, the following is the order of precedence (from highest to lowest):

1. Arithmetic operations
2. Simple conditions
3. NOT
4. AND
5. OR

The truth value of a complex condition (whether parenthesized or not) is the truth value that results from the interaction of all the stated logical operators on either of the following:

- The individual truth values of simple conditions
- The intermediate truth values of conditions logically combined or logically negated

A complex condition can be either of the following:

- A negated simple condition
- A combined condition (which can be negated)

Negated simple conditions

A simple condition is negated through the use of the logical operator NOT.

Format

►►NOT—condition-1—◄◄

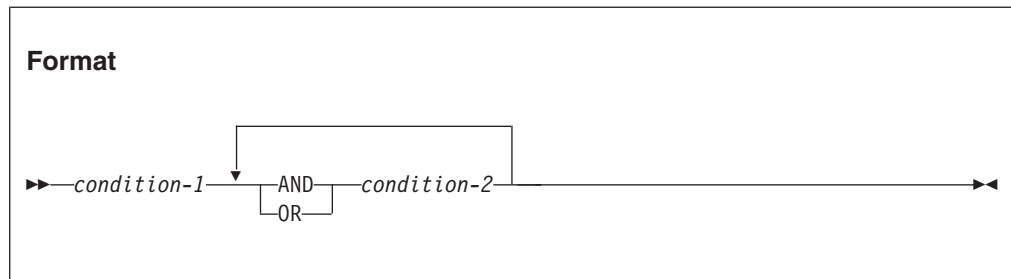
The negated simple condition gives the opposite truth value of the simple condition. That is, if the truth value of the simple condition is true, then the truth value of that same negated simple condition is false, and vice versa.

Placing a negated simple condition within parentheses does not change its truth value. That is, the following two statements are equivalent:

NOT A IS EQUAL TO B.
NOT (A IS EQUAL TO B).

Combined conditions

Two or more conditions can be logically connected to form a combined condition.



The condition to be combined can be any of the following:

- A simple-condition
- A negated simple-condition
- A combined condition
- A negated combined condition (that is, the NOT logical operator followed by a combined condition enclosed in parentheses)
- A combination of the preceding conditions that is specified according to the rules in the following table

Table 33. Combined conditions—permissible element sequences

Combined condition element	Left most	When not leftmost, can be immediately preceded by:	Right most	When not rightmost, can be immediately followed by:
simple-condition	Yes	OR NOT AND (Yes	OR AND)
OR AND	No	simple-condition)	No	simple-condition NOT (
NOT	Yes	OR AND (No	simple-condition (
(Yes	OR NOT AND (No	simple-condition NOT (
)	No	simple-condition)	Yes	OR AND)

Parentheses are never needed when either ANDs or ORs (but not both) are used exclusively in one combined condition. However, parentheses might be needed to modify the implicit precedence rules to maintain the correct logical relation of operators and operands.

There must be a one-to-one correspondence between left and right parentheses, with each left parenthesis to the left of its corresponding right parenthesis.

The following table illustrates the relationships between logical operators and conditions C1 and C2.

Table 34. Logical operators and evaluation results of combined conditions

Value for C1	Value for C2	C1 AND C2	C1 OR C2	NOT (C1 AND C2)	NOT C1 AND C2	NOT (C1 OR C2)	NOT C1 OR C2
True	True	True	True	False	False	False	True
False	True	False	True	True	True	False	True
True	False	False	True	True	False	False	False
False	False	False	False	True	False	True	True

Order of evaluation of conditions

Parentheses, both explicit and implicit, define the level of inclusiveness within a complex condition. Two or more conditions connected by only the logical operators AND or OR at the same level of inclusiveness establish a hierarchical level within a complex condition. Therefore an entire complex condition is a nested structure of hierarchical levels, with the entire complex condition being the most inclusive hierarchical level.

Within this context, the evaluation of the conditions within an entire complex condition begins at the left of the condition. The constituent connected conditions within a hierarchical level are evaluated in order from left to right, and evaluation of that hierarchical level terminates as soon as a truth value for it is determined, regardless of whether all the constituent connected conditions within that hierarchical level have been evaluated.

Values are established for arithmetic expressions and functions if and when the conditions that contain them are evaluated. Similarly, negated conditions are evaluated if and when it is necessary to evaluate the complex condition that they represent. For example:

NOT A IS GREATER THAN B OR A + B IS EQUAL TO C AND D IS POSITIVE

is evaluated as if parenthesized as follows:

(NOT (A IS GREATER THAN B)) OR
(((A + B) IS EQUAL TO C) AND (D IS POSITIVE))

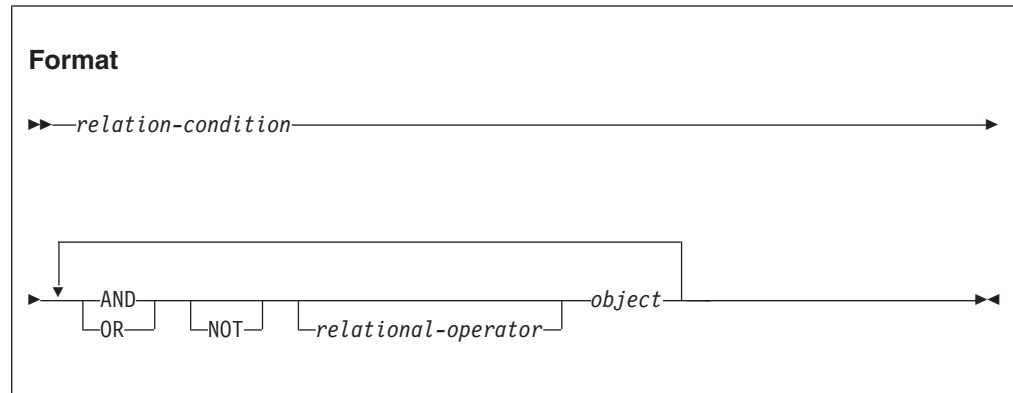
Order of evaluation:

1. (NOT (A IS GREATER THAN B)) is evaluated, giving some intermediate truth value, *t1*. If *t1* is true, the combined condition is true, and no further evaluation takes place. If *t1* is false, evaluation continues as follows.
2. (A + B) is evaluated, giving some intermediate result, *x*.
3. (*x* IS EQUAL TO C) is evaluated, giving some intermediate truth value, *t2*. If *t2* is false, the combined condition is false, and no further evaluation takes place. If *t2* is true, the evaluation continues as follows.
4. (D IS POSITIVE) is evaluated, giving some intermediate truth value, *t3*. If *t3* is false, the combined condition is false. If *t3* is true, the combined condition is true.

Abbreviated combined relation conditions

When relation-conditions are written consecutively, any relation-condition after the first can be abbreviated in one of two ways:

- Omission of the subject
- Omission of the subject and relational operator



In any consecutive sequence of relation-conditions, both forms of abbreviation can be specified. The abbreviated condition is evaluated as if:

1. The last stated subject is the missing subject.
2. The last stated relational operator is the missing relational operator.

The resulting combined condition must comply with the rules for element sequences in combined conditions, as shown in “Combined conditions” on page 282.

If NOT is immediately followed by GREATER THAN, >, LESS THAN, <, EQUAL TO, or =, then the NOT participates as part of the relational operator. NOT in any other position is considered a logical operator (and thus results in a negated relation condition).

Using parentheses

You can use parentheses in combined relation conditions to specify an intended order of evaluation. Using parentheses can also help improve the readability of conditional expressions.

The following rules govern the use of parentheses in abbreviated combined relation conditions:

1. Parentheses can be used to change the order of evaluation of the logical operators AND and OR.
2. The word NOT participates as part of the relational operator when it is immediately followed by GREATER THAN, >, LESS THAN, <, EQUAL TO, or =.
3. NOT in any other position is considered a logical operator and thus results in a negated relation condition. If you use NOT as a logical operator, only the relation condition immediately following the NOT is negated; the negation is not propagated through the abbreviated combined relation condition along with the subject and relational operator.

4. The logical NOT operator can appear within a parenthetical expression that immediately follows a relational operator.
5. When a left parenthesis appears immediately after the relational operator, the relational operator is distributed to all objects enclosed in the parentheses. In the case of a "distributed" relational operator, the subject and relational operator remain current after the right parenthesis which ends the distribution. The following three restrictions apply to cases where the relational operator is distributed throughout the expression:
 - a. A simple condition cannot appear within the scope of the distribution.
 - b. Another relational operator cannot appear within the scope of the distribution.
 - c. The logical operator NOT cannot appear immediately after the left parenthesis, which defines the scope of the distribution.
6. Evaluation proceeds from the least to the most inclusive condition.
7. There must be a one-to-one correspondence between left and right parentheses, with each left parenthesis to the left of its corresponding right parenthesis. If the parentheses are unbalanced, the compiler inserts a parenthesis and issues an E-level message. However, if the compiler-inserted parenthesis results in the truncation of the expression, you will receive an S-level diagnostic message.
8. The last stated subject is inserted in place of the missing subject.
9. The last stated relational operator is inserted in place of the missing relational operator.
10. Insertion of the omitted subject or relational operator ends when:
 - a. Another simple condition is encountered.
 - b. A condition-name is encountered.
 - c. A right parenthesis is encountered that matches a left parenthesis that appears to the left of the subject.
11. In any consecutive sequence of relation conditions, you can use both abbreviated relation conditions that contain parentheses and those that do not.
12. Consecutive logical NOT operators cancel each other and result in an S-level message. Note, however, that an abbreviated combined relation condition can contain two consecutive NOT operators when the second NOT is part of a relational operator. For example, you can abbreviate the first condition as the second condition listed below.

A = B and not A not = C

A = B and not not = C

The following table summarizes the rules for forming an abbreviated combined relation condition.

Table 35. Abbreviated combined conditions: permissible element sequences

Combined condition element	Left- most	When not leftmost, can be immediately preceded by:	Right- most	When not rightmost, can be immediately followed by:
Subject	Yes	NOT (No	Relational operator
Object	No	Relational operator AND OR NOT (Yes	AND OR)

Table 35. Abbreviated combined conditions: permissible element sequences (continued)

Combined condition element	Left- most	When not leftmost, can be immediately preceded by:	Right- most	When not rightmost, can be immediately followed by:
Relational operator	No	Subject AND OR NOT	No	Object (
AND OR	No	Object)	No	Object Relational operator NOT (
NOT	Yes	AND OR (No	Subject Object Relational operator (
(Yes	Relational operator AND OR NOT (No	Subject Object NOT (
)	No	Object)	Yes	AND OR)

The following table shows examples of abbreviated combined relation conditions, with and without parentheses, and their unabbreviated equivalents.

Table 36. Abbreviated combined conditions: unabbreviated equivalents

Abbreviated combined relation condition	Equivalent
A = B AND NOT < C OR D	((A = B) AND (A NOT < C)) OR (A NOT < D)
A NOT > B OR C	(A NOT > B) OR (A NOT > C)
NOT A = B OR C	(NOT (A = B)) OR (A = C)
NOT (A = B OR < C)	NOT ((A = B) OR (A < C))
NOT (A NOT = B AND C AND NOT D)	NOT (((A NOT = B) AND (A NOT = C)) AND (NOT (A NOT = D))))

Statement categories

There are four categories of COBOL statements:

- “Imperative statements”
- “Conditional statements” on page 288
- “Delimited scope statements” on page 289
- “Compiler-directing statements” on page 290

Imperative statements

An *imperative statement* either specifies an unconditional action to be taken by the program or is a conditional statement terminated by its explicit scope terminator (see “Delimited scope statements” on page 289). A series of imperative statements

can be specified wherever an imperative statement is allowed. A conditional statement that is terminated by its explicit scope terminator is also classified as an imperative statement (see “Delimited scope statements” on page 289). The following lists contain the COBOL imperative statements.

Arithmetic

- ADD¹
- COMPUTE¹
- DIVIDE¹
- MULTIPLY¹
- SUBTRACT¹

1. Without the ON SIZE ERROR or the NOT ON SIZE ERROR phrase.

Data movement

- ACCEPT (DATE, DAY, DAY-OF-WEEK, TIME)
- INITIALIZE
- INSPECT
- MOVE
- SET
- STRING²
- UNSTRING²
- XML GENERATE⁸
- XML PARSE⁸

2. Without the ON OVERFLOW or the NOT ON OVERFLOW phrase.

8. Without the ON EXCEPTION or NOT ON EXCEPTION phrase.

Ending

- STOP RUN
- EXIT PROGRAM
- EXIT METHOD
- GOBACK

Input-output

- ACCEPT *identifier*
- CLOSE
- DELETE³
- DISPLAY
- OPEN
- READ⁴
- REWRITE³
- START³
- STOP *literal*
- WRITE⁵

3. Without the INVALID KEY or the NOT INVALID KEY phrase.

4. Without the AT END or NOT AT END, and INVALID KEY or NOT INVALID KEY phrases.

5. Without the INVALID KEY or NOT INVALID KEY, and END-OF-PAGE or NOT END-OF-PAGE phrases.

Ordering

- MERGE
- RELEASE
- RETURN⁶
- SORT

6. Without the AT END or NOT AT END phrase.

Procedure-branching

- ALTER
- EXIT
- GO TO
- PERFORM

Program or method linkage

- CALL⁷
- CANCEL
- INVOKE

7. Without the ON OVERFLOW phrase, and without the ON EXCEPTION or NOT ON EXCEPTION phrase.

Table-handling

- SET

Conditional statements

A *conditional statement* specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value. (See “Conditional expressions” on page 262.) The following lists contain COBOL statements that become conditional when a *condition* (for example, ON SIZE ERROR or ON OVERFLOW) is included and when the statement is not terminated by its explicit scope terminator.

Arithmetic

- ADD ... ON SIZE ERROR
- ADD ... NOT ON SIZE ERROR
- COMPUTE ... ON SIZE ERROR
- COMPUTE ... NOT ON SIZE ERROR
- DIVIDE ... ON SIZE ERROR
- DIVIDE ... NOT ON SIZE ERROR
- MULTIPLY ... ON SIZE ERROR
- MULTIPLY ... NOT ON SIZE ERROR
- SUBTRACT ... ON SIZE ERROR
- SUBTRACT ... NOT ON SIZE ERROR

Data movement

- STRING ... ON OVERFLOW
- STRING ... NOT ON OVERFLOW
- UNSTRING ... ON OVERFLOW
- UNSTRING ... NOT ON OVERFLOW
- XML GENERATE ... ON EXCEPTION
- XML GENERATE ... NOT ON EXCEPTION
- XML PARSE ... ON EXCEPTION
- XML PARSE ... NOT ON EXCEPTION

Decision

- IF
- EVALUATE

Input-output

- DELETE ... INVALID KEY
- DELETE ... NOT INVALID KEY
- READ ... AT END
- READ ... NOT AT END
- READ ... INVALID KEY
- READ ... NOT INVALID KEY
- REWRITE ... INVALID KEY
- REWRITE ... NOT INVALID KEY
- START ... INVALID KEY
- START ... NOT INVALID KEY
- WRITE ... AT END-OF-PAGE
- WRITE ... NOT AT END-OF-PAGE
- WRITE ... INVALID KEY
- WRITE ... NOT INVALID KEY

Ordering

- RETURN ... AT END
- RETURN ... NOT AT END

Program or method linkage

- CALL ... ON OVERFLOW
- CALL ... ON EXCEPTION
- CALL ... NOT ON EXCEPTION
- INVOKE ... ON EXCEPTION
- INVOKE ... NOT ON EXCEPTION

Table-handling

- SEARCH

Delimited scope statements

In general, a DELIMITED SCOPE statement uses an explicit scope terminator to turn a conditional statement into an imperative statement. The resulting imperative statement can then be nested. Explicit scope terminators can also be used to

terminate the scope of an imperative statement. Explicit scope terminators are provided for all COBOL statements that can have conditional phrases.

Unless explicitly specified otherwise, a delimited scope statement can be specified wherever an imperative statement is allowed by the rules of the language.

Explicit scope terminators

An *explicit scope terminator* marks the end of certain procedure division statements. A conditional statement that is delimited by its explicit scope terminator is considered an imperative statement and must follow the rules for imperative statements.

These are the explicit scope terminators:

- END-ADD
- END-CALL
- END-COMPUTE
- END-DELETE
- END-DIVIDE
- END-EVALUATE
- END-IF
- END-INVOKE
- END-MULTIPLY
- END-PERFORM
- END-READ
- END-RETURN
- END-REWRITE
- END-SEARCH
- END-START
- END-STRING
- END-SUBTRACT
- END-UNSTRING
- END-WRITE
- END-XML

Implicit scope terminators

At the end of any sentence, an *implicit scope terminator* is a separator period that terminates the scope of all previous statements not yet terminated.

An unterminated conditional statement cannot be contained by another statement.

Except for nesting conditional statements within IF statements, nested statements must be imperative statements and must follow the rules for imperative statements. You should not nest conditional statements.

Compiler-directing statements

Statements that direct the compiler to take a specified action are discussed in Chapter 23, “Compiler-directing statements,” on page 537.

Statement operations

COBOL statements perform the following types of operations:

- Arithmetic
- Data manipulation
- Input/output
- Procedure branching

There are several phrases common to arithmetic and data manipulation statements, such as:

- CORRESPONDING phrase
- GIVING phrase
- ROUNDED phrase
- SIZE ERROR phrases

CORRESPONDING phrase

The CORRESPONDING (CORR) phrase causes ADD, SUBTRACT, and MOVE operations to be performed on elementary data items of the same name if the alphanumeric group item or national group item to which they belong is specified. A national group is processed as a group item when the CORRESPONDING phrase is used.

Both identifiers that follow the keyword CORRESPONDING must name group items. In this discussion, these identifiers are referred to as *identifier-1* and *identifier-2*. *identifier-1* references the sending group item. *identifier-2* references the receiving group item.

Two subordinate data items, one from *identifier-1* and one from *identifier-2*, correspond if the following conditions are true:

- In an ADD or SUBTRACT statement, both of the data items are elementary numeric data items. Other data items are ignored.
- In a MOVE statement, at least one of the data items is an elementary item, and the move is permitted by the move rules.
- The two subordinate items have the same name and the same qualifiers up to but not including *identifier-1* and *identifier-2*.
- The subordinate items are not identified by the keyword FILLER.
- Neither *identifier-1* nor *identifier-2* is described as a level 66, 77, or 88 item, and neither is described as an index data item. Neither *identifier-1* nor *identifier-2* can be reference-modified.
- Neither *identifier-1* nor *identifier-2* is described with USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE.
- The subordinate items do not include a REDEFINES, RENAMES, OCCURS, USAGE INDEX, USAGE POINTER, USAGE PROCEDURE-POINTER, USAGE FUNCTION-POINTER, or USAGE OBJECT REFERENCE clause in their descriptions.

However, *identifier-1* and *identifier-2* themselves can contain or be subordinate to items that contain a REDEFINES or OCCURS clause in their descriptions.

- The name of each subordinate data item that satisfies these conditions is unique after application of implicit qualifiers.

identifier-1, *identifier-2*, or both can be subordinate to a FILLER item.

For example, consider two data hierarchies defined as follows:

```
05 ITEM-1 OCCURS 6.  
    10 ITEM-A PIC S9(3).  
    10 ITEM-B PIC +99.9.  
    10 ITEM-C PIC X(4).  
    10 ITEM-D REDEFINES ITEM-C PIC 9(4).  
    10 ITEM-E USAGE COMP-1.  
    10 ITEM-F USAGE INDEX.  
05 ITEM-2.  
    10 ITEM-A PIC 99.  
    10 ITEM-B PIC +9V9.  
    10 ITEM-C PIC A(4).  
    10 ITEM-D PIC 9(4).  
    10 ITEM-E PIC 9(9) USAGE COMP.  
    10 ITEM-F USAGE INDEX.
```

If ADD CORR ITEM-2 TO ITEM-1(x) is specified, ITEM-A and ITEM-A(x), ITEM-B and ITEM-B(x), and ITEM-E and ITEM-E(x) are considered to be corresponding and are added together. ITEM-C and ITEM-C(x) are not included because they are not numeric. ITEM-D and ITEM-D(x) are not included because ITEM-D(x) includes a REDEFINES clause in its data description. ITEM-F and ITEM-F(x) are not included because they are index data items. Note that ITEM-1 is valid as either *identifier-1* or *identifier-2*.

If any of the individual operations in the ADD CORRESPONDING statement produces a size error condition, *imperative-statement-1* in the ON SIZE ERROR phrase is not executed until all of the individual additions are completed.

GIVING phrase

The value of the identifier that follows the word GIVING is set equal to the calculated result of the arithmetic operation. Because this identifier is not involved in the computation, it can be a numeric-edited item.

ROUNDED phrase

After decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is compared with the number of places provided for the fraction of the resultant identifier.

When the size of the fractional result exceeds the number of places provided for its storage, truncation occurs unless ROUNDED is specified. When ROUNDED is specified, the least significant digit of the resultant identifier is increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

When the resultant identifier is described by a PICTURE clause that contains rightmost Ps and when the number of places in the calculated result exceeds the number of integer positions specified, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated.

In a floating-point arithmetic operation, the ROUNDED phrase has no effect; the result of a floating-point operation is always rounded. For more information on floating-point arithmetic expressions, see *Fixed-point contrasted with floating-point arithmetic* in the *COBOL for AIX Programming Guide*.

When the ARITH(EXTEND) compiler option is in effect, the ROUNDED phrase is not supported for arithmetic receivers with 31 digit positions to the right of the decimal point. For example, neither X nor Y below is valid as a receiver with the ROUNDED phrase:

```
01 X PIC V31.
01 Y PIC P(30)9(1).

...
COMPUTE X ROUNDED = A + B
COMPUTE Y ROUNDED = A - B
```

Otherwise, the ROUNDED phrase is fully supported for extended-precision arithmetic statements.

SIZE ERROR phrases

A size error condition can occur in four different ways:

- When the absolute value of the result of an arithmetic evaluation, after decimal point alignment, exceeds the largest value that can be contained in the result field.
- When division by zero occurs.
- When the result of an arithmetic statement is stored in a windowed date field and the year of the result falls outside the century window. For example, given YEARWINDOW(1940) (which specifies a century window of 1940–2039), the following SUBTRACT statement causes a size error:

```
01 WINDOWED-YEAR DATE FORMAT YY PICTURE 99
    VALUE IS 50.

...
SUBTRACT 20 FROM WINDOWED-YEAR
ON SIZE ERROR imperative-statement
```

The size error occurs because the result of the subtraction, a windowed date field, has an effective year value of 1930, which falls outside the century window. For details on how windowed date fields are treated as if they were converted to expanded date format, see “Subtraction that involves date fields” on page 261.

For more information about how size errors can occur when using date fields, see “Storing arithmetic results that involve date fields” on page 261.

- In an exponential expression, as indicated in the following table:

Table 37. Exponentiation size error conditions

Size error	Action taken when a SIZE ERROR clause is present	Action taken when a SIZE ERROR clause is not present
Zero raised to zero power	The SIZE ERROR imperative is executed.	The value returned is 1, and a message is issued.
Zero raised to a negative number	The SIZE ERROR imperative is executed.	The program is terminated abnormally.
A negative number raised to a fractional power	The SIZE ERROR imperative is executed.	The absolute value of the base is used, and a message is issued.

The size error condition applies only to final results, not to any intermediate results.

If the resultant identifier is defined with usage `BINARY`, `COMPUTATIONAL`, `COMPUTATIONAL-4`, or `COMPUTATIONAL-5`, the largest value that the resultant data item can contain is the value implied by the item's decimal `PICTURE` character-string, regardless of the `TRUNC` compiler option in effect.

If the `ROUNDED` phrase is specified, rounding takes place before size error checking.

When a size error occurs, the subsequent action of the program depends on whether the `ON SIZE ERROR` phrase is specified.

If the `ON SIZE ERROR` phrase is not specified and a size error condition occurs, truncation rules apply and the value of the affected resultant identifier is computed.

If the `ON SIZE ERROR` phrase is specified and a size error condition occurs, the value of the resultant identifier affected by the size error is not altered; that is, the error results are not placed in the receiving identifier. After completion of the execution of the arithmetic operation, the imperative statement in the `ON SIZE ERROR` phrase is executed, control is transferred to the end of the arithmetic statement, and the `NOT ON SIZE ERROR` phrase, if specified, is ignored.

For `ADD CORRESPONDING` and `SUBTRACT CORRESPONDING` statements, if an individual arithmetic operation causes a size error condition, the `ON SIZE ERROR` imperative statement is not executed until all the individual additions or subtractions have been completed.

If the `NOT ON SIZE ERROR` phrase has been specified and, after execution of an arithmetic operation, a size error condition does not exist, the `NOT ON SIZE ERROR` phrase is executed.

When both the `ON SIZE ERROR` and `NOT ON SIZE ERROR` phrases are specified, and the statement in the phrase that is executed does not contain any explicit transfer of control, then if necessary an implicit transfer of control is made after execution of the phrase to the end of the arithmetic statement.

Arithmetic statements

The arithmetic statements are used for computations. Individual operations are specified by the `ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE` statements. These individual operations can be combined symbolically in a formula that uses the `COMPUTE` statement.

Arithmetic statement operands

The data descriptions of operands in an arithmetic statement need not be the same. Throughout the calculation, the compiler performs any necessary data conversion and decimal point alignment.

Size of operands

If the `ARITH(COMPAT)` compiler option is in effect, the maximum size of each operand is 18 decimal digits. If the `ARITH(EXTEND)` compiler option is in effect, the maximum size of each operand is 31 decimal digits.

The *composite of operands* is a hypothetical data item resulting from aligning the operands at the decimal point and then superimposing them on one another.

If the ARITH(COMPAT) compiler option is in effect, the composite of operands can be a maximum of 30 digits. If the ARITH(EXTEND) compiler option is in effect, the composite of operands can be a maximum of 31 digits.

The following table shows how the composite of operands is determined for arithmetic statements:

Table 38. How the composite of operands is determined

Statement	Determination of the composite of operands
SUBTRACT ADD	Superimposing all operands in a given statement except those following the word GIVING
MULTIPLY	Superimposing all receiving data items
DIVIDE	Superimposing all receiving data items except the REMAINDER data item
COMPUTE	Restriction does not apply

For example, assume that each item is defined as follows in the data division:

A PICTURE 9(7)V9(5).
B PICTURE 9(11)V99.
C PICTURE 9(12)V9(3).

If the following statement is executed, the composite of operands consists of 17 decimal digits:

ADD A B TO C

It has the following implicit description:

COMPOSITE-OF-OPERANDS PICTURE 9(12)V9(5).

In the ADD and SUBTRACT statements, if the composite of operands is 30 digits or less with the ARITH(COMPAT) compiler option, or 31 digits or less with the ARITH(EXTEND) compiler option, the compiler ensures that enough places are carried so that no significant digits are lost during execution.

In all arithmetic statements, it is important to define data with enough digits and decimal places to ensure the desired accuracy in the final result. For more information, see *Appendix A. Intermediate results and arithmetic precision* in the *COBOL for AIX Programming Guide*.

Overlapping operands

When operands in an arithmetic statement share part of their storage (that is, when the operands overlap), the result of the execution of such a statement is unpredictable.

Multiple results

When an arithmetic statement has multiple results, execution conceptually proceeds as follows:

1. The statement performs all arithmetic operations to find the result to be placed in the receiving items, and stores that result in a temporary location.

2. A sequence of statements transfers or combines the value of this temporary result with each single receiving field. The statements are considered to be written in the same left-to-right order in which the multiple results are listed.

For example, executing the following statement:

```
ADD A, B, C, TO C, D(C), E.
```

is equivalent to executing the following series of statements:

```
ADD A, B, C GIVING TEMP.  
ADD TEMP TO C.  
ADD TEMP TO D(C).  
ADD TEMP TO E.
```

In the above example, TEMP is a compiler-supplied temporary result field. When the addition operation for D(C) is performed, the subscript C contains the new value of C.

Data manipulation statements

The following COBOL statements move and inspect data: ACCEPT, INITIALIZE, INSPECT, MOVE, READ, RELEASE, RETURN, REWRITE, SET, STRING, UNSTRING, WRITE, XML PARSE, and XML GENERATE.

Overlapping operands

When the sending and receiving fields of a data manipulation statement share a part of their storage (that is, when the operands overlap), the result of the execution of such a statement is unpredictable.

Input-output statements

COBOL input-output statements transfer data to and from files stored on external media, and also control low-volume data that is obtained from or sent to an input/output device.

In COBOL, the unit of file data made available to the program is a record. You need only be concerned with such records. Provision is automatically made for such operations as the movement of data into buffers, internal storage, validity checking, error correction (where feasible), blocking and deblocking, and volume-switching procedures.

The description of the file in the environment division and the data division governs which input-output statements are allowed in the procedure division. Permissible statements for sequential files are shown in Table 52 on page 394, and permissible statements for indexed files and relative files are shown in Table 53 on page 394. Permissible statements for line sequential files are shown in Table 54 on page 395.

Common processing facilities

There are several common processing facilities that apply to more than one input-output statement. The common processing facilities provided are:

- “File status key” on page 297
- “Invalid key condition” on page 300
- “INTO and FROM phrases” on page 301

- “File position indicator” on page 302

Discussions in the following sections use the terms *volume* and *reel*. The term *volume* refers to all non-unit-record input-output devices. The term *reel* applies only to tape devices. Treatment of direct-access devices in the sequential access mode is logically equivalent to the treatment of tape devices.

File status key

If the FILE STATUS clause is specified in the file-control entry, a value is placed in the specified file status key (the two-character data item named in the FILE STATUS clause) during execution of any request on that file; the value indicates the status of that request. The value is placed in the file status key before execution of any EXCEPTION/ERROR declarative, INVALID KEY phrase, or AT END phrase associated with the request.

There are two file status key data-names. One is described by *data-name-1* in the FILE STATUS clause of the file-control entry. This is a two-character data item with the first character known as file status key 1 and the second character known as file status key 2. The combinations of possible values and their meanings are shown in Table 39.

The other file status key is described by *data-name-8* in the FILE STATUS clause of the file-control entry. *data-name-8* does not apply to line-sequential files. For more information about *data-name-8*, see “FILE STATUS clause” on page 141.

Table 39. File status key values and meanings

High-order digit	Meaning	Low-order digit	Meaning
0	Successful completion	0	No further information
		2	This file status value applies only to indexed files with alternate keys that allow duplicates. The input-output statement was successfully executed, but a duplicate key was detected. For a READ statement, the key value for the current key of reference was equal to the value of the same key in the next record within the current key of reference. For a REWRITE or WRITE statement, the record just written created a duplicate key value for at least one alternate record key for which duplicates are allowed.
		4	A READ statement was successfully executed, but the length of the record being processed did not conform to the fixed file attributes for that file, or the record for an RSD file was not followed by a new-line character.
		5	An OPEN statement was successfully executed, but the referenced optional file was unavailable at the time the OPEN statement was executed. The file had been created if the open mode was I-O or EXTEND.
		7	For a CLOSE statement with the NO REWIND, REEL/UNIT, or FOR REMOVAL phrase or for an OPEN statement with the NO REWIND phrase, the referenced file was on a non-reel/unit medium.

Table 39. File status key values and meanings (continued)

High-order digit	Meaning	Low-order digit	Meaning
1	At-end condition	0	A sequential READ statement was attempted and no next logical record existed in the file because the end of the file had been reached. Or the first READ was attempted on an optional input file that was unavailable.
		4	A sequential READ statement was attempted for a relative file, and the number of significant digits in the relative record number was larger than the size of the relative key data item described for the file.
2	Invalid key condition	1	A sequence error exists for a sequentially accessed indexed file. The prime record key value was changed by the program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file. Or the ascending requirements for successive record key values were violated.
		2	An attempt was made to write a record that would create a duplicate key in a relative file. Or an attempt was made to write or rewrite a record that would create a duplicate prime record key or a duplicate alternate record key without the DUPLICATES phrase in an indexed file.
		3	An attempt was made to randomly access a record that does not exist in the file. Or a START or random READ statement was attempted on an optional input file that was unavailable.
		4	An attempt was made to write beyond the externally defined boundaries of a relative or indexed file. Or a sequential WRITE statement was attempted for a relative file and the number of significant digits in the relative record number was larger than the size of the relative key data item described for the file.

Table 39. File status key values and meanings (continued)

High-order digit	Meaning	Low-order digit	Meaning
3	Permanent error condition	0	A WRITE statement with the BEFORE, AFTER phrase was attempted on an RSD file.
		3	<p>An OPEN statement was unsuccessful because concatenated files were specified, but one or more of the requirements for concatenation were not satisfied. Possible violations are:</p> <ul style="list-style-type: none"> • The file ORGANIZATION was not SEQUENTIAL or LINE SEQUENTIAL. • The ACCESS MODE was not SEQUENTIAL. • The OPEN mode was not INPUT. • Too many file identifiers were specified in the concatenation, or memory could not be obtained to record the file identifiers that were specified. <p>An OPEN or READ statement was unsuccessful because a nonoptional file in a concatenation was not available. For example, the file did not exist or you did not have sufficient access permissions.</p> <p><i>Data-name-8</i>, if specified, contains the original file status value, the failing file identifier, and any additional information that would have been supplied in <i>data-name-8</i> for a nonconcatenated file specification.</p>
		4	A permanent error exists because of a boundary violation; an attempt was made to write beyond the externally defined boundaries of a sequential file.
		5	An OPEN statement with the INPUT, I-O, or EXTEND phrase was attempted on a nonoptional file that was unavailable.
		7	<p>An OPEN statement was attempted on a file that would not support the open mode specified in the OPEN statement. Possible violations are:</p> <ul style="list-style-type: none"> • The EXTEND or OUTPUT phrase was specified but the file would not support write operations. • The I-O phrase was specified but the file would not support the input and output operations permitted. • The INPUT phrase was specified but the file would not support read operations.
		8	An OPEN statement was attempted on a file previously closed with lock.
		9	<p>The OPEN statement was unsuccessful because a conflict was detected between the fixed file attributes and the attributes specified for that file in the program. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the maximum record size, the record type (fixed or variable), and the blocking factor.</p> <p>An RSD file was declared with the indexed or relative organization, variable-length records, or the LINAGE clause.</p> <p>File status 39 is not supported for line-sequential files.</p>

Table 39. File status key values and meanings (continued)

High-order digit	Meaning	Low-order digit	Meaning
4	Logic error condition	1	An OPEN statement was attempted for a file in the open mode.
		2	A CLOSE statement was attempted for a file not in the open mode.
		3	For a mass storage file in the sequential access mode, the last input-output statement executed for the associated file prior to the execution of a REWRITE statement was not a successfully executed READ statement. For relative and indexed files in the sequential access mode, the last input-output statement executed for the file prior to the execution of a DELETE or REWRITE statement was not a successfully executed READ statement.
		4	A boundary violation exists because an attempt was made to rewrite a record to a file and the record was not the same size as the record being replaced. Or an attempt was made to write or rewrite a record that was larger than the largest or smaller than the smallest record allowed by the RECORD IS VARYING clause of the associated file-name.
		6	A sequential READ statement was attempted on a file open in the input or I-O mode and no valid next record had been established because: <ul style="list-style-type: none"> • The preceding READ statement was unsuccessful but did not cause an at-end condition. • The preceding READ statement caused an at-end condition.
		7	The execution of a READ statement was attempted on a file not open in the input or I-O mode.
		8	The execution of a WRITE statement was attempted on a file not open in the I-O, output, or extend mode.
		9	The execution of a DELETE or REWRITE statement was attempted on a file not open in the I-O mode.
9	Implementor-defined condition	0	No further information.
		1	Authorization failure
		2	Logic error
		3	Resource unavailable
		4	Concurrent open error
		5	Invalid or incomplete file information
		6	File system unavailable
		8	Open failed due to locked file
		9	Record access failed due to locked record

Invalid key condition

The invalid key condition can occur during execution of a START, READ, WRITE, REWRITE, or DELETE statement. When an invalid key condition occurs, the input-output statement that caused the condition is unsuccessful.

When the invalid key condition is recognized, actions are taken in the following order:

1. If the FILE STATUS clause is specified in the file-control entry, a value is placed into the file status key to indicate an invalid key condition, as shown in Table 39 on page 297.
2. If the INVALID KEY phrase is specified in the statement that caused the condition, control is transferred to the INVALID KEY imperative statement. Any EXCEPTION/ERROR declarative procedure specified for this file is not executed. Execution then continues according to the rules for each statement specified in the imperative statement.
3. If the INVALID KEY phrase is not specified in the input-output statement for a file and an applicable EXCEPTION/ERROR procedure exists, that procedure is executed. The NOT INVALID KEY phrase, if specified, is ignored.

Both the INVALID KEY phrase and the EXCEPTION/ERROR procedure can be omitted.

If the invalid key condition does not exist after execution of the input-output operation, the INVALID KEY phrase is ignored, if specified, and the following actions are taken:

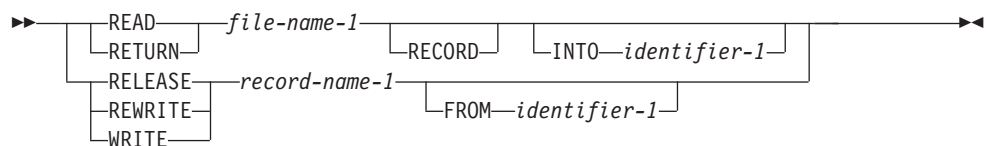
- If an exception condition that is not an invalid key condition exists, control is transferred according to the rules of the USE statement following the execution of any USE AFTER EXCEPTION procedure.
- If no exception condition exists, control is transferred to the end of the input-output statement or the imperative statement specified in the NOT INVALID KEY phrase, if it is specified.

INTO and FROM phrases

The INTO and FROM phrases are valid for READ, RETURN, RELEASE, REWRITE, and WRITE statements.

You must specify an identifier that is the name of an entry in the working-storage section or the linkage section, or of a record description for another previously opened file.

Format: INTO and FROM phrases of input-output statements



- *record-name-1* and *identifier-1* must not refer to the same storage area.
- If *record-name-1* or *identifier-1* refers to a national group item, the item is processed as an elementary data item of category national.
- The INTO phrase can be specified in a READ or RETURN statement.

The result of the execution of a READ or RETURN statement with the INTO phrase is equivalent to the application of the following rules in the order specified:

- The execution of the same READ or RETURN statement without the INTO phrase.

- The current record is moved from the record area to the area specified by *identifier-1* according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified in the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the READ or RETURN statement was unsuccessful. Any subscripting or reference-modification associated with *identifier-1* is evaluated after the record has been read or returned and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by *identifier-1*.

- The FROM phrase can be specified in a RELEASE, REWRITE, or WRITE statement.

The result of the execution of a RELEASE, REWRITE, or WRITE statement with the FROM phrase is equivalent to the execution of the following statements in the order specified:

1. MOVE *identifier-1* TO *record-name-1*
2. The same RELEASE, REWRITE, or WRITE statement without the FROM phrase

After the execution of the RELEASE, REWRITE or WRITE statement is complete, the information in the area referenced by *identifier-1* is available even though the information in the area referenced by *record-name-1* is unavailable, except as specified by the SAME RECORD AREA clause.

File position indicator

The file position indicator is a conceptual entity used in this document to facilitate exact specification of the next record (or, alternatively, the previous record) to be accessed within a given file during certain sequences of input-output operations. The setting of the file position indicator is affected only by the OPEN, CLOSE, READ and START statements. The concept of a file position indicator has no meaning for a file opened in the output or extend mode.

Chapter 21. Procedure division statements

Statements, sentences, and paragraphs in the procedure division are executed sequentially except when a procedure branching statement such as EXIT, GO TO, PERFORM, GOBACK, or STOP is used.

ACCEPT statement

The ACCEPT statement transfers data or system date-related information into the data area referenced by the specified identifier. There is no editing or error checking of the incoming data.

Data transfer

Format 1: data transfer

```
➡—ACCEPT—identifier-1—FROM—mnemonic-name-1—environment-name—➡
```

Format 1 transfers data from an input source into the data item referenced by *identifier-1* (the receiving area). When the FROM phrase is omitted, the system input device is assumed.

Format 1 is useful for exceptional situations in a program when operator intervention (to supply a given message, code, or exception indicator) is required. The operator must of course be supplied with the appropriate messages with which to reply.

The input file must be a byte stream file (for example, a file consisting of text data with records delimited by a record terminator). You can create a byte stream file in your COBOL program using line-sequential file I-O or with the DISPLAY statement. (Most text editors can be used to create a byte stream file also.)

The input file cannot be a SdU, SFS, or STL file (including sequential, relative, or indexed files).

If the source of the ACCEPT statement is a file, and the receiving area is filled without using the full record delimited by the record terminator, the remainder of the input record is used in the next ACCEPT statement for the file. The record delimiter characters are removed from the input data before the input records are moved into the receiving area.

If the source for the ACCEPT statement is a keyboard, the data entered at the keyboard followed by the Enter key is treated as the input data. If the input data is shorter than the receiving area, the area is padded with spaces of the appropriate representation for the receiving area.

identifier-1

The receiving area. Can be:

- An alphanumeric group item
- A national group item
- An elementary data item of usage DISPLAY, DISPLAY-1, or NATIONAL

A national group item is processed as an elementary data item of category national.

If identifier-1 is of usage NATIONAL and the source of the input data is a keyboard, the input is converted from the native code page (the code page indicated by the run-time locale) to national character representation.

mnemonic-name-1

Specifies the input device. *mnemonic-name-1* must be associated in the SPECIAL-NAMES paragraph with an environment-name. See “SPECIAL-NAMES paragraph” on page 110.

environment-name

Identifies the source of input data. An environment-name from the names given in Table 5 on page 112 can be specified.

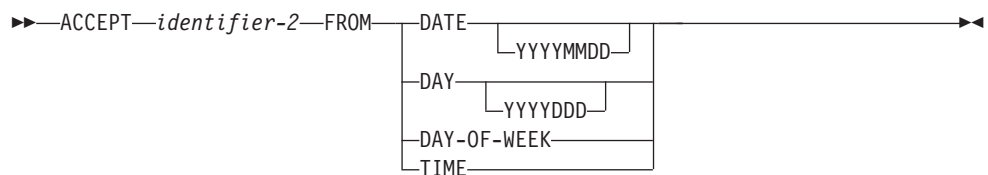
ACCEPT with an environment-name uses the source associated with that environment-name by environment variable assignment in the operating environment. If the environment variable that corresponds to the COBOL environment name is not set, ACCEPT from SYSIN or SYSIPT is from the system logical input device; ACCEPT from CONSOLE is from the user's keyboard. For more information about environment variables, see *Setting and accessing environment variables* in the *COBOL for AIX Programming Guide*.

If the device is the same as that used for READ statements for a LINE SEQUENTIAL file, results are unpredictable.

System date-related information transfer

System information contained in the specified conceptual data items DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, or TIME, can be transferred into the data item referenced by *identifier-2*. The transfer must follow the rules for the MOVE statement without the CORRESPONDING phrase. See “MOVE statement” on page 380.

Format 2: system information transfer



identifier-2

The receiving area. Can be:

- An alphanumeric group item
- A national group item
- An elementary data item of one of the following categories:
 - alphanumeric
 - alphanumeric-edited
 - numeric-edited (with usage DISPLAY or NATIONAL)
 - national
 - national-edited

- numeric
- internal floating-point
- external floating-point (with usage DISPLAY or NATIONAL)

A national group item is processed as an elementary data item of category national.

Format 2 accesses the current date in two formats: the day of the week or the time of day as carried by the system (which can be useful in identifying when a particular run of an object program was executed). You can also use format 2 to supply the date in headings and footings.

The current date and time can also be accessed with the intrinsic function CURRENT-DATE, which also supports four-digit year values and provides additional information (see “CURRENT-DATE” on page 502).

DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, and TIME

The conceptual data items DATE, DATE YYYYMMDD, DAY, DAY YYYYDDD, DAY-OF-WEEK, and TIME implicitly have USAGE DISPLAY. Because these are conceptual data items, they cannot be described in the COBOL program.

The content of the conceptual data items is moved to the receiving area using the rules of the MOVE statement. If the receiving area is of usage NATIONAL, the data is converted to national character representation.

DATE

Has the implicit PICTURE 9(6). If the DATEPROC compiler option is in effect, then the returned value has implicit DATE FORMAT YYXXXX, and *identifier-2* must be defined with this date format.

The sequence of data elements (from left to right) is:

Two digits for the year
Two digits for the month
Two digits for the day

Thus 27 April 2003 is expressed as 030427.

DATE YYYYMMDD

Has the implicit PICTURE 9(8). If the DATEPROC compiler option is in effect, then the returned value has implicit DATE FORMAT YYYYXXXX, and *identifier-2* must be defined with this date format.

The sequence of data elements (from left to right) is:

Four digits for the year
Two digits for the month
Two digits for the day

Thus 27 April 2003 is expressed as 20030427.

DAY

Has the implicit PICTURE 9(5). If the DATEPROC compiler option is in effect, then the returned value has implicit DATE FORMAT YYXXXX, and *identifier-2* must be defined with this date format.

The sequence of data elements (from left to right) is:

Two digits for the year
Three digits for the day

Thus 27 April 2003 is expressed as 03117.

DAY YYYYDDD

Has the implicit PICTURE 9(7). If the DATEPROC compiler option is in effect, then the returned value has implicit DATE FORMAT YYYYXXX, and *identifier-2* must be defined with this date format.

The sequence of data elements (from left to right) is:

Four digits for the year
Three digits for the day

Thus 27 April 2003 is expressed as 2003117.

DAY-OF-WEEK

Has the implicit PICTURE 9(1).

The single data element represents the day of the week according to the following values:

1 represents Monday	5 represents Friday
2 represents Tuesday	6 represents Saturday
3 represents Wednesday	7 represents Sunday
4 represents Thursday	

Thus Wednesday is expressed as 3.

TIME Has the implicit PICTURE 9(8).

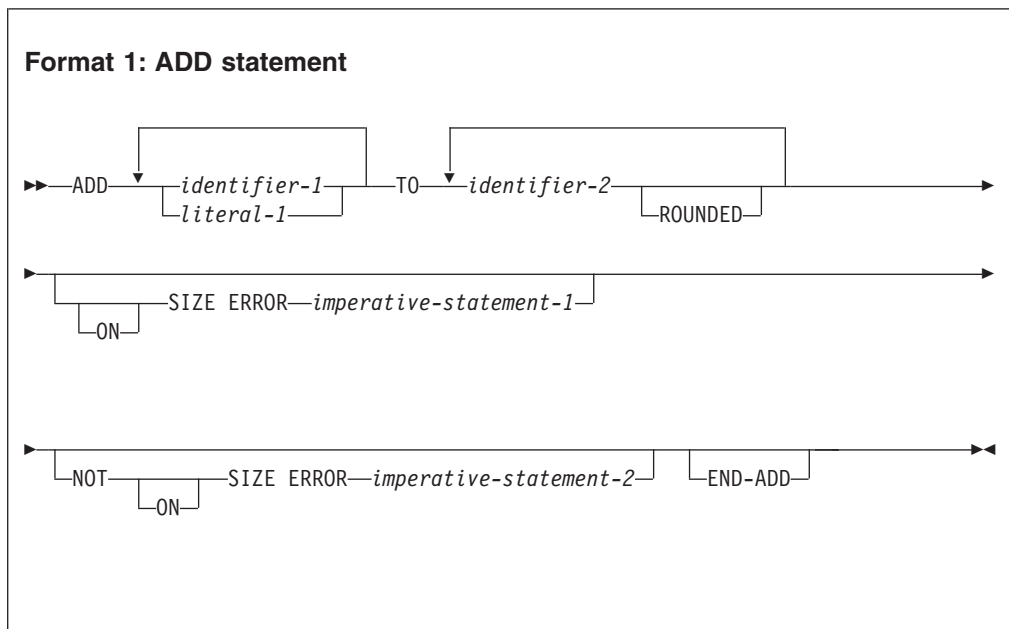
The sequence of data elements (from left to right) is:

Two digits for hour of day
Two digits for minute of hour
Two digits for second of minute
Two digits for hundredths of second

Thus 2:41 PM is expressed as 14410000.

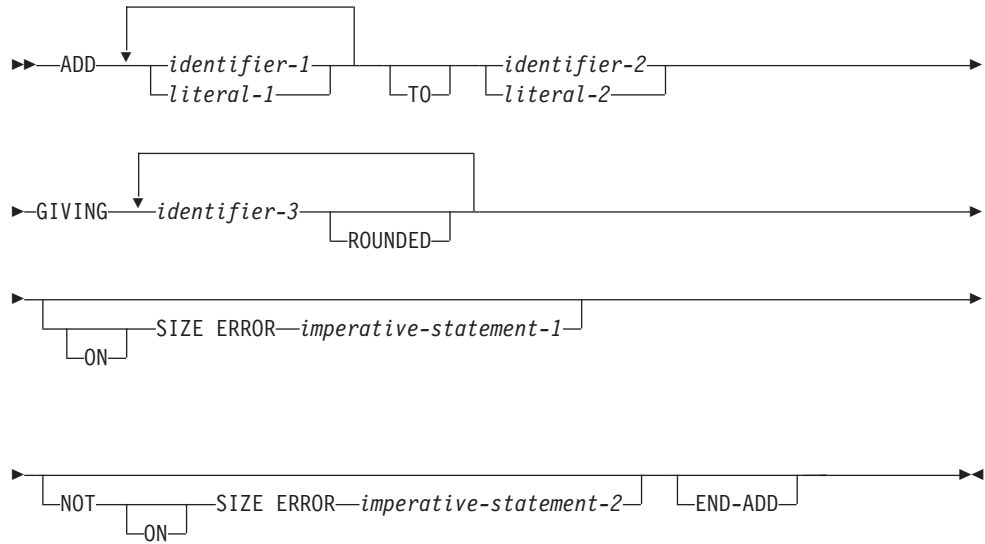
ADD statement

The ADD statement sums two or more numeric operands and stores the result.



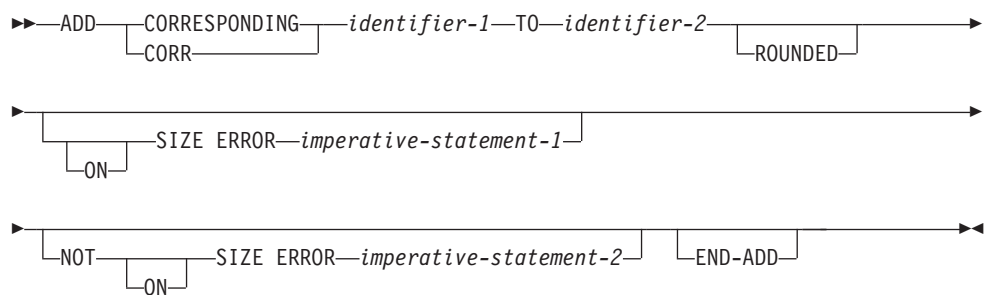
All identifiers or literals that precede the keyword **TO** are added together, and this sum is added to and stored in *identifier-2*. This process is repeated for each successive occurrence of *identifier-2* in the left-to-right order in which *identifier-2* is specified.

Format 2: ADD statement with GIVING phrase



The values of the operands that precede the word GIVING are added together, and the sum is stored as the new value of each data item referenced by *identifier-3*.

Format 3: ADD statement with CORRESPONDING phrase



Elementary data items within *identifier-1* are added to and stored in the corresponding elementary items within *identifier-2*.

For all formats:

identifier-1, identifier-2

In format 1, must name an elementary numeric item.

In format 2, must name an elementary numeric item except when following the word GIVING. Each identifier that follows the word GIVING must name an elementary numeric or numeric-edited item.

In format 3, must name an alphanumeric group item or national group item.

The following restrictions apply to date fields:

- In format 1, *identifier-2* can specify one or more date fields. *identifier-1* must not specify a date field.
- In format 2, either *identifier-1* or *identifier-2* (but not both) can specify at most one date field. If *identifier-1* or *identifier-2* specifies a date field, then every instance of *identifier-3* must specify a date field that is compatible with the date field specified by *identifier-1* or *identifier-2*. That is, they must have the same date format, except for the year part, which can be windowed or expanded.

If neither *identifier-1* nor *identifier-2* specifies a date field, *identifier-3* can specify one or more date fields without any restriction on the date formats.

- In format 3, only corresponding elementary items within *identifier-2* can be date fields. There is no restriction on the format of these date fields.
- A year-last date field is allowed in an ADD statement only as *identifier-1* and when the result of the addition is a nondate.

There are two steps to determining the result of an ADD statement that involves one or more date fields:

1. Addition: determine the result of the addition operation, as described under “Addition that involves date fields” on page 260.
2. Storage: determine how the result is stored in the receiving field. (In formats 1 and 3, the receiving field is *identifier-2*; in Format 3, the receiving field is the GIVING *identifier-3*.) For details, see “Storing arithmetic results that involve date fields” on page 261.

literal

Must be a numeric literal.

Floating-point data items and literals can be used anywhere that a numeric data item or literal can be specified.

When the ARITH(COMPAT) compiler option is in effect, the composite of operands can contain a maximum of 30 digits. When the ARITH(EXTEND) compiler option is in effect, the composite of operands can contain a maximum of 31 digits. For more information, see “Arithmetic statement operands” on page 294 and the details on arithmetic intermediate results in *Appendix A. Intermediate results and arithmetic precision* in the *COBOL for AIX Programming Guide*.

ROUNDED phrase

For formats 1, 2, and 3, see “ROUNDED phrase” on page 292.

SIZE ERROR phrases

For formats 1, 2, and 3, see “SIZE ERROR phrases” on page 293.

CORRESPONDING phrase (format 3)

See “CORRESPONDING phrase” on page 291.

END-ADD phrase

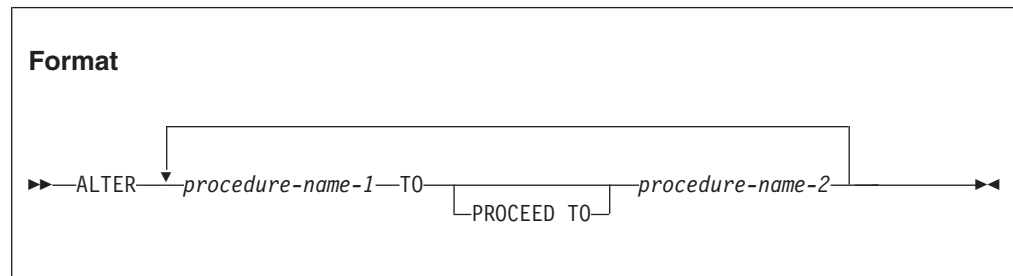
This explicit scope terminator serves to delimit the scope of the ADD statement. END-ADD permits a conditional ADD statement to be nested in another conditional statement. END-ADD can also be used with an imperative ADD statement.

For more information, see “Delimited scope statements” on page 289.

ALTER statement

The ALTER statement changes the transfer point specified in a GO TO statement.

The ALTER statement encourages the use of unstructured programming practices; the EVALUATE statement provides the same function as the ALTER statement but helps to ensure that a program is well-structured.



The ALTER statement modifies the GO TO statement in the paragraph named by *procedure-name-1*. Subsequent executions of the modified GO TO statement transfer control to *procedure-name-2*.

procedure-name-1

Must name a procedure division paragraph that contains only one sentence: a GO TO statement without the DEPENDING ON phrase.

procedure-name-2

Must name a procedure division section or paragraph.

Before the ALTER statement is executed, when control reaches the paragraph specified in *procedure-name-1*, the GO TO statement transfers control to the paragraph specified in the GO TO statement. After execution of the ALTER statement however, the next time control reaches the paragraph specified in *procedure-name-1*, the GO TO statement transfers control to the paragraph specified in *procedure-name-2*.

The ALTER statement acts as a program switch, allowing, for example, one sequence of execution during initialization and another sequence during the bulk of file processing.

Altered GO TO statements in programs with the INITIAL attribute are returned to their initial states each time the program is entered.

Do not use the ALTER statement in programs that have the RECURSIVE attribute, in methods, or in programs compiled with the THREAD option.

Segmentation considerations

A GO TO statement that is coded in an independent segment must not be referenced by an ALTER statement in a segment with a different *priority-number*. All other uses of the ALTER statement are valid and are performed even if the GO TO referenced by the ALTER statement is in a fixed segment.

Altered GO TO statements in independent segments are returned to their initial state when control is transferred to the independent segment that contains the ALTERED GO TO from another independent segment with a different *priority-number*.

This transfer of control can take place because of:

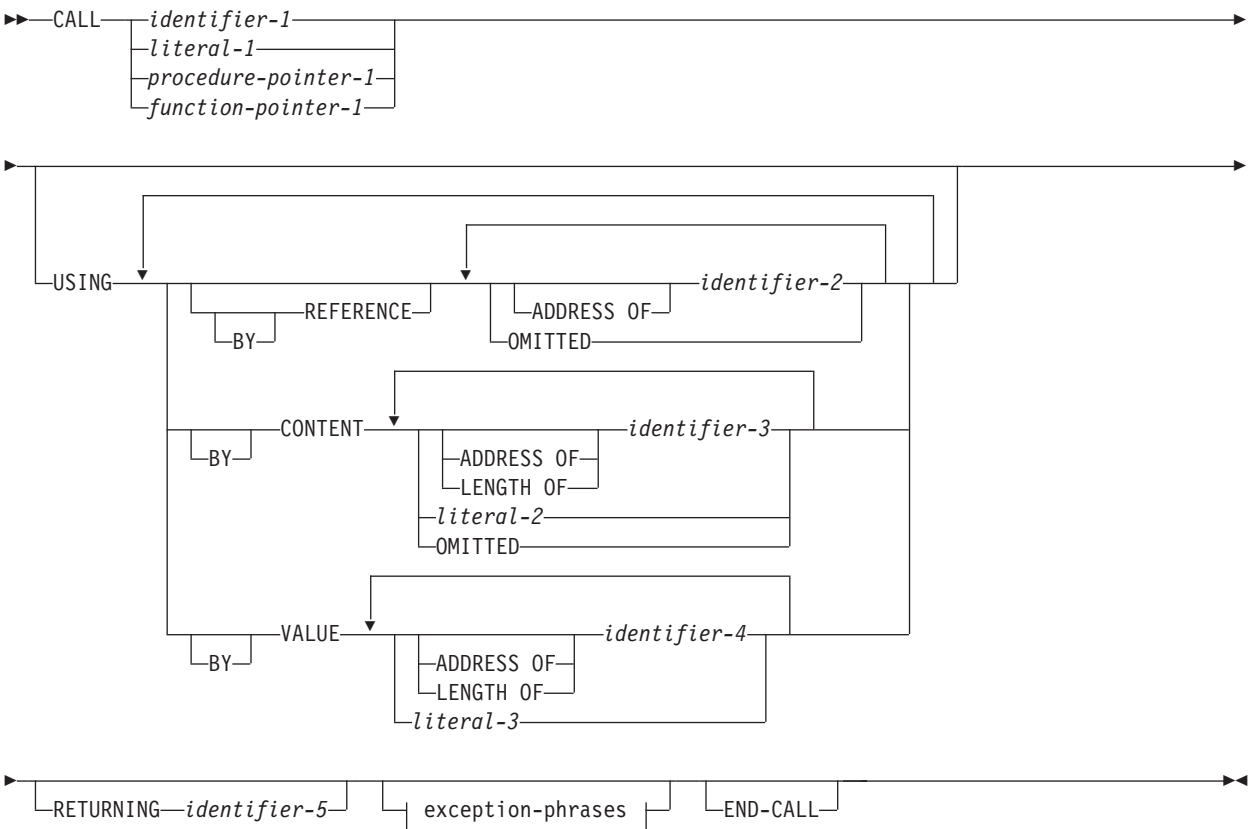
- The effect of previous statements
- An explicit transfer of control with a PERFORM or GO TO statement
- A sort or merge statement with the INPUT or OUTPUT phrase specified

CALL statement

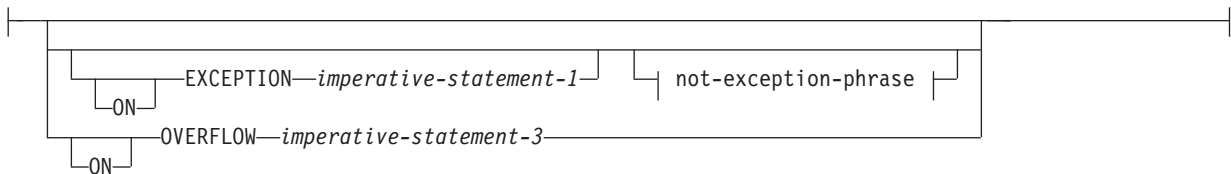
The CALL statement transfers control from one object program to another within the run unit.

The program containing the CALL statement is the calling program; the program identified in the CALL statement is the called subprogram. Called programs can contain CALL statements; however, only programs defined with the RECURSIVE clause can execute a CALL statement that directly or indirectly calls itself.

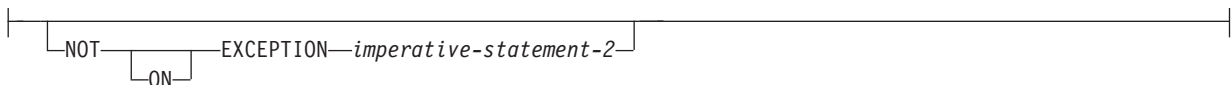
Format



exception-phrases:



not-exception-phrase:



identifier-1, literal-1

literal-1 must be an alphanumeric literal. *identifier-1* must be an alphanumeric, alphabetic, or numeric data item described with USAGE DISPLAY such that its value can be a program-name.

The rules of formation for program-names are dependent on the PGMNAME compiler option. For details, see the discussion of program-names in “PROGRAM-ID paragraph” on page 98 and also the description of PGMNAME in the *COBOL for AIX Programming Guide*.

identifier-1 cannot be a windowed date field.

Usage note: Do not specify the name of a class or method in the CALL statement.

procedure-pointer-1

Must be defined with USAGE IS PROCEDURE-POINTER and must be set to a valid program entry point; otherwise, the results of the CALL statement are undefined.

After a program has been canceled by COBOL, released by PL/I or C, or deleted by assembler, any procedure-pointers that had been set to that program's entry point are no longer valid.

function-pointer-1

Must be defined with USAGE IS FUNCTION-POINTER and must be set to a valid function or program entry point; otherwise, the results of the CALL statement are undefined.

After a program has been canceled by COBOL, released by PL/I or C, or deleted by the assembler, any function-pointers that had been set to that function or program's entry point are no longer valid.

When the called subprogram is to be entered at the beginning of the procedure division, *literal-1* or the contents of *identifier-1* must specify the program-name of the called subprogram.

When the called subprogram is entered through an ENTRY statement, *literal-1* or the contents of *identifier-1* must be the same as the name specified in the called subprogram's ENTRY statement.

For information about how the compiler resolves calls to program-names found in multiple programs, see "Conventions for program-names" on page 84.

USING phrase

The USING phrase specifies arguments that are passed to the target program.

Include the USING phrase in the CALL statement only if there is a USING phrase in the procedure division header or the ENTRY statement through which the called program is run. The number of operands in each USING phrase must be identical.

For more information about the USING phrase, see "The procedure division header" on page 251.

The sequence of the operands in the USING phrase of the CALL statement and in the corresponding USING phrase in the called subprogram's procedure division header or ENTRY statement determines the correspondence between the operands used by the calling and called programs. This correspondence is positional.

The values of the parameters referenced in the USING phrase of the CALL statement are made available to the called subprogram at the time the CALL statement is executed. The description of the data items in the called program must describe the same number of character positions as the description of the corresponding data items in the calling program.

The BY CONTENT, BY REFERENCE, and BY VALUE phrases apply to parameters that follow them until another BY CONTENT, BY REFERENCE, or BY VALUE

phrase is encountered. BY REFERENCE is assumed if you do not specify a BY CONTENT, BY REFERENCE, or BY VALUE phrase prior to the first parameter.

BY REFERENCE phrase

If the BY REFERENCE phrase is either specified or implied for a parameter, the corresponding data item in the calling program occupies the same storage area as the data item in the called program.

identifier-2

Can be any data item of any level in the data division. *identifier-2* cannot be a function-identifier.

If it is defined in the linkage section or file section, you must have already provided addressability for *identifier-2* prior to invocation of the CALL statement. You can do this by coding either one of the following: SET ADDRESS OF *identifier-2* TO pointer or PROCEDURE/ENTRY USING.

ADDRESS OF *identifier-2*

identifier-2 must be a level-01 or level-77 item defined in the linkage section.

OMITTED

Indicates that no argument is passed.

BY CONTENT phrase

If the BY CONTENT phrase is specified or implied for a parameter, the called program cannot change the value of this parameter as referenced in the CALL statement's USING phrase, though the called program can change the value of the data item referenced by the corresponding data-name in the called program's procedure division header. Changes to the parameter in the called program do not affect the corresponding argument in the calling program.

identifier-3

Can be any data item of any level in the data division. *identifier-3* cannot be a function identifier.

If defined in the linkage section or file section, you must have already provided addressability for *identifier-3* prior to invocation of the CALL statement. You can do this by coding one of the following:

- SET ADDRESS OF *identifier-3* TO pointer
- PROCEDURE DIVISION USING
- ENTRY . . . USING

literal-2

Can be:

- An alphanumeric literal
- A figurative constant (except ALL *literal* or NULL/NULLS)
- A DBCS literal
- A national literal

LENGTH OF special register

For information about the LENGTH OF special register, see "LENGTH OF" on page 19.

ADDRESS OF *identifier-3*

identifier-3 must be a data item of any level except 66 or 88 defined in the linkage section, the working-storage section, or the local-storage section.

OMITTED

Indicates that no argument is passed.

For alphanumeric literals, the called subprogram should describe the parameter as PIC X(*n*) USAGE DISPLAY, where *n* is the number of characters in the literal.

For DBCS literals, the called subprogram should describe the parameter as PIC G(*n*) USAGE DISPLAY-1, or PIC N(*n*) with implicit or explicit USAGE DISPLAY-1, where *n* is the length of the literal.

For national literals, the called subprogram should describe the parameter as PIC N(*n*) with implicit or explicit USAGE NATIONAL, where *n* is the length of the literal.

BY VALUE phrase

The BY VALUE phrase applies to all arguments that follow until overridden by another BY REFERENCE or BY CONTENT phrase.

If the BY VALUE phrase is specified or implied for an argument, the value of the argument is passed, not a reference to the sending data item. The called program can modify the formal parameter that corresponds to the BY VALUE argument, but any such changes do not affect the argument because the called program has access to a temporary copy of the sending data item.

Although BY VALUE arguments are primarily intended for communication with non-COBOL programs (such as C), they can also be used for COBOL-to-COBOL invocations. In this case, BY VALUE must be specified or implied for both the argument in the CALL USING phrase and the corresponding formal parameter in the procedure division USING phrase.

identifier-4

Must be an elementary data item in the data division. It must be one of the following:

- Binary (USAGE BINARY, COMP, COMP-4, or COMP-5)
- Floating point (USAGE COMP-1 or COMP-2)
- Function-pointer (USAGE FUNCTION-POINTER)
- Pointer (USAGE POINTER)
- Procedure-pointer (USAGE PROCEDURE-POINTER)
- Object reference (USAGE OBJECT REFERENCE)
- One single-byte alphanumeric character (such as PIC X or PIC A)
- One national character (PIC N), described as an elementary data item of category national.

The following can also be passed BY VALUE:

- Reference-modified item of usage display and length 1
- Reference-modified item of usage national and length 1
- SHIFT-IN and SHIFT-OUT special registers
- LINAGE-COUNTER special register when it is usage binary

ADDRESS OF *identifier-4*

identifier-4 must be a data item of any level except 66 or 88 defined in the linkage section, the working-storage section, or the local-storage section.

LENGTH OF special register

A LENGTH OF special register passed BY VALUE is treated as a PIC 9(9) binary when the ADDR(32) compiler option is specified, or as a PIC 9(18) binary when the ADDR(64) compiler option is specified. For information about the LENGTH OF special register, see “LENGTH OF” on page 19.

literal-3

Must be one of the following:

- A numeric literal
- A figurative constant ZERO
- A one-character alphanumeric literal
- A one-character national literal
- A symbolic character
- A single-byte figurative constant
 - SPACE
 - QUOTE
 - HIGH-VALUE
 - LOW-VALUE

ZERO is treated as a numeric value; a fullword binary zero is passed.

If *literal-3* is a fixed-point numeric literal, it must have a precision of nine or fewer digits. In this case, a fullword binary representation of the literal value is passed.

If *literal-3* is a floating-point numeric literal, an 8-byte internal floating-point (COMP-2) representation of the value is passed.

literal-3 must not be a DBCS literal.

RETURNING phrase

identifier-5

The RETURNING data item, which can be any data item defined in the data division. The return value of the called program is implicitly stored into *identifier-5*.

You can specify the RETURNING phrase for calls to functions written in COBOL, C, or in other programming languages that use C linkage conventions. If you specify the RETURNING phrase on a CALL to a COBOL subprogram:

- The called subprogram must specify the RETURNING phrase on its procedure division header.
- *identifier-5* and the corresponding procedure division RETURNING identifier in the target program must have the same PICTURE, USAGE, SIGN, SYNCHRONIZE, JUSTIFIED, and BLANK WHEN ZERO clauses (except that PICTURE clause currency symbols can differ, and periods and commas can be interchanged due to the DECIMAL POINT IS COMMA clause).

When the target returns, its return value is assigned to *identifier-5* using the rules for the SET statement if *identifier-6* is of usage INDEX, POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE.

When *identifier-5* is of any other usage, the rules for the MOVE statement are used.

The CALL ... RETURNING data item is an output-only parameter. On entry to the called program, the initial state of the PROCEDURE DIVISION RETURNING data item has an undefined and unpredictable value. You must initialize the PROCEDURE DIVISION RETURNING data item in the called program before you reference its value. The value that is passed back to the calling program is the final value of the PROCEDURE DIVISION RETURNING data item when the called program returns.

If an EXCEPTION or OVERFLOW occurs, *identifier-5* is not changed. *identifier-5* must not be reference-modified.

The RETURN-CODE special register is not set by execution of CALL statements that include the RETURNING phrase.

ON EXCEPTION phrase

An exception condition occurs when the called subprogram cannot be made available. At that time, one of the following two actions will occur:

1. If the ON EXCEPTION phrase is specified, control is transferred to *imperative-statement-1*. Execution then continues according to the rules for each statement specified in *imperative-statement-1*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement. Otherwise, upon completion of the execution of *imperative-statement-1*, control is transferred to the end of the CALL statement and the NOT ON EXCEPTION phrase, if specified, is ignored.
2. If the ON EXCEPTION phrase is not specified in the CALL statement, the NOT ON EXCEPTION phrase, if specified, is ignored.

NOT ON EXCEPTION phrase

If an exception condition does not occur (that is, the called subprogram can be made available), control is transferred to the called program. After control is returned from the called program, control is transferred to:

- *imperative-statement-2*, if the NOT ON EXCEPTION phrase is specified.
- The end of the CALL statement in any other case. (If the ON EXCEPTION phrase is specified, it is ignored.)

If control is transferred to *imperative-statement-2*, execution continues according to the rules for each statement specified in *imperative-statement-2*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement. Otherwise, upon completion of the execution of *imperative-statement-2*, control is transferred to the end of the CALL statement.

ON OVERFLOW phrase

The ON OVERFLOW phrase has the same effect as the ON EXCEPTION phrase.

END-CALL phrase

This explicit scope terminator serves to delimit the scope of the CALL statement. END-CALL permits a conditional CALL statement to be nested in another conditional statement. END-CALL can also be used with an imperative CALL statement.

For more information, see “Delimited scope statements” on page 289.

CANCEL statement

The CANCEL statement ensures that the referenced subprogram is entered in initial state the next time that it is called.

Format



identifier-1, literal-1

literal-1 must be an alphanumeric literal. *identifier-1* must be an alphanumeric, alphabetic, or zoned decimal data item such that its value can be a program-name. The rules of formation for program-names are dependent on the PGMNAME compiler option. For details, see the discussion of program-names in “PROGRAM-ID paragraph” on page 98 and the description of PGMNAME in the *COBOL for AIX Programming Guide*.

identifier-1 cannot be a windowed date field.

literal-1 or the contents of *identifier-1* must be the same as a literal or the contents of an identifier specified in an associated CALL statement.

Do not specify the name of a class or a method in the CANCEL statement.

After a CANCEL statement for a called subprogram has been executed, that subprogram no longer has a logical connection to the program. The contents of data items in external data records described by the subprogram are not changed when that subprogram is canceled. If a CALL statement is executed later by any program in the run unit naming the same subprogram, that subprogram is entered in its initial state.

When a CANCEL statement is executed, all programs contained within the program referenced in the CANCEL statement are also canceled. The result is the same as if a valid CANCEL were executed for each contained program in the reverse order in which the programs appear in the separately compiled program.

A CANCEL statement closes all open files that are associated with an internal file connector in the program named in an explicit CANCEL statement. USE procedures associated with those files are not executed.

You can cancel a called subprogram in any of the following ways:

- By referencing it as the operand of a CANCEL statement
- By terminating the run unit of which the subprogram is a member
- By executing an EXIT PROGRAM statement or a GOBACK statement in the called subprogram if that subprogram possesses the initial attribute

No action is taken when a CANCEL statement is executed if the specified program:

- Has not been called in this run unit by another IBM COBOL program.
- Has been called and subsequently canceled

In a multithreaded environment, a program cannot execute a CANCEL statement naming a program that is active on any thread. The named program must be completely inactive.

Called subprograms can contain CANCEL statements. However, a called subprogram must not execute a CANCEL statement that directly or indirectly cancels the calling program itself or that cancels any program higher than itself in the calling hierarchy. In such a case, the run unit is terminated.

A program named in a CANCEL statement must be a program that has been called and has executed an EXIT PROGRAM statement or a GOBACK statement.

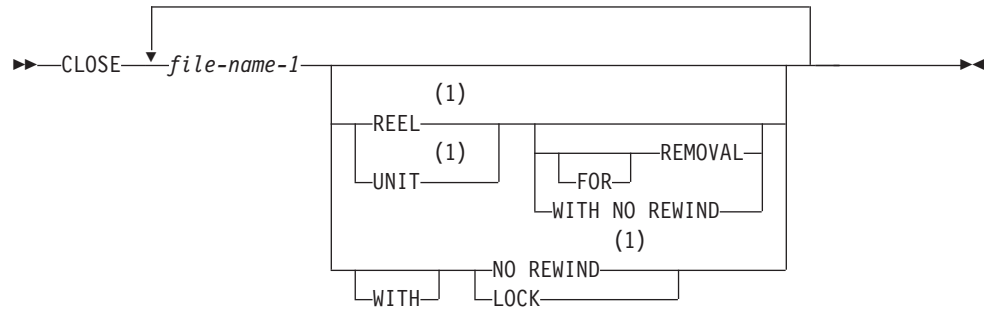
A program can cancel a program that it did not call, provided that, in the calling hierarchy, the program that executes the CANCEL statement is higher than or equal to the program it is canceling. For example:

A calls B and B calls C	(When A receives control, it can cancel C.)
A calls B and A calls C	(When C receives control, it can cancel B.)

CLOSE statement

The CLOSE statement terminates the processing of volumes and files.

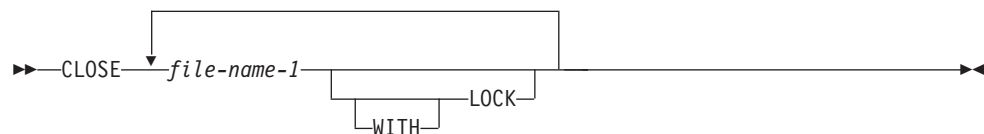
Format 1: CLOSE statement for sequential files



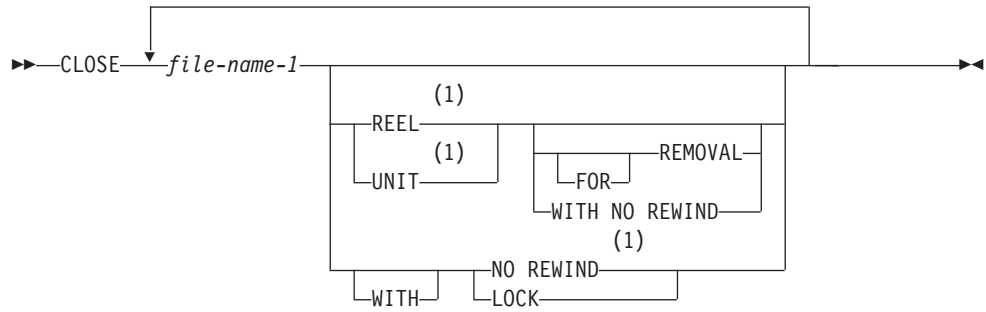
Notes:

- 1 The UNIT, REEL, and NO REWIND phases are treated as a comment, although the file status will be set to 07, indicating a successful completion of a CLOSE for a non-reel/unit medium.

Format 2: CLOSE statement for indexed and relative files



Format 3: CLOSE statement for line-sequential files



Notes:

- 1 The UNIT, REEL, and NO REWIND phases are treated as a comment, although the file status will be set to 07, indicating a successful completion of a CLOSE for a non-reel/unit medium.

file-name-1

Designates the file upon which the CLOSE statement is to operate. If more than one file-name is specified, the files need not have the same organization or access. *file-name-1* must not be a sort or merge file.

REEL and UNIT

REEL and UNIT are syntax checked, but have no effect on the execution of the program.

WITH NO REWIND and FOR REMOVAL

WITH NO REWIND and FOR REMOVAL are syntax checked, but have no effect on the execution of the program.

A CLOSE statement can be executed only for a file in an open mode. After successful execution of a CLOSE statement (without the REEL/UNIT phrase if using format 1):

- The record area associated with the file-name is no longer available. Unsuccessful execution of a CLOSE statement leaves availability of the record data undefined.
- An OPEN statement for the file must be executed before any other input/output statement can be executed for the file and before data is moved to a record description entry associated with the file.
- Any record locks and file locks held by the file connector on the closed file are released.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the CLOSE statement is executed.

If the file is in an open status and the execution of a CLOSE statement is unsuccessful, the EXCEPTION/ERROR procedure (if specified) for this file is executed.

Effect of CLOSE statement on file types

If the SELECT OPTIONAL clause is specified in the file-control entry for a file, and the file is not available at run time, standard end-of-file processing is not performed.

Files are divided into the following types:

Non-reel/unit

A file whose input or output medium is such that rewinding, reels, and units have no meaning. All SdU, SFS, RSD, DB2, and STL files are non-reel/unit file types.

Sequential single volume

A sequential file that is contained entirely on one volume. More than one file can be contained on this volume. All SdU, SFS, RSD, DB2, and STL files are single volume.

Sequential multivolume

A sequential file that is contained on more than one volume. The concept of volume has no meaning for SdU, SFS, RSD, DB2, or STL files.

The permissible combinations of CLOSE statement phrases are shown in the following tables:

- For sequential files: Sequential files and CLOSE statement phrases
- For indexed and relative files: Table 41
- For line-sequential files: Table 42

The meaning of each key letter is shown in Table 43 on page 327.

Table 40. Sequential files and CLOSE statement phrases

CLOSE statement phrases	Non-reel/ unit	Sequential single-volume	Sequential multivolume
CLOSE	C	C, G	A, C, G
CLOSE WITH LOCK	C, E	C, E, G	A, C, E, G

Table 41. Indexed and relative file types and CLOSE statement phrases

CLOSE statement phrases	Action
CLOSE	C
CLOSE WITH LOCK	C,E

Table 42. Line-sequential file types and CLOSE statement phrases

CLOSE statement phrases	Action
CLOSE	C
CLOSE WITH LOCK	C,E

Table 43. Meanings of key letters for sequential file types

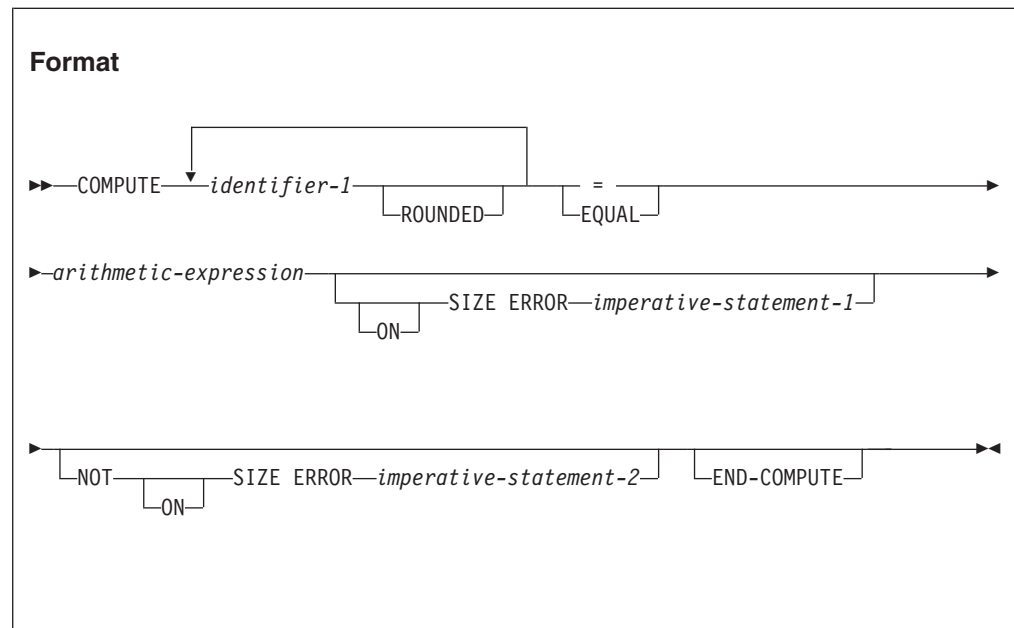
Key	Actions taken
A	<p>Previous volumes unaffected</p> <p>Input and input-output files: Standard volume-switch processing is performed for all previous volumes (except those controlled by a previous CLOSE REEL/UNIT statement). Any subsequent volumes are not processed.</p> <p>Output files: Standard volume-switch processing is performed for all previous volumes (except those controlled by a previous CLOSE REEL/UNIT statement).</p>
C	<p>Close file</p> <p>Input and input-output files: If the file is at its end, and label records are specified, the standard ending label procedure is performed. Standard system closing procedures are then performed.</p> <p>If the file is at its end, and label records are not specified, label processing does not take place, but standard system closing procedures are performed.</p> <p>If the file is not at its end, standard system closing procedures are performed, but there is no ending label processing.</p> <p>Output files: If label records are specified, standard ending label procedures are performed. Standard system closing procedures are then performed.</p> <p>If label records are not specified, ending label procedures are not performed, but standard system closing procedures are performed.</p>
E	<p>File lock: The compiler ensures that this file cannot be opened again during this execution of the object program. If the file is a tape unit, it will be rewound and unloaded.</p>
G	<p>Rewind: The current volume is positioned at its physical beginning.</p>

COMPUTE statement

The COMPUTE statement assigns the value of an arithmetic expression to one or more data items.

With the COMPUTE statement, arithmetic operations can be combined without the restrictions on receiving data items imposed by the rules for the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements.

When arithmetic operations are combined, the COMPUTE statement can be more efficient than the separate arithmetic statements written in a series.



identifier-1

Must name an elementary numeric item or an elementary numeric-edited item.

Can name an elementary floating-point data item.

If *identifier-1* or the result of *arithmetic expression* (or both) are date fields, see “Storing arithmetic results that involve date fields” on page 261 for details on how the result is stored in *identifier-1*. If a year-last date field is specified as *identifier-1*, the result of *arithmetic expression* must be a nondate.

arithmetic-expression

Can be any arithmetic expression, as defined in “Arithmetic expressions” on page 257.

When the COMPUTE statement is executed, the value of *arithmetic expression* is calculated and stored as the new value of each data item referenced by *identifier-1*.

An arithmetic expression consisting of a single identifier, numeric function, or literal allows the user to set the value of the data items that are referenced by *identifier-1* equal to the value of that identifier, function, or literal.

A year-last date field must not be specified in the arithmetic expression.

ROUNDED phrase

For a discussion of the ROUNDED phrase, see “ROUNDED phrase” on page 292.

SIZE ERROR phrases

For a discussion of the SIZE ERROR phrases, see “SIZE ERROR phrases” on page 293.

END-COMPUTE phrase

This explicit scope terminator serves to delimit the scope of the COMPUTE statement. END-COMPUTE permits a conditional COMPUTE statement to be nested in another conditional statement. END-COMPUTE can also be used with an imperative COMPUTE statement.

For more information, see “Delimited scope statements” on page 289.

CONTINUE statement

The CONTINUE statement is a no operation statement. CONTINUE indicates that no executable instruction is present.

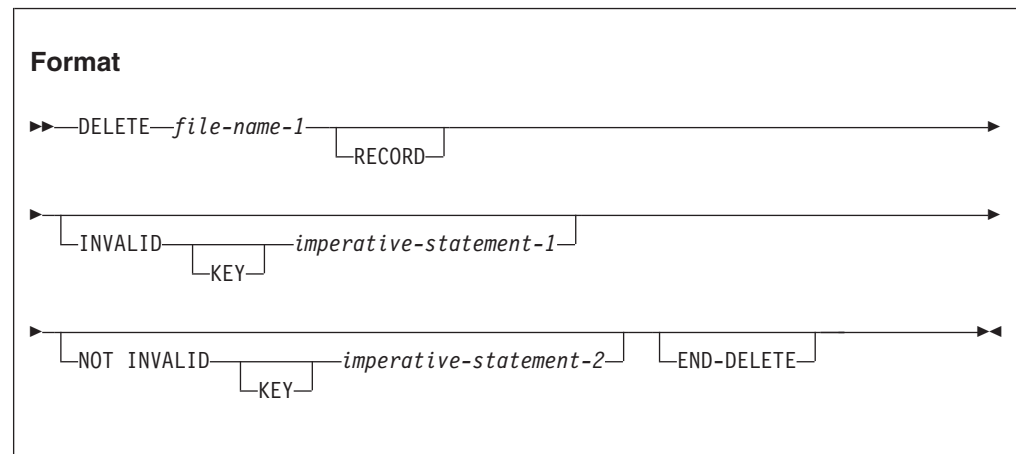
Format

►►—CONTINUE—►◄

DELETE statement

The DELETE statement removes a record from an indexed or relative file. For indexed files, the key can then be reused for record addition. For relative files, the space is then available for a new record with the same RELATIVE KEY value.

When the DELETE statement is executed, the associated file must be open in I-O mode.



file-name-1

Must be defined in an FD entry in the data division and must be the name of an indexed or relative file.

After successful execution of a DELETE statement, the record is removed from the file and can no longer be accessed.

Execution of the DELETE statement does not affect the contents of the record area associated with *file-name-1* or the content of the data item referenced by the data-name specified in the DEPENDING ON phrase of the RECORD clause associated with *file-name-1*.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the DELETE statement is executed.

The file position indicator is not affected by execution of the DELETE statement.

Sequential access mode

For a file in sequential access mode, the previous input/output statement must be a successfully executed READ statement. When the DELETE statement is executed, the system removes the record that was retrieved by that READ statement.

For a file in sequential access mode, the INVALID KEY and NOT INVALID KEY phrases must not be specified. An EXCEPTION/ERROR procedure can be specified.

Random or dynamic access mode

In random or dynamic access mode, DELETE statement execution results depend on the file organization: indexed or relative.

When the DELETE statement is executed, the system removes the record identified by the contents of the prime RECORD KEY data item for indexed files, or the RELATIVE KEY data item for relative files. If the file does not contain such a record, an INVALID KEY condition exists. (See “Invalid key condition” on page 300.)

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

Transfer of control after the successful execution of a DELETE statement, with the NOT INVALID KEY phrase specified, is to the imperative statement associated with the phrase.

END-DELETE phrase

This explicit scope terminator serves to delimit the scope of the DELETE statement. END-DELETE permits a conditional DELETE statement to be nested in another conditional statement. END-DELETE can also be used with an imperative DELETE statement.

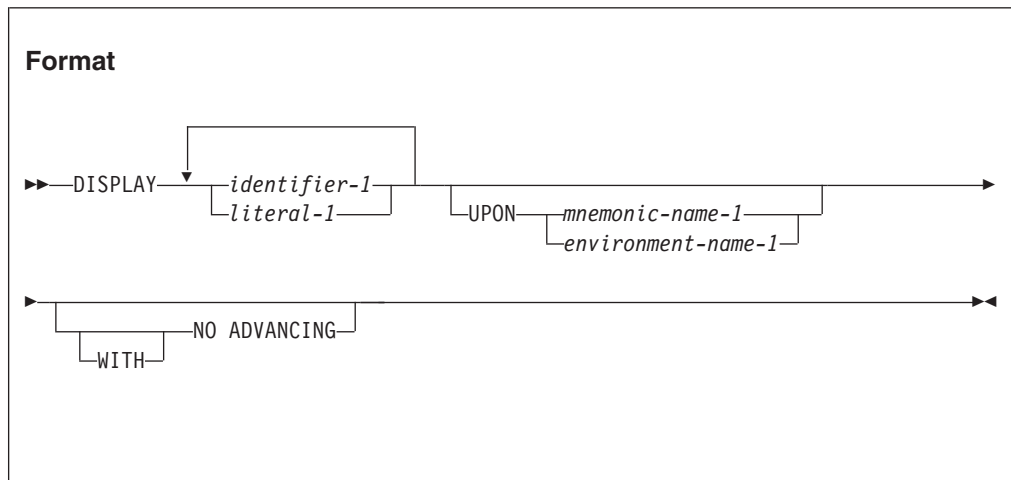
For more information, see “Delimited scope statements” on page 289.

DISPLAY statement

The DISPLAY statement transfers the contents of each operand to the output device. The contents are displayed on the output device in the order, left to right, in which the operands are listed.

The target file is determined by checking the COBOL environment-name (CONSOLE, SYSIN, SYSIPT, SYSOUT, SYSLIST, SYSLST, SYSPUNCH, and SYSPCH). If an environment variable is defined corresponding to the COBOL environment-name, the value of the environment-variable is used as the system file identifier. If the environment variable that corresponds to the COBOL environment name is not set, DISPLAY on SYSOUT, SYSLIST, or SYSLST is to the system logical output device (stdout).

For SYSPUNCH and SYSPCH, the DISPLAY statement fails unless the corresponding environment variable is set to point to a valid target. For more information about environment variables, see *Setting and accessing environment variables* in the *COBOL for AIX Programming Guide*.



identifier-1

Identifier-1 references the data that is to be displayed. *Identifier-1* can reference any data item except an item of usage PROCEDURE-POINTER, FUNCTION-POINTER, OBJECT REFERENCE, or INDEX. *Identifier-1* cannot be an index-name.

If *identifier-1* is a binary, internal decimal, or internal floating-point data item, *identifier-1* is converted automatically to external format as follows:

- Binary and internal decimal items are converted to zoned decimal. Negative signed values cause a low-order sign overpunch.
- Internal floating-point numbers are converted to external floating-point numbers for display such that:
 - A COMP-1 item will display as if it had an external floating-point PICTURE clause of `-.9(8)E-99`.
 - A COMP-2 item will display as if it had an external floating-point PICTURE clause of `-.9(17)E-99`.

Data items defined with USAGE POINTER are converted to a zoned decimal number that has an implicit PICTURE clause of `PIC 9(10)`.

Data items described with usage NATIONAL are converted to the code page associated with the current locale.

No other categories of data require conversion.

Date fields are treated as nondates when specified in a DISPLAY statement. That is, the DATE FORMAT is ignored and the content of the data item is transferred to the output device as is.

DBCS data items, explicitly or implicitly defined as USAGE DISPLAY-1, are transferred to the sending field of the output device.

Both DBCS and non-DBCS operands can be specified in a single DISPLAY statement.

literal-1

Can be any literal or any figurative constant as specified in “Figurative constants” on page 13. When a figurative constant is specified, only a single occurrence of that figurative constant is displayed.

UPON

environment-name-1 or the environment name associated with *mnemonic-name-1* must be associated with an output device. See “SPECIAL-NAMES paragraph” on page 110.

When the UPON phrase is omitted, the system's logical output device is assumed. The list of valid environment-names in a DISPLAY statement is shown in Table 5 on page 112.

WITH NO ADVANCING

When specified, the positioning of the output device will not be changed in any way following the display of the last operand. If the output device is capable of positioning to a specific character position, it will remain positioned at the character position immediately following the last character of the last operand displayed. If the output device is not capable of positioning to a specific character position, only the vertical position, if applicable, is affected. This can cause overprinting.

If the WITH NO ADVANCING phrase is not specified, after the last operand has been transferred to the output device, the positioning of the output device will be reset to the leftmost position of the next line of the device.

The DISPLAY statement transfers the data in the sending field to the output device. The size of the sending field is the total byte count of all operands listed. If the output device is capable of receiving data of the same size as the data item being transferred, then the data item is transferred. If the output device is not capable of receiving data of the same size as the data item being transferred, then one of the following applies:

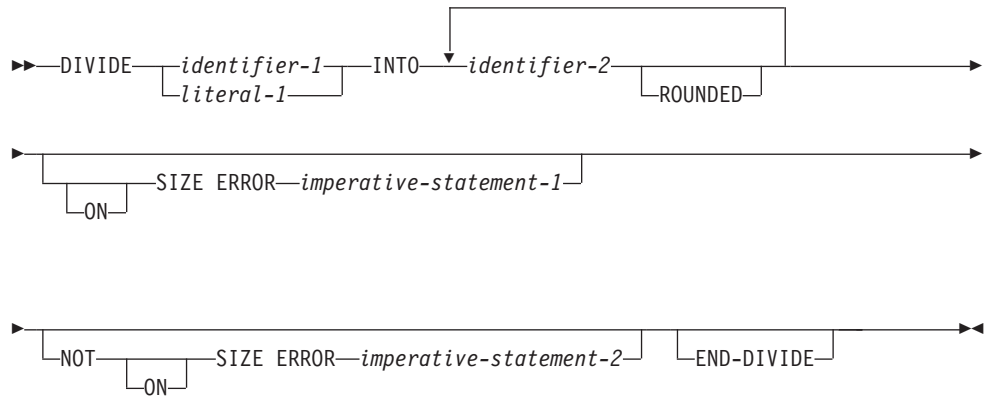
- If the total count is less than the device maximum, the remaining rightmost positions are padded with spaces.
- If the total count exceeds the maximum, as many records are written as are needed to display all operands. Any operand being printed or displayed when the end of a record is reached is continued in the next record.

If a DBCS operand must be split across multiple records, it will be split only on a double-byte boundary.

DIVIDE statement

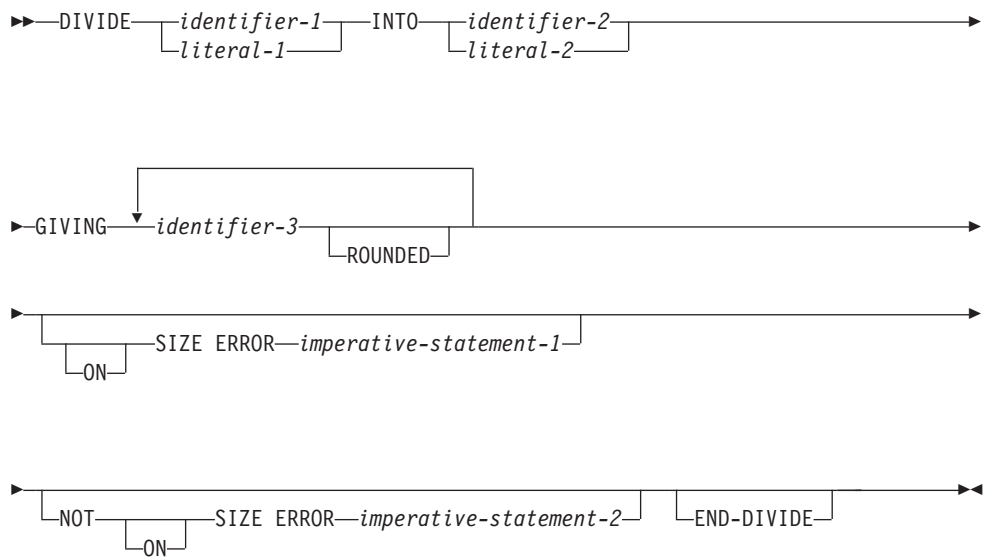
The DIVIDE statement divides one numeric data item into or by others and sets the values of data items equal to the quotient and remainder.

Format 1: DIVIDE statement



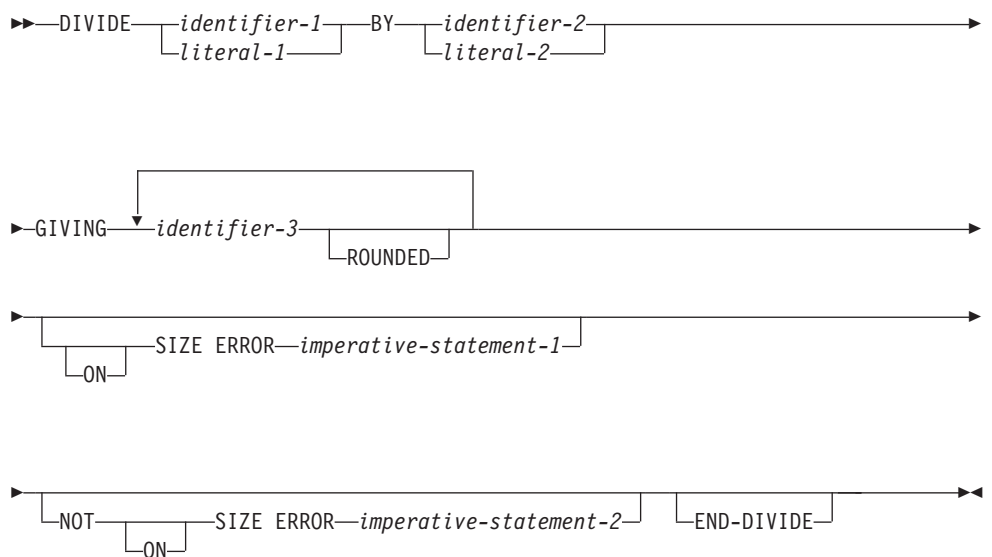
In format 1, the value of *identifier-1* or *literal-1* is divided into the value of *identifier-2*, and the quotient is then stored in *identifier-2*. For each successive occurrence of *identifier-2*, the division takes place in the left-to-right order in which *identifier-2* is specified.

Format 2: DIVIDE statement with INTO and GIVING phrases



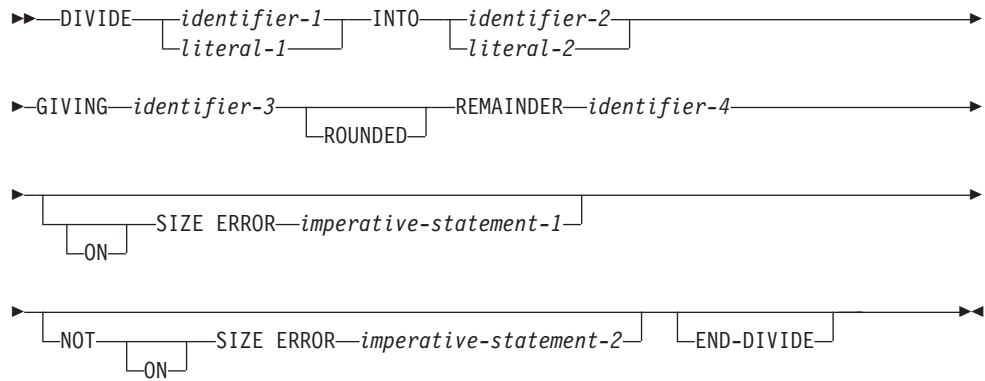
In format 2, the value of *identifier-1* or *literal-1* is divided into the value of *identifier-2* or *literal-2*. The value of the quotient is stored in each data item referenced by *identifier-3*.

Format 3: DIVIDE statement with BY and GIVING phrases



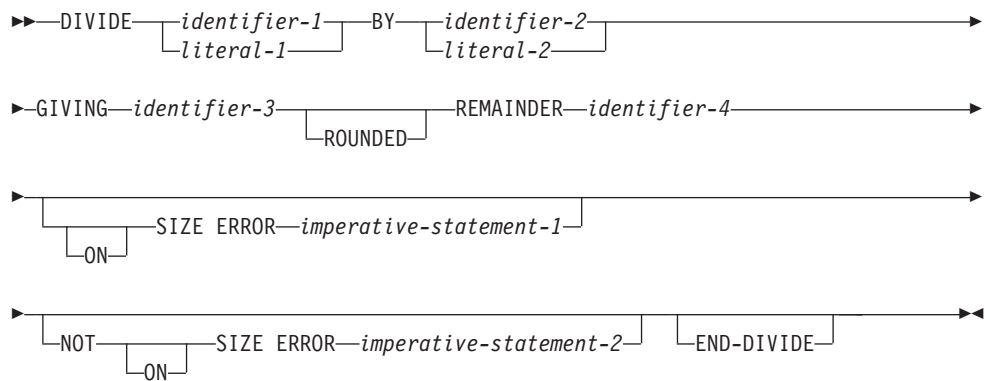
In format 3, the value of *identifier-1* or *literal-1* is divided by the value of *identifier-2* or *literal-2*. The value of the quotient is stored in each data item referenced by *identifier-3*.

Format 4: DIVIDE statement with INTO and REMAINDER phrases



In format 4, the value of *identifier-1* or *literal-1* is divided into *identifier-2* or *literal-2*. The value of the quotient is stored in *identifier-3*, and the value of the remainder is stored in *identifier-4*.

Format 5: DIVIDE statement with BY and REMAINDER phrases



In format 5, the value of *identifier-1* or *literal-1* is divided by *identifier-2* or *literal-2*. The value of the quotient is stored in *identifier-3*, and the value of the remainder is stored in *identifier-4*.

For all formats:

identifier-1, *identifier-2*

Must name an elementary numeric data item. *identifier-1* and *identifier-2* cannot be date fields.

identifier-3, identifier-4

Must name an elementary numeric or numeric-edited item.

If *identifier-3* or *identifier-4* is a date field, see “Storing arithmetic results that involve date fields” on page 261 for details on how the quotient or remainder is stored in *identifier-3*.

literal-1, literal-2

Must be a numeric literal.

In formats 1, 2, and 3, floating-point data items and literals can be used anywhere that a numeric data item or literal can be specified.

In formats 4 and 5, floating-point data items or literals cannot be used.

ROUNDED phrase

For formats 1, 2, and 3, see “ROUNDED phrase” on page 292.

For formats 4 and 5, the quotient used to calculate the remainder is in an intermediate field. The value of the intermediate field is truncated rather than rounded.

REMAINDER phrase

The result of subtracting the product of the quotient and the divisor from the dividend is stored in *identifier-4*. If *identifier-3*, the quotient, is a numeric-edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient.

The REMAINDER phrase is invalid if the receiver or any of the operands is a floating-point item.

Any subscripts for *identifier-4* in the REMAINDER phrase are evaluated after the result of the divide operation is stored in *identifier-3* of the GIVING phrase.

SIZE ERROR phrases

For formats 1, 2, and 3, see “SIZE ERROR phrases” on page 293.

For formats 4 and 5, if a size error occurs in the quotient, no remainder calculation is meaningful. Therefore, the contents of the quotient field (*identifier-3*) and the remainder field (*identifier-4*) are unchanged.

If size error occurs in the remainder, the contents of the remainder field (*identifier-4*) are unchanged.

In either of these cases, you must analyze the results to determine which situation has actually occurred.

For information about the NOT ON SIZE ERROR phrase, see “SIZE ERROR phrases” on page 293.

END-DIVIDE phrase

This explicit scope terminator serves to delimit the scope of the DIVIDE statement. END-DIVIDE turns a conditional DIVIDE statement into an imperative statement that can be nested in another conditional statement. END-DIVIDE can also be used with an imperative DIVIDE statement.

For more information, see “Delimited scope statements” on page 289.

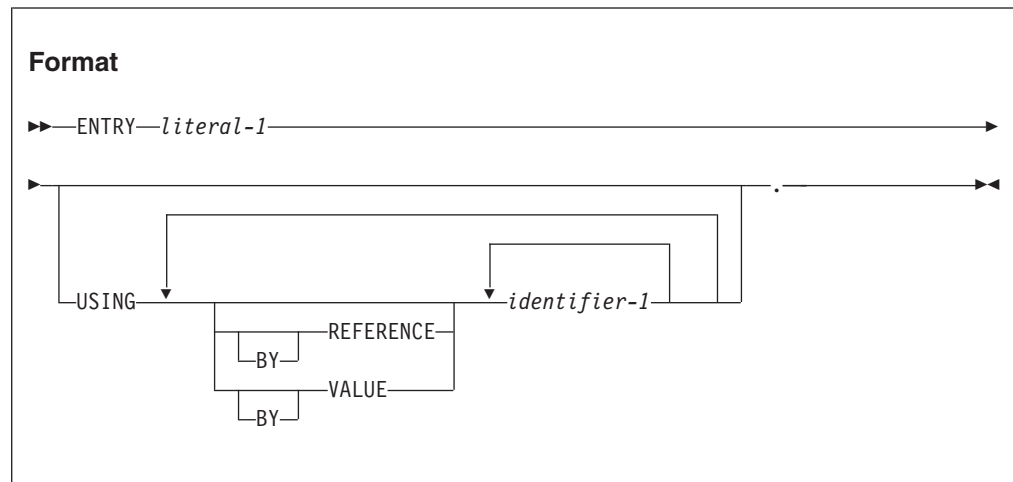
ENTRY statement

The ENTRY statement establishes an alternate entry point into a COBOL called subprogram.

The ENTRY statement cannot be used in:

- Programs that specify a return value using the procedure division RETURNING phrase. For details, see the discussion of the RETURNING phrase under “The procedure division header” on page 251.
- Nested program. See “Nested programs” on page 83 for a description of nested programs.

When a CALL statement that specifies the alternate entry point is executed in a calling program, control is transferred to the next executable statement following the ENTRY statement.



literal-1

Must be an alphanumeric literal that conform to the rules for the formation of a program-name in an outermost program (see “PROGRAM-ID paragraph” on page 98).

Must not match the program-ID or any other ENTRY literal in this program.

Must not be a figurative constant.

Execution of the called program begins at the first executable statement following the ENTRY statement whose literal corresponds to the literal or identifier specified in the CALL statement.

The entry point name on the ENTRY statement can be affected by the PGMNAME compiler option. For details, see *PGMNAME* in the *COBOL for AIX Programming Guide*.

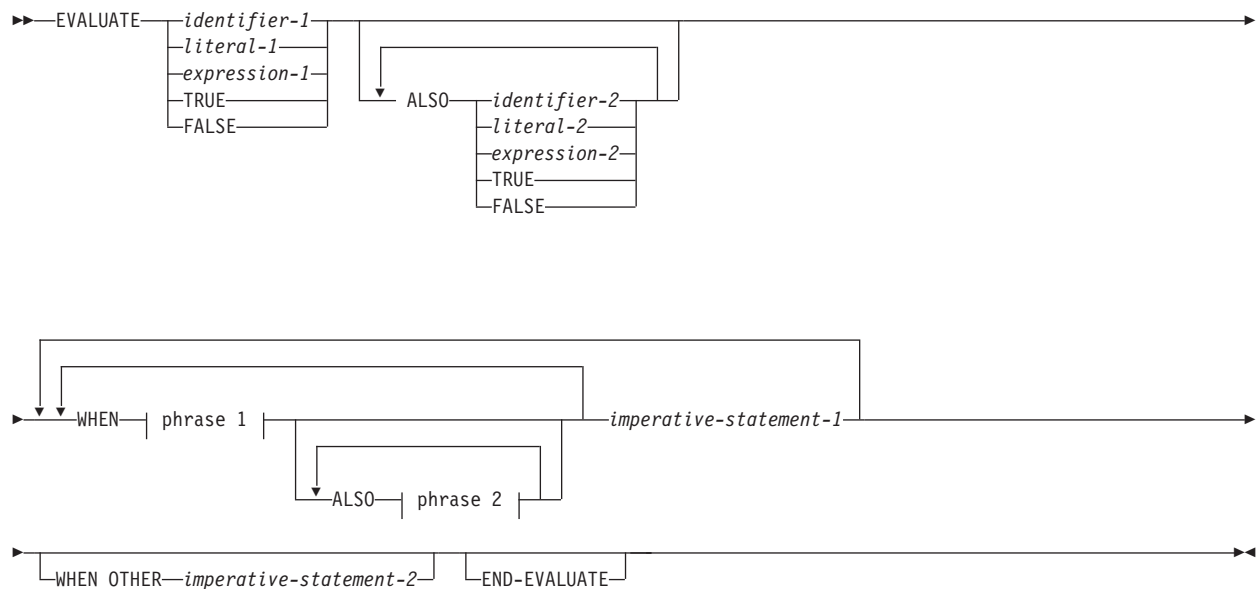
USING phrase

For a discussion of the USING phrase, see “The procedure division header” on page 251.

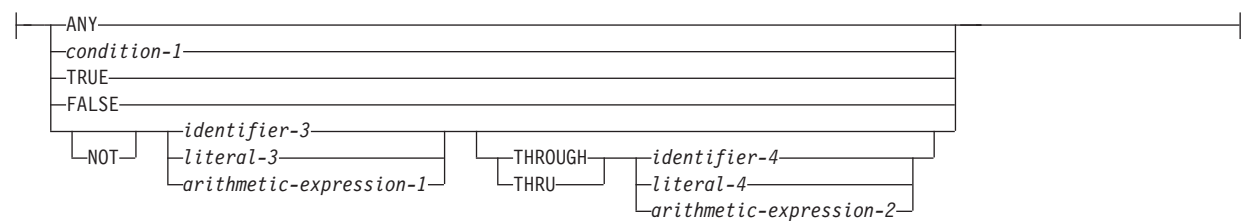
EVALUATE statement

The EVALUATE statement provides a shorthand notation for a series of nested IF statements. It can evaluate multiple conditions. The subsequent action depends on the results of these evaluations.

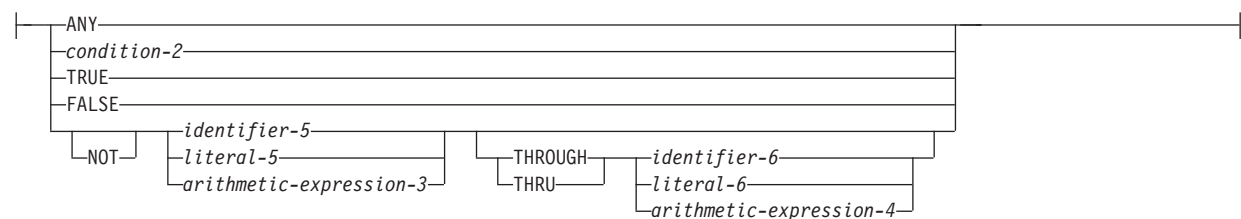
Format



phrase 1:



phrase 2:



Operands before the WHEN phrase

Are interpreted in one of two ways, depending on how they are specified:

- Individually, they are called selection *subjects*.
- Collectively, they are called a *set* of selection subjects.

Operands in the WHEN phrase

Are interpreted in one of two ways, depending on how they are specified:

- Individually, they are called selection *objects*.
- Collectively, they are called a *set* of selection objects.

ALSO

Separates selection subjects within a set of selection subjects; separates selection objects within a set of selection objects.

THROUGH and THRU

Are equivalent.

Two operands connected by a THRU phrase must be of the same class. The two operands thus connected constitute a single selection object.

The number of selection objects within each set of selection objects must be equal to the number of selection subjects.

Each selection object within a set of selection objects must correspond to the selection subject having the same ordinal position within the set of selection subjects, according to the following rules:

- Identifiers, literals, or arithmetic expressions appearing within a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects. For comparisons involving date fields, see “Comparison of date fields” on page 276.
- *condition-1*, *condition-2*, or the word TRUE or FALSE appearing as a selection object must correspond to a conditional expression or the word TRUE or FALSE in the set of selection subjects.
- The word ANY can correspond to a selection subject of any type.

END-EVALUATE phrase

This explicit scope terminator serves to delimit the scope of the EVALUATE statement. END-EVALUATE permits a conditional EVALUATE statement to be nested in another conditional statement.

For more information, see “Delimited scope statements” on page 289.

Determining values

The execution of the EVALUATE statement operates as if each selection subject and selection object were evaluated and assigned a numeric, alphanumeric, DBCS, or national character value; a range of numeric, alphanumeric, DBCS, or national character values; or a truth value. These values are determined as follows:

- Any selection subject specified by *identifier-1*, *identifier-2*, ... and any selection object specified by *identifier-3* or *identifier-5* without the NOT or THRU phrase are assigned the value and class of the data item that they reference.
- Any selection subject specified by *literal-1*, *literal-2*, ... and any selection object specified by *literal-3* or *literal-5* without the NOT or THRU phrase are assigned

the value and class of the specified literal. If *literal-3* or *literal-5* is the figurative constant ZERO, QUOTE, or SPACE, the figurative constant is assigned the class of the corresponding selection subject.

- Any selection subject in which *expression-1*, *expression-2*, ... is specified as an *arithmetic* expression, and any selection object without the NOT or THRU phrase in which *arithmetic-expression-1* or *arithmetic-expression-3* is specified, are assigned numeric values according to the rules for evaluating an arithmetic expression. (See "Arithmetic expressions" on page 257.)
- Any selection subject in which *expression-1*, *expression-2*, ... is specified as a *conditional* expression, and any selection object in which *condition-1* or *condition-2* is specified, are assigned a truth value according to the rules for evaluating conditional expressions. (See "Conditional expressions" on page 262.)
- Any selection subject or any selection object specified by the words TRUE or FALSE is assigned a truth value. The truth value "true" is assigned to those items specified with the word TRUE, and the truth value "false" is assigned to those items specified with the word FALSE.
- Any selection object specified by the word ANY is not further evaluated.
- If the THRU phrase is specified for a selection object without the NOT phrase, the range of values includes all values that, when compared to the selection subject, are greater than or equal to the first operand and less than or equal to the second operand according to the rules for comparison. If the first operand is greater than the second operand, there are no values in the range.
- If the NOT phrase is specified for a selection object, the values assigned to that item are all values not equal to the value, or range of values, that would have been assigned to the item had the NOT phrase been omitted.

Comparing selection subjects and objects

The execution of the EVALUATE statement then proceeds as if the values assigned to the selection subjects and selection objects were compared to determine whether any WHEN phrase satisfies the set of selection subjects. This comparison proceeds as follows:

1. Each selection object within the set of selection objects for the first WHEN phrase is compared to the selection subject having the same ordinal position within the set of selection subjects. One of the following conditions must be satisfied if the comparison is to be satisfied:
 - a. If the items being compared are assigned numeric, alphanumeric, DBCS, or national character values, or a range of numeric, alphanumeric, DBCS, or national character values, the comparison is satisfied if the value, or one value in the range of values, assigned to the selection object is equal to the value assigned to the selection subject according to the rules for comparison.
 - b. If the items being compared are assigned truth values, the comparison is satisfied if the items are assigned identical truth values.
 - c. If the selection object being compared is specified by the word ANY, the comparison is always satisfied, regardless of the value of the selection subject.
2. If the above comparison is satisfied for every selection object within the set of selection objects being compared, the WHEN phrase containing that set of selection objects is selected as the one satisfying the set of selection subjects.
3. If the above comparison is not satisfied for every selection object within the set of selection objects being compared, that set of selection objects does not satisfy the set of selection subjects.

4. This procedure is repeated for subsequent sets of selection objects in the order of their appearance in the source text, until either a WHEN phrase satisfying the set of selection subjects is selected or until all sets of selection objects are exhausted.

Executing the EVALUATE statement

After the comparison operation is completed, execution of the EVALUATE statement proceeds as follows:

- If a WHEN phrase is selected, execution continues with the first *imperative-statement-1* following the selected WHEN phrase. Note that multiple WHEN statements are allowed for a single *imperative-statement-1*.
- If no WHEN phrase is selected and a WHEN OTHER phrase is specified, execution continues with *imperative-statement-2*.
- If no WHEN phrase is selected and no WHEN OTHER phrase is specified, execution continues with the next executable statement following the scope delimiter.
- The scope of execution of the EVALUATE statement is terminated when execution reaches the end of the scope of the selected WHEN phrase or WHEN OTHER phrase, or when no WHEN phrase is selected and no WHEN OTHER phrase is specified.

EXIT statement

The EXIT statement provides a common end point for a series of procedures.

Format

►►—*paragraph-name*—.—EXIT.—◄◄

The EXIT statement enables you to assign a procedure-name to a given point in a program.

The EXIT statement is treated as a CONTINUE statement. Any statements following the EXIT statement are executed.

EXIT METHOD statement

The EXIT METHOD statement specifies the end of an invoked method.

Format

►►EXIT METHOD.—————►◄

You can specify EXIT METHOD only in the procedure division of a method. EXIT METHOD causes the executing method to terminate, and control returns to the invoking statement. If the containing method specifies the procedure division RETURNING phrase, the value in the data item referred to by the RETURNING phrase becomes the result of the method invocation.

If you need method-specific data to be in the *last-used* state on each invocation, declare it in method working-storage. If you need method-specific data to be in the *initial* state on each invocation, declare it in method local-storage.

If control reaches an EXIT METHOD statement in a method definition, control returns to the point that immediately follows the INVOKE statement in the invoking program or method. The state of the invoking program or method is identical to that which existed at the time it executed the INVOKE statement.

The contents of data items and the contents of data files shared between the invoking program or method and the invoked method could have changed. The state of the invoked method is not altered except that the end of the ranges of all PERFORM statements executed by the method are considered to have been reached.

The EXIT METHOD statement does not have to be the last statement in a sequence of imperative statements, but the statements following the EXIT METHOD will not be executed.

When there is no next executable statement in an invoked method, an implicit EXIT METHOD statement is executed.

EXIT PROGRAM statement

The EXIT PROGRAM statement specifies the end of a called program and returns control to the calling program.

You can specify EXIT PROGRAM only in the procedure division of a program. EXIT PROGRAM must not be used in a declarative procedure in which the GLOBAL phrase is specified.

Format

►►—EXIT PROGRAM.—————►

If control reaches an EXIT PROGRAM statement in a program that does not possess the INITIAL attribute while operating under the control of a CALL statement (that is, the CALL statement is active), control returns to the point in the calling routine (program or method) immediately following the CALL statement. The state of the calling routine is identical to that which existed at the time it executed the CALL statement. The contents of data items and the contents of data files shared between the calling and called routine could have been changed. The state of the called program or method is not altered except that the ends of the ranges of all executed PERFORM statements are considered to have been reached.

The execution of an EXIT PROGRAM statement in a called program that possesses the INITIAL attribute is equivalent also to executing a CANCEL statement referencing that program.

If control reaches an EXIT PROGRAM statement, and no CALL statement is active, control passes through the exit point to the next executable statement.

If a subprogram specifies the procedure division RETURNING phrase, the value in the data item referred to by the RETURNING phrase becomes the result of the subprogram invocation.

The EXIT PROGRAM statement should be the last statement in a sequence of imperative statements. When it is not, statements following the EXIT PROGRAM will not be executed if a CALL statement is active.

When there is no next executable statement in a called program, an implicit EXIT PROGRAM statement is executed.

GOBACK statement

The GOBACK statement functions like the EXIT PROGRAM statement when it is coded as part of a called program (or the EXIT METHOD statement when GOBACK is coded as part of an invoked method) and like the STOP RUN statement when coded in a main program.

The GOBACK statement specifies the logical end of a called program or invoked method.

Format

►► GOBACK ◀◀

A GOBACK statement should appear as the only statement or as the last of a series of imperative statements in a sentence because any statements following the GOBACK are not executed. GOBACK must not be used in a declarative procedure in which the GLOBAL phrase is specified.

If control reaches a GOBACK statement while a CALL statement is active, control returns to the point in the calling program or method immediately following the CALL statement, as in the EXIT PROGRAM statement.

If control reaches a GOBACK statement while an INVOKE statement is active, control returns to the point in the invoking program or method immediately following the INVOKE statement, as in the EXIT METHOD statement.

In addition, the execution of a GOBACK statement in a called program that possesses the INITIAL attribute is equivalent to executing a CANCEL statement referencing that program.

The table below shows the action taken for the GOBACK statement in both a main program and a subprogram.

Termination statement	Main program	Subprogram
GOBACK	Returns to the calling program. (Can be the system, which causes the application to end.)	Returns to the calling program.

GO TO statement

The GO TO statement transfers control from one part of the procedure division to another. The types of GO TO statements are:

- Unconditional
- Conditional
- Altered

Unconditional GO TO

The unconditional GO TO statement transfers control to the first statement in the paragraph or section identified by *procedure-name*, unless the GO TO statement has been modified by an ALTER statement. (See “ALTER statement” on page 312.)

Format 1: unconditional GO TO statement

The diagram shows the syntax for an unconditional GO TO statement. It starts with a double arrow pointing to the keyword 'GO'. Below 'GO' is a bracket labeled 'TO'. A line extends from the 'TO' bracket to the right, ending with a double arrow. The text *procedure-name-1* is placed on this line between the 'TO' bracket and the final double arrow.

procedure-name-1

Must name a procedure or a section in the same procedure division as the GO TO statement.

When the unconditional GO TO statement is not the last statement in a sequence of imperative statements, the statements following the GO TO are not executed.

When a paragraph is referred to by an ALTER statement, the paragraph must consist of a paragraph-name followed by an unconditional or altered GO TO statement.

Conditional GO TO

The conditional GO TO statement transfers control to one of a series of procedures, depending on the value of the data item referenced by *identifier-1*.

Format 2: conditional GO TO statement

The diagram shows the syntax for a conditional GO TO statement. It starts with a double arrow pointing to the keyword 'GO'. Below 'GO' is a bracket labeled 'TO'. A line extends from the 'TO' bracket to the right, ending with a double arrow. The text *procedure-name-1* is placed on this line between the 'TO' bracket and the keyword 'DEPENDING'. Below 'DEPENDING' is a bracket labeled 'ON'. A line extends from the 'ON' bracket to the right, ending with a double arrow. The text *identifier-1* is placed on this line between the 'ON' bracket and the final double arrow. A curved arrow points from the line between 'TO' and 'DEPENDING' back to the 'TO' bracket, indicating a loop or conditional jump.

procedure-name-1

Must be a procedure or a section in the same procedure division as the GO TO statement. The number of procedure-names must not exceed 255.

identifier-1

Must be a numeric elementary data item that is an integer. *identifier-1* cannot be a windowed date field.

If 1, control is transferred to the first statement in the procedure named by the first occurrence of *procedure-name-1*.

If 2, control is transferred to the first statement in the procedure named by the second occurrence of *procedure-name-1*, and so forth.

If the value of *identifier* is anything other than a value within the range of 1 through *n* (where *n* is the number of *procedure-names* specified in this GO TO statement), no control transfer occurs. Instead, control passes to the next statement in the normal sequence of execution.

Altered GO TO

The altered GO TO statement transfers control to the first statement of the paragraph named in the ALTER statement.

You cannot specify the altered GO TO statement in the following:

- A program or method that has the RECURSIVE attribute
- A program compiled with the THREAD compiler option

An ALTER statement referring to the paragraph that contains the altered GO TO statement should be executed before the GO TO statement is executed. Otherwise, the GO TO statement acts like a CONTINUE statement.

Format 3: altered GO TO statement

►► *paragraph-name* . —GO TO . —————►◄

When an ALTER statement refers to a paragraph, the paragraph can consist only of the *paragraph-name* followed by an unconditional or altered GO TO statement.

MORE-LABELS GO TO

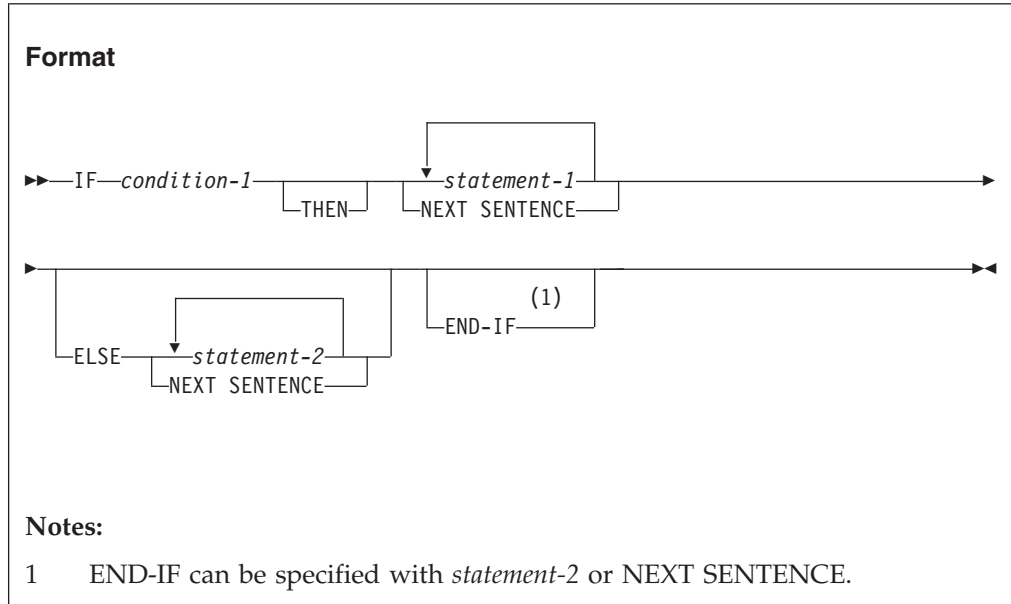
GO TO MORE-LABELS is syntax checked, but has no effect on the execution of the program.

Format 4: MORE-LABELS GO TO statement

►► GO TO *MORE-LABELS* —————►◄

IF statement

The IF statement evaluates a condition and provides for alternative actions in the object program, depending on the evaluation.



condition-1

Can be any simple or complex condition, as described in “Conditional expressions” on page 262.

statement-1, statement-2

Can be any one of the following:

- An imperative statement
- A conditional statement
- An imperative statement followed by a conditional statement

NEXT SENTENCE

The NEXT SENTENCE phrase transfers control to an implicit CONTINUE statement immediately following the next *separator period*.

When NEXT SENTENCE is specified with END-IF, control does not pass to the statement following the END-IF. Instead, control passes to the statement after the closest following period.

END-IF phrase

This explicit scope terminator serves to delimit the scope of the IF statement. END-IF permits a conditional IF statement to be nested in another conditional statement. For more information about explicit scope terminators, see “Delimited scope statements” on page 289.

The scope of an IF statement can be terminated by any of the following:

- An END-IF phrase at the same level of nesting
- A separator period

- If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting

Transferring control

If the condition tested is true, one of the following actions takes place:

- If *statement-1* is specified, *statement-1* is executed. If *statement-1* contains a procedure branching or conditional statement, control is transferred according to the rules for that statement. If *statement-1* does not contain a procedure-branching statement, the ELSE phrase, if specified, is ignored, and control passes to the next executable statement after the corresponding END-IF or separator period.
- If NEXT SENTENCE is specified, control passes to an implicit CONTINUE statement immediately preceding the next separator period.

If the condition tested is false, one of the following actions takes place:

- If ELSE *statement-2* is specified, *statement-2* is executed. If *statement-2* contains a procedure-branching or conditional statement, control is transferred, according to the rules for that statement. If *statement-2* does not contain a procedure-branching or conditional statement, control is passed to the next executable statement after the corresponding END-IF or separator period.
- If ELSE NEXT SENTENCE is specified, control passes to an implicit CONTINUE STATEMENT immediately preceding the next separator period.
- If neither ELSE *statement-2* nor ELSE NEXT SENTENCE is specified, control passes to the next executable statement after the corresponding END-IF or separator period.

When the ELSE phrase is omitted, all statements following the condition and preceding the corresponding END-IF or the separator period for the sentence are considered to be part of *statement-1*.

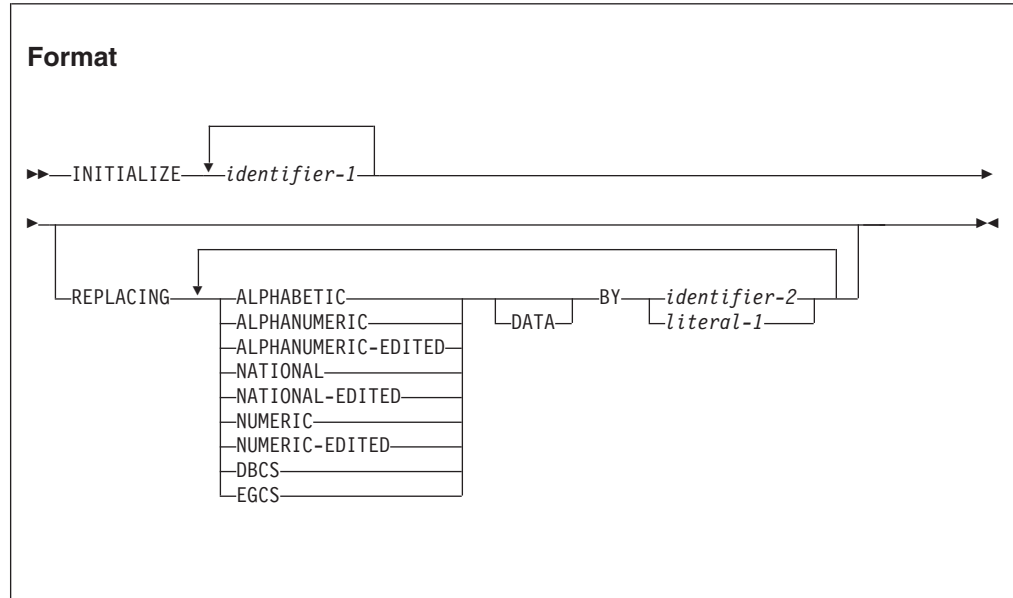
Nested IF statements

When an IF statement appears as *statement-1* or *statement-2*, or as part of *statement-1* or *statement-2*, that IF statement is *nested*.

Nested IF statements are considered to be matched IF, ELSE, and END-IF combinations proceeding from left to right. Thus, any ELSE encountered is matched with the nearest preceding IF that either has not been already matched with an ELSE or has not been implicitly or explicitly terminated. Any END-IF encountered is matched with the nearest preceding IF that has not been implicitly or explicitly terminated.

INITIALIZE statement

The INITIALIZE statement sets selected categories of data fields to predetermined values. It is functionally equivalent to one or more MOVE statements.



identifier-1

Receiving areas.

identifier-1 must reference one of the following:

- An alphanumeric group item
- A national group item
- An elementary data item of one of the following categories:
 - Alphabetic
 - Alphanumeric
 - Alphanumeric-edited
 - DBCS
 - External floating-point
 - Internal floating-point
 - National
 - National-edited
 - Numeric
 - Numeric-edited
- A special register that is valid as a receiving operand in a MOVE statement with *identifier-2* or *literal-1* as the sending operand.

When *identifier-1* references a national group item, *identifier-1* is processed as a group item.

identifier-2, literal-1

Sending areas.

When *identifier-2* references a national group item, *identifier-2* is processed as an elementary data item of category national.

identifier-2 must reference an elementary data item (or a national group item treated as elementary) that is valid as a sending operand in a MOVE statement with *identifier-1* as the receiving operand.

literal-1 must be a literal that is valid as a sending operand in a MOVE statement with *identifier-1* as the receiving operand.

A subscripted item can be specified for *identifier-1*. A complete table can be initialized only by specifying *identifier-1* as a group that contains the complete table.

Usage note: The data description entry for *identifier-1* can contain the DEPENDING phrase of the OCCURS clause. However, you cannot use the INITIALIZE statement to initialize a variably-located item or a variable-length item.

The data description entry for *identifier-1* must not contain a RENAME clause.

Special registers can be specified for *identifier-1* and *identifier-2* only if they are valid receiving fields or sending fields, respectively, for the implied MOVE statements.

REPLACING phrase

When the REPLACING phrase is specified:

- *identifier-2* must reference an item of a category that is valid as a sending operand in a MOVE statement to an item of the corresponding category specified in the REPLACING phrase.
- *literal-1* must be of a category that is valid as a sending operand in a MOVE statement to an item of the corresponding category specified in the REPLACING phrase.
- A floating-point literal, a data item of category internal floating-point, or a data item of category external floating point is treated as if it were in the NUMERIC category.
- The same category cannot be repeated in a REPLACING phrase.

The keyword following the word REPLACING corresponds to a category of data shown in “Classes and categories of data” on page 160.

When the REPLACING phrase is not specified:

- SPACE is the implied sending item for receiving items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.
- ZERO is the implied sending item for receiving items of category numeric or numeric-edited.

INITIALIZE statement rules

1. Whether *identifier-1* references an elementary item, an alphanumeric group item, or a national group item, all operations are performed as if a series of MOVE statements had been written, each of which had an elementary item as a receiving field.

If the REPLACING phrase is specified:

- If *identifier-1* references an alphanumeric group item or a national group item, any elementary item within the data item referenced by *identifier-1* is initialized only if it belongs to a category specified in the REPLACING phrase.

Initialization takes place as if the data item referenced by *identifier-2* or *literal-1* were the sending operand in an implicit MOVE statement to the receiving item. All elementary receiving fields, including all occurrences of table items within the group, are initialized, with the following exceptions:

- Index data items
 - Object references
 - Data items defined with USAGE IS POINTER, USAGE IS FUNCTION-POINTER, or USAGE IS PROCEDURE-POINTER
 - Elementary FILLER data items
 - Items that are subordinate to *identifier-1* and contain a REDEFINES clause, or any items subordinate to such an item. (However, *identifier-1* can contain a REDEFINES clause or be subordinate to a redefining item.)
2. The areas referenced by *identifier-1* are initialized in the order (left to right) of the appearance of *identifier-1* in the statement. Within a group receiving field, affected elementary items are initialized in the order of their definition within the group.
 3. If *identifier-1* occupies the same storage area as *identifier-2*, the result of the execution of this statement is undefined, even if these operands are defined by the same data description entry.

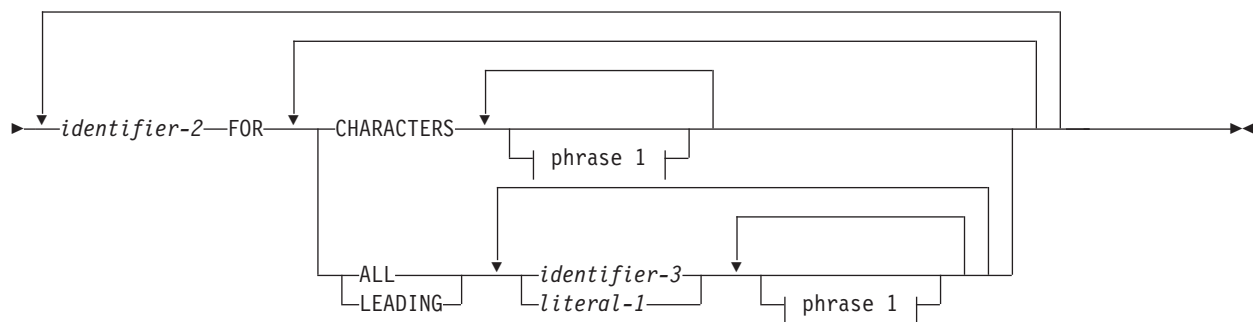
INSPECT statement

The INSPECT statement examines characters or groups of characters in a data item and does the following:

- Counts the occurrences of a specific character (alphanumeric, DBCS, or national) in a data item (formats 1 and 3).
- Counts the occurrences of specific characters and fills all or portions of a data item with specified characters, such as spaces or zeros (formats 2 and 3).
- Converts all occurrences of specific characters in a data item to user-supplied replacement characters (format 4).

Format 1: INSPECT statement with TALLYING phrase

►—INSPECT—*identifier-1*—TALLYING—►

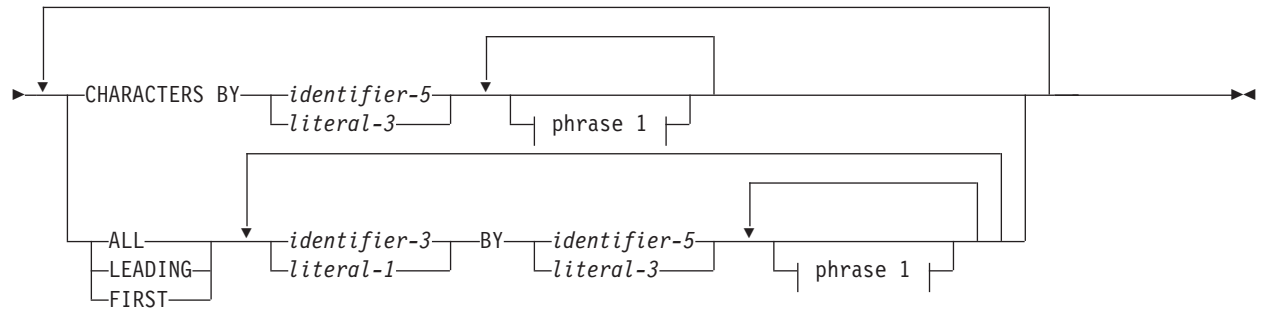


phrase 1:

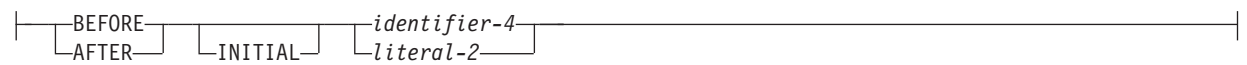
| BEFORE | AFTER | INITIAL | *identifier-4* | *literal-2* |

Format 2: INSPECT statement with REPLACING phrase

►►—INSPECT—*identifier-1*—REPLACING—►

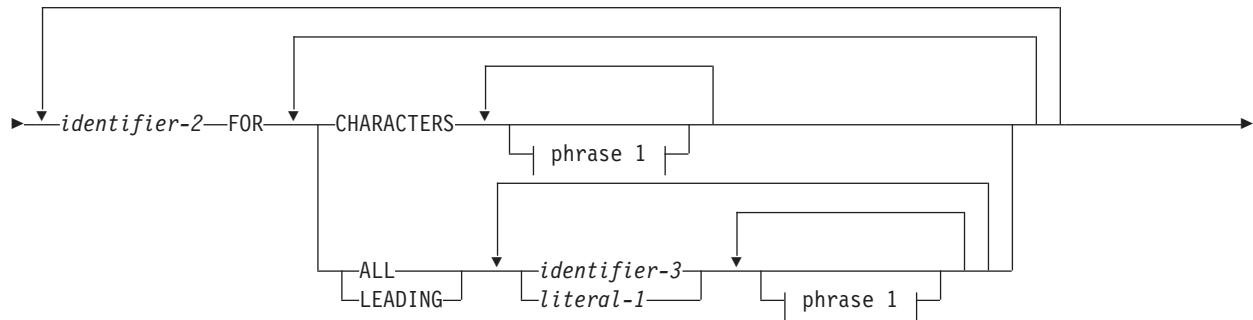


phrase 1:

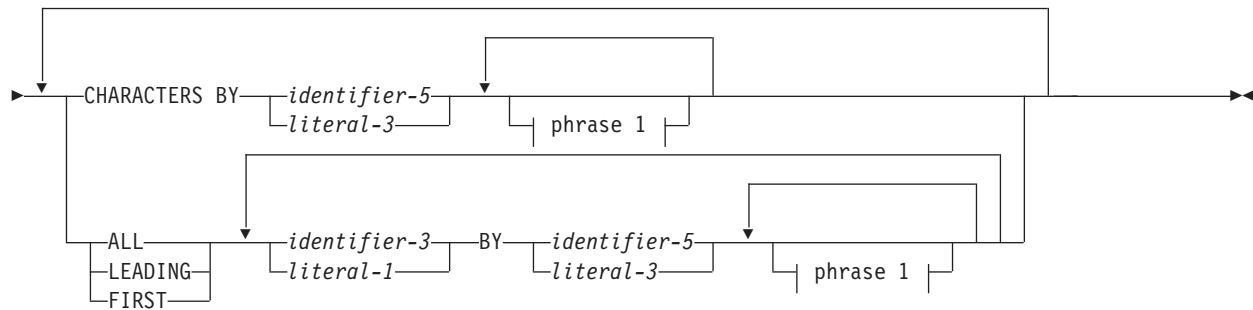


Format 3: INSPECT statement with TALLYING and REPLACING phrases

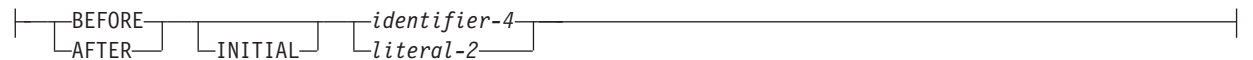
►—INSPECT—*identifier-1*—TALLYING—►



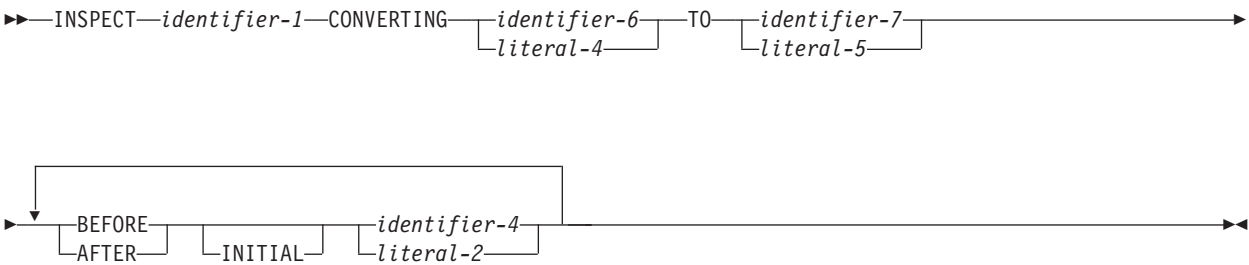
►—REPLACING—►



phrase 1:



Format 4: INSPECT statement with CONVERTING phrase



identifier-1

Is the *inspected item* and can be any of the following:

- An alphanumeric group item or a national group item
- An elementary data item described explicitly or implicitly with usage DISPLAY, DISPLAY-1, or NATIONAL. The item can have any category that is valid for the selected usage.

identifier-3, identifier-4, identifier-5, identifier-6, identifier-7

Must reference an elementary data item described explicitly or implicitly with usage DISPLAY, DISPLAY-1, or NATIONAL.

literal-1, literal-2, literal-3, literal-4

Must be of category alphanumeric, DBCS, or national.

When *identifier-1* is of usage NATIONAL, literals must be of category national.

When *identifier-1* is of usage DISPLAY-1, literals must be of category DBCS.

When *identifier-1* is of usage DISPLAY, literals must be of category alphanumeric.

When *identifier-1* is of usage DISPLAY-1 (DBCS) literals may be the figurative constant SPACE.

When *identifier-1* is of usage DISPLAY or NATIONAL, literals can be any figurative constant that does not begin with the word ALL, as specified in “Figurative constants” on page 13. The figurative constant is treated as a one-character alphanumeric literal when *identifier-1* is of usage DISPLAY, and as a one-character national literal when *identifier-1* is of usage NATIONAL.

All identifiers (except *identifier-2*) must have the same usage as *identifier-1*. All literals must have category alphanumeric, DBCS, or national when *identifier-1* has usage DISPLAY, DISPLAY-1, or NATIONAL, respectively.

None of the identifiers in an INSPECT statement can be windowed date fields.

TALLYING phrase (formats 1 and 3)

This phrase counts the occurrences of a specific character or special character in a data item.

When *identifier-1* is a DBCS data item, DBCS characters are counted; when *identifier-1* is a data item of usage national, national characters (encoding units) are counted; otherwise, alphanumeric characters (bytes) are counted.

identifier-2

Is the *count field*, and must be an elementary integer item defined without the symbol P in its PICTURE character-string.

identifier-2 cannot be of category external floating-point.

You must initialize *identifier-2* before execution of the INSPECT statement begins.

Usage note: The count field can be an integer data item defined with usage NATIONAL.

identifier-3 or literal-1

Is the *tallying field* (the item whose occurrences will be tallied).

CHARACTERS

When CHARACTERS is specified and neither the BEFORE nor AFTER phrase is specified, the count field (*identifier-2*) is increased by 1 for each character (including the space character) in the inspected item (*identifier-1*). Thus, execution of an INSPECT statement with the TALLYING phrase increases the value in the count field by the number of character positions in the inspected item.

ALL When ALL is specified and neither the BEFORE nor AFTER phrase is specified, the count field (*identifier-2*) is increased by 1 for each nonoverlapping occurrence of the tallying comparand (*identifier-3 or literal-1*) in the inspected item (*identifier-1*), beginning at the leftmost character position and continuing to the rightmost.

LEADING

When LEADING is specified and neither the BEFORE nor AFTER phrase is specified, the count field (*identifier-2*) is increased by 1 for each contiguous nonoverlapping occurrence of the tallying comparand in the inspected item (*identifier-1*), provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle for which the tallying comparand is eligible to participate.

FIRST (format 3 only)

When FIRST is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces the leftmost occurrence of the subject field in the inspected item (*identifier-1*).

REPLACING phrase (formats 2 and 3)

This phrase fills all or portions of a data item with specified characters, such as spaces or zeros.

identifier-3 or literal-1

Is the *subject field*, which identifies the characters to be replaced.

identifier-5 or literal-3

Is the *substitution field* (the item that replaces the subject field).

The subject field and the substitution field must be the same length.

CHARACTERS BY

When the CHARACTERS BY phrase is used, the substitution field must be one character position in length.

When CHARACTERS BY is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces each character in the inspected item (*identifier-1*), beginning at the leftmost character position and continuing to the rightmost.

ALL When ALL is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces each nonoverlapping occurrence of the subject field in the inspected item (*identifier-1*), beginning at the leftmost character position and continuing to the rightmost.

LEADING

When LEADING is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces each contiguous nonoverlapping occurrence of the subject field in the inspected item (*identifier-1*), provided that the leftmost such occurrence is at the point where comparison began in the first comparison cycle for which this substitution field is eligible to participate.

FIRST When FIRST is specified and neither the BEFORE nor AFTER phrase is specified, the substitution field replaces the leftmost occurrence of the subject field in the inspected item (*identifier-1*).

When both the TALLYING and REPLACING phrases are specified (format 3), the INSPECT statement is executed as if an INSPECT TALLYING statement (format 1) were specified, immediately followed by an INSPECT REPLACING statement (format 2).

The following replacement rules apply:

- When the subject field is a figurative constant, the one-character substitution field replaces each character in the inspected item that is equivalent to the figurative constant.
- When the substitution field is a figurative constant, the substitution field replaces each nonoverlapping occurrence of the subject field in the inspected item.
- When the subject and substitution fields are character-strings, the character-string specified in the substitution field replaces each nonoverlapping occurrence of the subject field in the inspected item.
- After replacement has occurred in a given character position in the inspected item, no further replacement for that character position is made in this execution of the INSPECT statement.

BEFORE and AFTER phrases (all formats)

This phrase narrows the set of items being tallied or replaced.

No more than one BEFORE phrase and one AFTER phrase can be specified for any one ALL, LEADING, CHARACTERS, FIRST or CONVERTING phrase.

identifier-4 or literal-2

Is the *delimiter*.

Delimiters are not counted or replaced.

INITIAL

The first occurrence of a specified item.

The BEFORE and AFTER phrases change how counting and replacing are done:

- When BEFORE is specified, counting or replacing of the inspected item (*identifier-1*) begins at the leftmost character position and continues until the first occurrence of the delimiter is encountered. If no delimiter is present in the inspected item, counting or replacing continues toward the rightmost character position.
- When AFTER is specified, counting or replacing of the inspected item (*identifier-1*) begins with the first character position to the right of the delimiter and continues toward the rightmost character position in the inspected item. If no delimiter is present in the inspected item, no counting or replacement takes place.

CONVERTING phrase (format 4)

This phrase converts all occurrences of a specific character or string of characters in a data item (*identifier-1*) to user-supplied replacement characters.

identifier-6 or *literal-4*

Specifies the character string to be *replaced*.

The same character must not appear more than once in either *literal-4* or *identifier-6*.

identifier-7 or *literal-5*

Specifies the *replacing* character string.

The replacing character string (*identifier-7* or *literal-5*) must be the same size as the replaced character string (*identifier-6* or *literal-4*).

A format-4 INSPECT statement is interpreted and executed as if a format-2 INSPECT statement had been written with a series of ALL phrases (one for each character of *literal-4*), specifying the same *identifier-1*. The effect is as if each single character of *literal-4* were referenced as *literal-1*, and the corresponding single character of *literal-5* referenced as *literal-3*. Correspondence between the characters of *literal-4* and the characters of *literal-5* is by ordinal position within the data item.

If *identifier-4*, *identifier-6*, or *identifier-7* occupies the same storage area as *identifier-1*, the result of the execution of this statement is undefined, even if they are defined by the same data description entry.

The following table describes the treatment of data items that can be used as an operand in the INSPECT statement:

Table 44. Treatment of the content of data items

When referenced by any identifier except <i>identifier-2</i> , the content of each item of category ...	Is treated ...
Alphanumeric or alphabetic	As an alphanumeric character string
DBCS	As a DBCS character string
National	As a national character string
Alphanumeric-edited, numeric-edited with usage DISPLAY, or numeric with usage DISPLAY (unsigned, external decimal)	As if redefined as category alphanumeric, with the INSPECT statement referring to an alphanumeric character string
National-edited, numeric-edited with usage NATIONAL or numeric with usage NATIONAL (unsigned, external decimal)	As if redefined as category national, with the INSPECT statement referring to a national character string

Table 44. Treatment of the content of data items (continued)

When referenced by any identifier except <i>identifier-2</i> , the content of each item of category ...	Is treated ...
Numeric with usage DISPLAY (signed, external decimal)	<p>As if moved to an unsigned external decimal item of usage DISPLAY with the same length as the identifier and then redefined as category alphanumeric, with the INSPECT statement referring to an alphanumeric character string</p> <p>If the sign is a separate character, the byte containing the sign is not examined and, therefore, not replaced.</p> <p>If the referenced item is <i>identifier-1</i>, the string that results from any replacing or converting action is copied back to <i>identifier-1</i>.</p>
Numeric with usage NATIONAL (signed, external decimal)	<p>As if moved to an unsigned external decimal item of usage NATIONAL with the same length as the identifier and then redefined as category national, with the INSPECT statement referring to a national character string</p> <p>If the sign is a separate character, the byte containing the sign is not examined and, therefore, not replaced.</p> <p>If the referenced item is <i>identifier-1</i>, the string that results from any replacing or converting action is copied back to <i>identifier-1</i>.</p>
External floating-point with usage DISPLAY	As if redefined as category alphanumeric, with the INSPECT statement referring to an alphanumeric character-string
External floating-point with usage NATIONAL	As if redefined as category national, with the INSPECT statement referring to a national character-string

Data flow

Except when the BEFORE or AFTER phrase is specified, inspection begins at the leftmost character position of the inspected item (*identifier-1*) and proceeds character-by-character to the rightmost position.

The comparands of the following phrases are compared in the left-to-right order in which they are specified in the INSPECT statement:

- TALLYING (*literal-1* or *identifier-3*, ...)
- REPLACING (*literal-3* or *identifier-5*, ...)

If any identifier is subscripted or reference modified, or is a function-identifier, the subscript, reference-modifier, or function is evaluated only once as the first operation in the execution of the INSPECT statement.

For examples of TALLYING and REPLACING, see *Tallying and replacing data items (INSPECT)* in the *COBOL for AIX Programming Guide*.

Comparison cycle

The comparison cycle consists of the following actions:

1. The first comparand is compared with an equal number of leftmost contiguous character positions in the inspected item. The comparand matches the inspected characters only if both are equal, character-for-character.

If the CHARACTERS phrase is specified, an implied one-character comparand is used. The implied character is always considered to match the inspected character in the inspected item.

2. If no match occurs for the first comparand and there are more comparands, the comparison is repeated for each successive comparand until either a match is found or all comparands have been acted upon.

3. Depending on whether a match is found, these actions are taken:

- If a match is found, tallying or replacing takes place as described in the TALLYING and REPLACING phrase descriptions.

If there are more character positions in the inspected item, the first character position following the rightmost matching character is now considered to be in the leftmost character position. The process described in actions 1 and 2 is then repeated.

- If no match is found and there are more character positions in the inspected item, the first character position following the leftmost inspected character is now considered to be in the leftmost character position. The process described in actions 1 and 2 is then repeated.

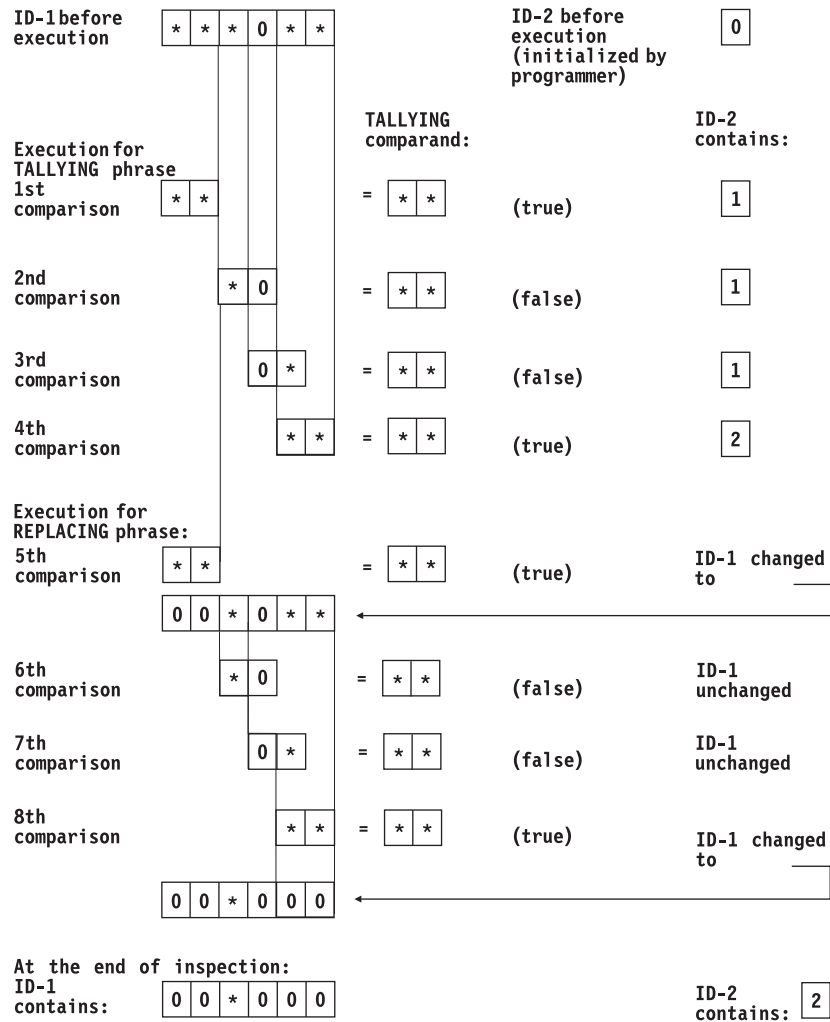
4. Actions 1 through 3 are repeated until the rightmost character position in the inspected item either has been matched or has been considered as being in the leftmost character position.

When the BEFORE or AFTER phrase is specified, the comparison cycle is modified, as described in “BEFORE and AFTER phrases (all formats)” on page 361.

Example of the INSPECT statement

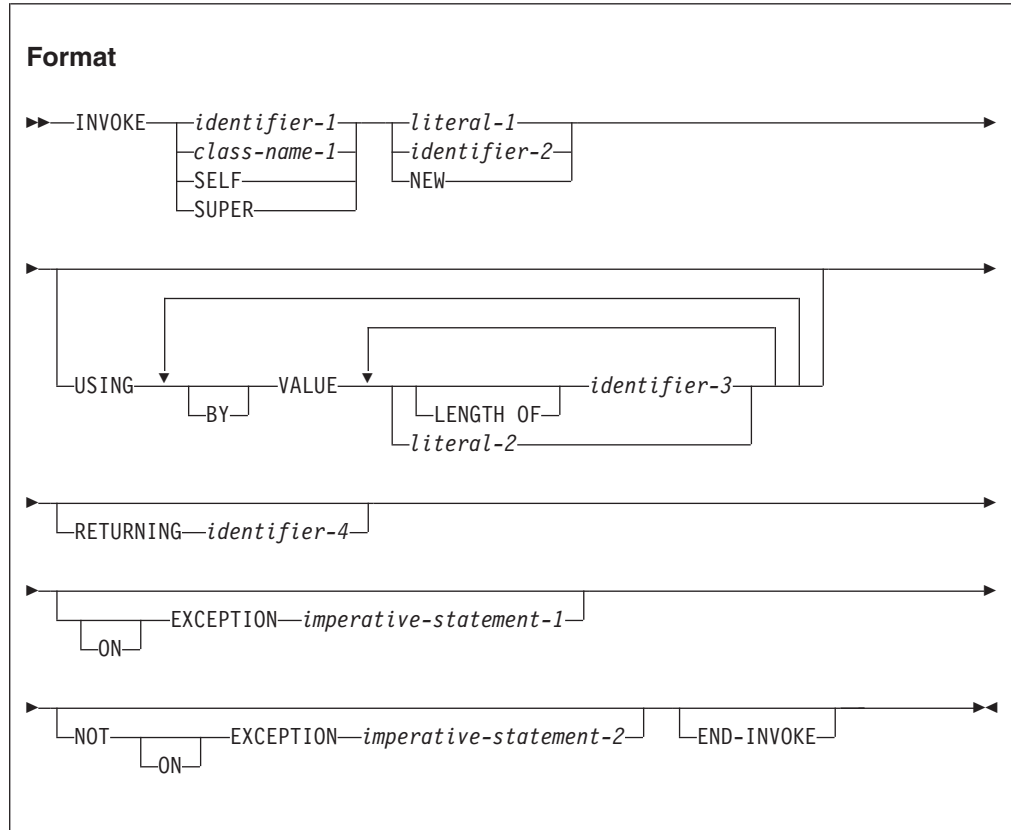
The following figure shows an example of INSPECT statement results.

INSPECT ID-1 TALLYING ID-2 FOR ALL '***' REPLACING ALL '***' BY ZEROS.



INVOKE statement

The INVOKE statement can create object instances of a COBOL or Java class and can invoke a method defined in a COBOL or Java class.



identifier-1

Must be defined as USAGE OBJECT REFERENCE. The contents of *identifier-1* specify the object on which a method is invoked.

When *identifier-1* is specified, either *literal-1* or *identifier-2* must be specified, identifying the name of the method to be invoked.

The results of the INVOKE statement are undefined if either:

- *identifier-1* does not contain a valid reference to an object.
- *identifier-1* contains NULL.

class-name-1

When *class-name-1* is specified together with *literal-1* or *identifier-2*, the INVOKE statement invokes a static or factory method of the class referenced by *class-name-1*. *literal-1* or *identifier-2* specifies the name of the method that is to be invoked. The method must be a static method if *class-name-1* is a Java class; the method must be a factory method if *class-name-1* is a COBOL class.

When *class-name-1* is specified together with NEW, the INVOKE statement creates a new object that is an instance of class *class-name-1*.

You must specify *class-name-1* in the REPOSITORY paragraph of the configuration section of the class or program that contains the INVOKE statement.

SELF An implicit reference to the object used to invoke the currently executing method. When SELF is specified, the INVOKE statement must appear within the procedure division of a method.

SUPER

An implicit reference to the object that was used to invoke the currently executing method. The resolution of the method to be invoked will ignore any methods declared in the class definition of the currently executing method and methods defined in any class derived from that class; thus the method invoked will be one that is inherited from an ancestor class.

literal-1

The value of *literal-1* is the name of the method to be invoked. The referenced object must support the method identified by *literal-1*.

literal-1 must be an alphanumeric literal or a national literal.

literal-1 is interpreted in a case-sensitive manner. The method name, the number of arguments, and the data types of the arguments in the USING phrase of the INVOKE statement are used to select the method with matching signature that is supported by the object. The method can be overloaded.

identifier-2

A data item of category alphabetic, alphanumeric, or national that at run time contains the name of the method to be invoked. The referenced object must support the method identified by *identifier-2*.

If *identifier-2* is specified, *identifier-1* must be defined as USAGE OBJECT REFERENCE without any optional phrases; that is, *identifier-1* must be a universal object reference.

The content of *identifier-2* is interpreted in a case-sensitive manner. The method name, the number of arguments, and the data types of the arguments in the USING phrase of the INVOKE statement are used to select the method with matching signature that is supported by the object. The method can be overloaded.

identifier-2 cannot be a windowed date field.

NEW The NEW operand specifies that the INVOKE statement is to create a new object instance of the class *class-name-1*. *class-name-1* must be specified.

When *class-name-1* is implemented in Java, the USING phrase of the INVOKE statement can be specified. The number of arguments and the data types of the arguments in the USING phrase of the INVOKE statement are used to select the Java constructor with matching signature that is supported by the class. An object instance of class *class-name-1* is allocated, the selected constructor (or the default constructor) is executed, and a reference to the created object is returned.

When *class-name-1* is implemented in COBOL, the USING phrase of the INVOKE statement must not be specified. An object instance of class *class-name-1* is allocated, instance data items are initialized to the values specified in associated VALUE clauses, and a reference to the created object is returned.

When NEW is specified, you must also specify a RETURNING phrase as described in “RETURNING phrase” on page 369.

USING phrase

The USING phrase specifies arguments that are passed to the target method. The argument data types and argument linkage conventions are restricted to those supported by Java. See “BY VALUE phrase” for details.

BY VALUE phrase

Arguments specified in an INVOKE statement must be passed BY VALUE.

The BY VALUE phrase specifies that the value of the argument is passed, not a reference to the sending data item. The invoked method can modify the formal parameter that corresponds to an argument passed by value, but changes do not affect the argument because the invoked method has access only to a temporary copy of the sending data item.

identifier-3

Must be an elementary data item in the data division. The data type of *identifier-3* must be one of the types supported for Java interoperation, as listed in “Interoperable data types for COBOL and Java” on page 371. Miscellaneous cases that are also supported as *identifier-3* are listed in “Miscellaneous argument types for COBOL and Java” on page 373, with their corresponding Java type.

Method arguments are always passed as native data types. If you specify compiler option FLOAT(HEX), you must specify the NATIVE phrase in the description of method arguments that are affected by that compiler option.

See “Conformance requirements for arguments” for additional requirements that apply to *identifier-3*.

literal-2

Must be of a type suitable for Java interoperation and must exactly match the type of the corresponding parameter in the target method. Supported literal forms are listed in “Miscellaneous argument types for COBOL and Java” on page 373, with their corresponding Java type.

literal-2 must not be a DBCS literal.

LENGTH OF *identifier-3*

Specifies that the length of *identifier-3* is passed as an argument in the LENGTH OF special register. A LENGTH OF special register passed BY VALUE is treated as a PIC 9(9) binary value. For information about the LENGTH OF special register, see “LENGTH OF” on page 19.

Conformance requirements for arguments

When *identifier-3* is an object reference, the following rules apply:

- A class-name must be specified in the data description entry for that object reference. That is, *identifier-3* must not be a universal object reference.
- The specified class-name must reference a class that is exactly the class of the corresponding parameter in the invoked method. That is, the class of *identifier-3* must not be a subclass or a superclass of the corresponding parameter's class.

When *identifier-3* is not an object reference, the following rules apply:

- If the target method is implemented in COBOL, the description of *identifier-3* must exactly match the description of the corresponding formal parameter in the target method.
- If the target method is implemented in Java, the description of *identifier-3* must correspond to the Java type of the formal parameter in the target method, as specified in “Interoperable data types for COBOL and Java” on page 371.

Usage note: Adherence to conformance requirements for arguments is the responsibility of the programmer. Conformance requirements are not verified by the compiler.

RETURNING phrase

The RETURNING phrase specifies a data item that will contain the value returned from the invoked method. You can specify the RETURNING phrase on the INVOKE statement when invoking methods that are written in COBOL or Java.

identifier-4

The RETURNING data item. *identifier-4*:

- Must be defined in the data division
- Must not be reference-modified
- Is not changed if an EXCEPTION occurs

The data type of *identifier-4* must be one of the types supported for Java interoperation, as listed in “Interoperable data types for COBOL and Java” on page 371. The value returned from a method is always a native data type. If you specify compiler option FLOAT(HEX), you must specify the NATIVE phrase in the description of returning items that are affected by that compiler option.

See “Conformance requirements for the RETURNING item” on page 370 for additional requirements that apply to *identifier-4*.

If *identifier-4* is specified and the target method is written in COBOL, the target method must have a RETURNING phrase in its procedure division header. When the target method returns, its return value is assigned to *identifier-4* using the rules for the SET statement if *identifier-4* is described with USAGE OBJECT REFERENCE; otherwise, the rules for the MOVE statement are used.

The RETURNING data item is an output-only parameter. On entry to the called method, the initial state of the PROCEDURE DIVISION RETURNING data item has an undefined and unpredictable value. You must initialize the PROCEDURE DIVISION RETURNING data item in the invoked method before you reference its value. The value that is passed back to the invoker is the final value of the PROCEDURE DIVISION RETURNING data item when the invoked method returns.

See *Managing local and global references* in the *COBOL for AIX Programming Guide* for discussion of local and global object references as defined in Java. These attributes affect the life-time of object references.

Usage note: The RETURN-CODE special register is not set by execution of INVOKE statements.

Conformance requirements for the RETURNING item

For INVOKE statements that specify *class-name-1* NEW, the RETURNING phrase is required. The returning item must be one of the following:

- A universal object reference
- An object reference specifying *class-name-1*
- An object reference specifying a superclass of *class-name-1*

For INVOKE statements without the NEW phrase, the RETURNING item specified in the method invocation and in the corresponding target method must satisfy the following requirements:

- The presence or absence of a return value must be the same on the INVOKE statement and in the target method.
- If the RETURNING item is not an object reference, the following rules apply:
 - If the target method is implemented in COBOL, the returning item in the INVOKE statement and the RETURNING item in the target method must have an identical data description entry.
 - If the target method is implemented in Java, the returning item in the INVOKE statement must correspond to the Java type of the method result, as described in “Interoperable data types for COBOL and Java” on page 371.
- If the RETURNING item is an object reference, the RETURNING item specified in the INVOKE statement must be an object reference typed exactly to the class of the returning item specified in the target method. That is, the class of *identifier-4* must not be a subclass or a superclass of the class of the returning item in the target method.

Usage note: Adherence to conformance requirements for returning items is the responsibility of the programmer. Conformance requirements are not verified by the compiler.

ON EXCEPTION phrase

An exception condition occurs when the identified object or class does not support a method with a signature that matches the signature of the method specified in the INVOKE statement. When an exception condition occurs, one of the following actions occurs:

- If the ON EXCEPTION phrase is specified, control is transferred to *imperative-statement-1*.
- If the ON EXCEPTION phrase is not specified, a severe runtime error occurs.

NOT ON EXCEPTION phrase

If an exception condition does not occur (that is, the identified method is supported by the specified object), control is transferred to the invoked method. After control is returned from the invoked method, control is then transferred:

1. To *imperative-statement-2*, if the NOT ON EXCEPTION phrase is specified.
2. To the end of the INVOKE statement if the NOT ON EXCEPTION phrase is not specified.

END-INVOKE phrase

This explicit scope terminator serves to delimit the scope of the INVOKE statement. An INVOKE statement that is terminated by END-INVOKE, along with its contained statements, becomes a unit that is treated as though it were an imperative statement. It can be specified as an imperative statement in a conditional statement; for example, in the exception phrase of another statement.

Interoperable data types for COBOL and Java

A subset of COBOL data types can be used for interoperation between COBOL and Java.

You can specify the interoperable data types as arguments in COBOL INVOKE statements and as the RETURNING item in COBOL INVOKE statements. Similarly, you can pass these types as arguments from a Java method invocation expression and receive them as parameters in the USING phrase or as the RETURNING item in the procedure division header of a COBOL method.

The following table lists the primitive Java types and the COBOL data types that are supported for interoperation and the correspondence between them.

Table 45. Interoperable Java and COBOL data types

Java data type	COBOL data type
boolean ¹	Conditional variable and two condition-names of the form: <i>level-number data-name PIC X.</i> 88 <i>data-name-false</i> VALUE X'00'. 88 <i>data-name-true</i> VALUE X'01' THROUGH X'FF'.
byte ²	Single-byte alphanumeric, PIC X or PIC A
short ³	USAGE BINARY, COMP, COMP-4, or COMP-5, with a PICTURE clause of the form S9(n), where 1 <= n <= 4
int ³	USAGE BINARY, COMP, COMP-4, or COMP-5, with a PICTURE clause of the form S9(n), where 5 <= n <= 9
long ³	USAGE BINARY, COMP, COMP-4, or COMP-5, with a PICTURE clause of the form S9(n), where 10 <= n <= 18
float ³	USAGE COMP-1
double ³	USAGE COMP-2
char	Single-character national: PIC N USAGE NATIONAL (an elementary data item of category national)
class types (object references)	USAGE OBJECT REFERENCE <i>class-name</i>
<ol style="list-style-type: none">1. COBOL for AIX interprets a PIC X argument or parameter as the Java boolean type only when the PIC X data item is followed by exactly two condition-names of the form shown. In all other cases, a PIC X argument or parameter is interpreted as the Java byte type.2. Both the NATIVE character format and the EBCDIC character format are interoperable with the Java byte type.3. Only the NATIVE floating point formats are interoperable with the Java long, float, and double types.	

In addition to the primitive types, Java Strings and arrays of Java primitive types can interoperate with COBOL. This requires specialized mechanisms provided by the COBOL runtime system and the Java Native Interface (JNI).

In a Java program, to pass array data to COBOL or to receive array data from COBOL, you declare the array types using the usual Java syntax. In the COBOL program, you declare the array as an object reference that contains an instance of one of the special classes provided for array support. Conversion between the Java and COBOL types is automatic at the time of method invocation.

In a Java program, to pass String data to COBOL or to receive String data from COBOL, you declare the array types using the usual Java syntax. In the COBOL program, you declare the String as an object reference that contains an instance of the special `jstring` class. Conversion between the Java and COBOL types is automatic at the time of method invocation. The following table lists the Java array and String data types and the corresponding special COBOL data types.

Table 46. Interoperable COBOL and Java array and String data types

Java data type	COBOL data type
<code>boolean[]</code>	object reference <code>jbooleanArray</code>
<code>byte[]</code>	object reference <code>jbyteArray</code>
<code>short[]</code>	object reference <code>jshortArray</code>
<code>int[]</code>	object reference <code>jintArray</code>
<code>long[]</code>	object reference <code>jlongArray</code>
<code>char[]</code>	object reference <code>jcharArray</code>
<code>Object[]</code>	object reference <code>jobjectArray</code>
<code>String</code>	object reference <code>jstring</code>

The following java array types are not currently supported:

Java data type	COBOL data type
<code>float[]</code>	object reference <code>jfloatArray</code>
<code>double[]</code>	object reference <code>jdoubleArray</code>

You must code an entry in the repository paragraph for each special class that you want to use, just as you do for other classes. For example, to use `jstring`, code the following entry:

Configuration Section.

Repository.

Class `jstring` is "`jstring`".

Alternatively, for the `String` type, the COBOL repository entry can specify an external class name of `java.lang.String`:

Repository.

Class `jstring` is "`java.lang.String`".

Callable services are provided by the Java Native Interface (JNI) for manipulating the COBOL objects of these types in COBOL. For example, callable services can be used to set COBOL alphanumeric or national data into a `jstring` object or to extract data from a `jstring` object. For details on use of JNI callable services for these and other purposes, see *Accessing JNI services* in the *COBOL for AIX Programming Guide*.

For details on repository entries for class definitions, see “REPOSITORY paragraph” on page 118. For examples, see *Example: external class-names and Java packages* in the *COBOL for AIX Programming Guide*.

Miscellaneous argument types for COBOL and Java

Miscellaneous cases of COBOL items that can be used as arguments in an INVOKE statement are listed in the following table along with the corresponding Java type.

Table 47. COBOL miscellaneous argument types and corresponding Java types

COBOL argument	Corresponding Java data type
Reference-modified item of usage display with length one (NATIVE or EBCDIC)	byte
Reference-modified item of usage national with length one (either an elementary data item of usage national or a national group item)	char
SHIFT-IN and SHIFT-OUT special registers	byte
LINAGE-COUNTER special register when its usage is binary	int
LENGTH OF special register	int

The following table lists COBOL literal types that can be used as arguments in an INVOKE statement, with the corresponding Java type.

Table 48. COBOL literal argument types and corresponding Java types

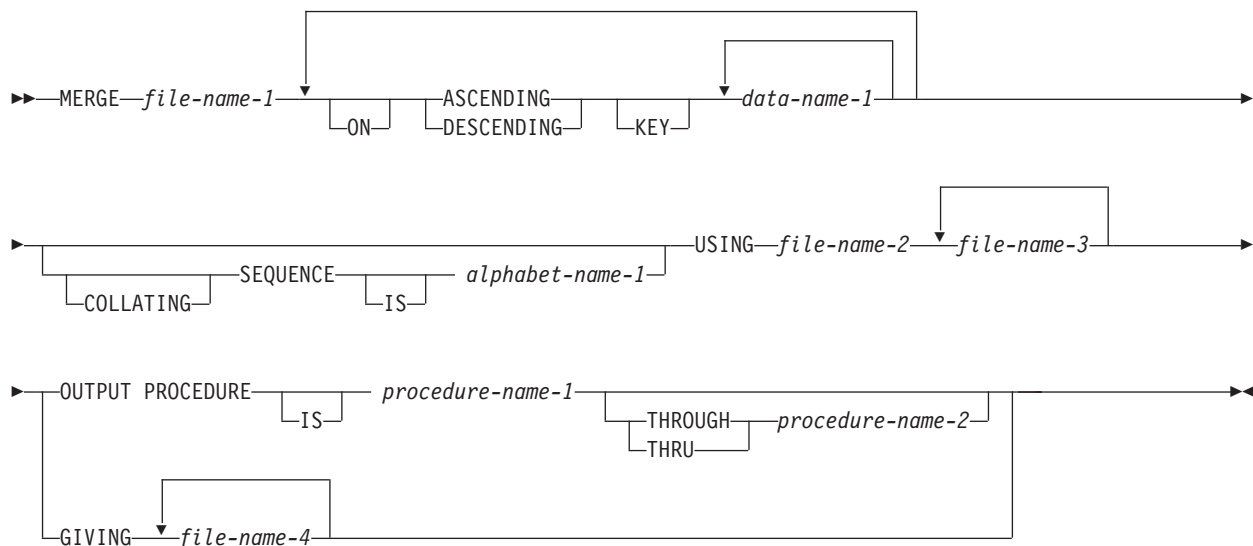
COBOL literal argument	Corresponding Java data type
Fixed-point numeric literal with no decimal positions and with nine digits or less	int
Floating-point numeric literal	double
Figurative constant ZERO	int
One-character alphanumeric literal	byte
One-character national literal	char
Symbolic character	byte
Figurative constants SPACE, QUOTE, HIGH-VALUE, or LOW-VALUE	byte

MERGE statement

The MERGE statement combines two or more identically sequenced files (that is, files that have already been sorted according to an identical set of ascending/descending keys) on one or more keys and makes records available in merged order to an output procedure or output file.

A MERGE statement can appear anywhere in the procedure division except in a declarative section.

Format



file-name-1

The name given in the SD entry that describes the records to be merged.

No file-name can be repeated in the MERGE statement.

No pair of file-names in a MERGE statement can be specified in the same SAME AREA, SAME SORT AREA, or SAME SORT-MERGE AREA clause. However, any file-names in the MERGE statement can be specified in the same SAME RECORD AREA clause.

When the MERGE statement is executed, all records contained in *file-name-2*, *file-name-3*, ... , are accepted by the merge program and then merged according to the keys specified.

ASCENDING/DESCENDING KEY phrase

This phrase specifies that records are to be processed in an ascending or descending sequence (depending on the phrase specified), based on the specified merge keys.

data-name-1

Specifies a KEY data item on which the merge will be based. Each such data-name must identify a data item in a record associated with *file-name-1*. The data-names following the word KEY are listed from left to right in the MERGE statement in order of decreasing significance without regard to how they are divided into KEY phrases. The leftmost data-name is the major key, the next data-name is the next most significant key, and so forth.

The following rules apply:

- A specific key data item must be physically located in the same position and have the same data format in each input file. However, it need not have the same data-name.
- If *file-name-1* has more than one record description, the KEY data items need be described in only one of the record descriptions.
- If *file-name-1* contains variable-length records, all of the KEY data-items must be contained within the first *n* character positions of the record, where *n* equals the minimum records size specified for *file-name-1*.
- KEY data items must not contain an OCCURS clause or be subordinate to an item that contains an OCCURS clause.
- KEY data items cannot be:
 - Variably located
 - Group items that contain variable-occurrence data items
 - Windowed date fields
 - Category numeric described with usage NATIONAL (national decimal type)
 - Category external floating-point described with usage NATIONAL (national floating-point)
 - Category DBCS
- KEY data items can be qualified.
- KEY data items can be any of the following data categories:
 - Alphabetic, alphanumeric, alphanumeric-edited
 - Numeric (except numeric with usage NATIONAL)
 - Numeric-edited (with usage DISPLAY or NATIONAL)
 - Internal floating-point or display floating-point
 - National or national-edited when the NCOLLSEQ(BINARY) compiler option is in effect. Binary collating sequence is applied to national keys.

The direction of the merge operation depends on the specification of the ASCENDING or DESCENDING keywords as follows:

- When ASCENDING is specified, the sequence is from the lowest key value to the highest key value.
- When DESCENDING is specified, the sequence is from the highest key value to the lowest key value.

If the KEY data item is alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited, the sequence of key values depends on the collating sequence used (see “COLLATING SEQUENCE phrase” on page 376 below).

If the KEY data item is described with usage NATIONAL, the sequence of the KEY values is based on the binary values of the national characters.

If the KEY is an external floating-point item with usage DISPLAY, the key is treated as category alphanumeric. The sequence in which the records are merged depends on the collating sequence used.

If the KEY is an external floating-point item with usage NATIONAL, the key is treated as category national.

If the KEY is an internal floating-point item, the sequence of key values is numeric order.

When the COLLATING SEQUENCE phrase is not specified, the key comparisons are performed according to the rules for comparison of operands in a relation condition. For details, see "General relation conditions" on page 267.

When the COLLATING SEQUENCE phrase is specified, the indicated collating sequence is used for key data items of alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-edited categories. For all other key data items, the comparisons are performed according to the rules for comparison of operands in a relation condition.

COLLATING SEQUENCE phrase

This phrase specifies the collating sequence to be used in alphanumeric comparisons for the KEY data items in this merge operation.

The COLLATING SEQUENCE phrase has no effect for keys that are not alphabetic or alphanumeric.

The COLLATING SEQUENCE phrase is valid only when a single-byte ASCII code page is in effect.

alphabet-name-1

Must be specified in the ALPHABET clause of the SPECIAL-NAMES paragraph. Any one of the alphabet-name clause phrases can be specified, with the following results:

STANDARD-1

The collating sequence is based on the character's hex value order.

STANDARD-2

The collating sequence is based on the character's hex value order.

NATIVE

The collating sequence indicated by the runtime locale is selected.

EBCDIC

The EBCDIC collating sequence is used for all alphanumeric comparisons. (The EBCDIC collating sequence is shown in "EBCDIC collating sequence" on page 585.)

literal The collating sequence established by the specification of literals in the ALPHABET-NAME clause is used for all alphanumeric comparisons.

When the COLLATING SEQUENCE phrase is omitted, the PROGRAM COLLATING SEQUENCE clause (if specified) in the OBJECT-COMPUTER

paragraph identifies the collating sequence to be used. When both the COLLATING SEQUENCE phrase of the MERGE statement and the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph are omitted, the COLLSEQ compiler option indicates the collating sequence used. If COLLSEQ(EBCDIC) is specified, the EBCDIC collating sequence is used. If COLLSEQ(LOCALE) is specified, the collating sequence as indicated by the locale is used. For more information about locales, see Appendix G, “Locale considerations,” on page 609.

USING phrase

file-name-2, file-name-3, ...

Specifies the input files.

During the MERGE operation, all the records on *file-name-2, file-name-3, ...* (that is, the input files) are transferred to *file-name-1*. At the time the MERGE statement is executed, these files must not be open. The input files are automatically opened, read, and closed. If DECLARATIVE procedures are specified for these files for input operations, the declaratives will be driven for errors if errors occur.

All input files must specify sequential or dynamic access mode and be described in FD entries in the data division.

If *file-name-1* contains variable-length records, the size of the records contained in the input files (*file-name-2, file-name-3, ...*) must be neither less than the smallest record nor greater than the largest record described for *file-name-1*. If *file-name-1* contains fixed-length records, the size of the records contained in the input files must not be greater than the largest record described for *file-name-1*. For more information, see *Sorting and merging files* in the *COBOL for AIX Programming Guide*.

GIVING phrase

file-name-4, ...

Specifies the output files.

When the GIVING phrase is specified, all the merged records in *file-name-1* are automatically transferred to the output files (*file-name-4, ...*).

All output files must specify sequential or dynamic access mode and be described in FD entries in the data division.

If the output files (*file-name-4, ...*) contain variable-length records, the size of the records contained in *file-name-1* must be neither less than the smallest record nor greater than the largest record described for the output files. If the output files contain fixed-length records, the size of the records contained in *file-name-1* must not be greater than the largest record described for the output files. For more information, see *Sorting and merging files* in the *COBOL for AIX Programming Guide*.

At the time the MERGE statement is executed, the output files (*file-name-4, ...*) must not be open. The output files are automatically opened, read, and closed. If DECLARATIVE procedures are specified for these files for output operations, the declaratives will be driven for errors if errors occur.

OUTPUT PROCEDURE phrase

This phrase specifies the name of a procedure that is to select or modify output records from the merge operation.

procedure-name-1

Specifies the first (or only) section or paragraph in the OUTPUT PROCEDURE.

procedure-name-2

Identifies the last section or paragraph of the OUTPUT PROCEDURE.

The OUTPUT PROCEDURE can consist of any procedure needed to select, modify, or copy the records that are made available one at a time by the RETURN statement in merged order from the file referenced by *file-name-1*. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, PERFORM, and XML PARSE statements in the range of the output procedure. The range also includes all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure. The range of the output procedure must not cause the execution of any MERGE, RELEASE, or SORT statement.

If an output procedure is specified, control passes to it after the file referenced by *file-name-1* has been sequenced by the MERGE statement. The compiler inserts a return mechanism at the end of the last statement in the output procedure and when control passes the last statement in the output procedure, the return mechanism provides the termination of the merge and then passes control to the next executable statement after the MERGE statement. Before entering the output procedure, the merge procedure reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.

The OUTPUT PROCEDURE phrase is similar to a basic PERFORM statement. For example, if you name a procedure in an OUTPUT PROCEDURE, that procedure is executed during the merging operation just as if it were named in a PERFORM statement. As with the PERFORM statement, execution of the procedure is terminated after the last statement completes execution. The last statement in an OUTPUT PROCEDURE can be the EXIT statement (see “EXIT statement” on page 345).

MERGE special registers

SORT-CONTROL special register

You identify the sort control file (through which you can specify additional options to the sort/merge function) with the SORT-CONTROL special register.

If you use a sort control file to specify control statements, the values specified in the sort control file take precedence over those in the other SORT special registers.

For information, see “SORT-CONTROL” on page 22.

SORT-RETURN special register

For information, see “SORT-RETURN” on page 24.

Segmentation considerations

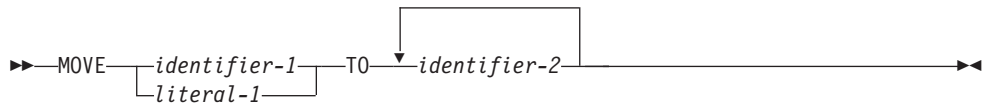
If a MERGE statement is coded in a fixed segment, any output procedure referenced by that MERGE statement must be either totally within a fixed segment or wholly contained in a single independent segment.

If a MERGE statement is coded in an independent segment, any output procedure referenced by that MERGE statement must be either totally within a fixed segment or wholly contained within the same independent segment as that MERGE statement.

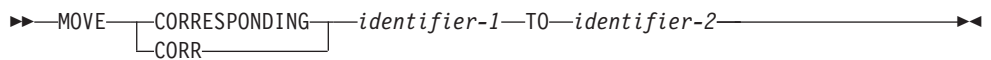
MOVE statement

The MOVE statement transfers data from one area of storage to one or more other areas.

Format 1: MOVE statement



Format 2: MOVE statement with CORRESPONDING phrase



CORR is an abbreviation for, and is equivalent to, CORRESPONDING.

identifier-1 , *literal-1*

The sending area.

identifier-2

The receiving areas. *identifier-2* must not reference an intrinsic function.

When format 1 is specified:

- All identifiers can reference alphanumeric group items, national group items, or elementary items.
- When one of *identifier-1* or *identifier-2* references a national group item and the other operand references an alphanumeric group item, the national group is processed as a group item; in all other cases, the national group item is processed as an elementary data item of category national.
- The data in the sending area is moved into the data item referenced by each *identifier-2* in the order in which the *identifier-2* data items are specified in the MOVE statement. See “Elementary moves” on page 381 and “Group moves” on page 386 below.

When format 2 is specified:

- Both identifiers must be group items.
- A national group item is processed as a group item (and not as an elementary data item of category national).
- Selected items in *identifier-1* are moved to *identifier-2* according to the rules for the “CORRESPONDING phrase” on page 291. The results are the same as if each pair of CORRESPONDING identifiers were referenced in a separate MOVE statement.

Data items described with the following types of usage cannot be specified in a MOVE statement:

- INDEX
- POINTER
- FUNCTION-POINTER
- PROCEDURE-POINTER
- OBJECT REFERENCE

A data item defined with a usage of INDEX, POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE can be part of an alphanumeric group item that is referenced in a MOVE CORRESPONDING statement; however, no movement of data from those data items takes place.

The evaluation of the length of the sending or receiving area can be affected by the DEPENDING ON phrase of the OCCURS clause (see “OCCURS clause” on page 192).

If the sending field (*identifier-1*) is reference-modified or subscripted, or is an alphanumeric or national function-identifier, the reference-modifier, subscript, or function is evaluated only once, immediately before data is moved to the first of the receiving operands.

Any length evaluation, subscripting, or reference-modification associated with a receiving field (*identifier-2*) is evaluated immediately before the data is moved into that receiving field.

For example, the result of the statement:

```
MOVE A(B) TO B, C(B).
```

is equivalent to:

```
MOVE A(B) TO TEMP.  
MOVE TEMP TO B.  
MOVE TEMP TO C(B).
```

where TEMP is defined as an intermediate result item. The subscript B has changed in value between the time that the first move took place and the time that the final move to C(B) is executed.

For further information about intermediate results, see *Appendix A. Intermediate results and arithmetic precision* in the *COBOL for AIX Programming Guide*.

After execution of a MOVE statement, the sending fields contain the same data as before execution.

Usage note: Overlapping operands in a MOVE statement can cause unpredictable results.

Elementary moves

An elementary move is one in which the receiving item is an elementary data item and the sending item is an elementary data item or a literal.

Valid operands belong to one of the following categories:

- **Alphabetic:** includes data items of category alphabetic and the figurative constant SPACE
- **Alphanumeric:** includes the following:
 - Data items of category alphanumeric
 - Alphanumeric functions
 - Alphanumeric literals
 - The figurative constant ALL *alphanumeric-literal* and all other figurative constants (except NULL) when used in a context that requires an alphanumeric sending item
- **Alphanumeric-edited:** includes data items of category alphanumeric-edited
- **DBCS:** includes data items of category DBCS, DBCS literals, and the figurative constant ALL DBCS-literal.
- **External floating-point:** includes data items of category external floating point (described with USAGE DISPLAY or USAGE NATIONAL) and floating-point literals.
- **Internal floating-point:** includes data items of category internal floating-point (defined as USAGE COMP-1 or USAGE COMP-2)
- **National:** includes the following:
 - National group items (treated as elementary item of category national)
 - Data items of category national
 - National literals
 - National functions
 - Figurative constants ZERO, SPACE, QUOTE, and ALL *national-literal* when used in a context that requires a national sending item
- **National-edited:** includes data items of category national-edited
- **Numeric:** includes the following:
 - Data items of category numeric
 - Numeric literals
 - The figurative constant ZERO (when ZERO is moved to a numeric or numeric-edited item).
- **Numeric-edited:** includes data items of category numeric-edited.

Elementary move rules

Any necessary conversion of data from one form of internal representation to another takes place during the move, along with any specified editing in, or de-editing implied by, the receiving item. The code page used for conversion to or from alphanumeric characters is the code page applicable to the specific data item at run time.

The following rules outline the execution of valid elementary moves. When the receiving field is:

Alphabetic:

- Alignment and any necessary space filling or truncation occur as described under “Alignment rules” on page 164.
- If the size of the sending item is greater than the size of the receiving item, excess characters on the right are truncated after the receiving item is filled.

Alphanumeric or alphanumeric-edited:

- If the sending item is a national decimal integer item, the sending data is converted to usage DISPLAY and treated as though it were moved to a temporary data item of category alphanumeric with the same number of character positions as the sending item. The resulting alphanumeric data item is treated as the sending item.
- Alignment and any necessary space filling or truncation take place, as described under “Alignment rules” on page 164.
- If the size of the sending item is greater than the size of the receiving item, excess characters on the right are truncated after the receiving item is filled.
- If the initial sending item has an operational sign, the unsigned value is used. If the operational sign occupies a separate character, that character is not moved, and the size of the sending item is considered to be one less character than the actual size.

DBCS:

- If the sending and receiving items are not the same size, the sending data is either truncated on the right or padded with DBCS spaces on the right. If the padding required is not in a multiple consistent with double-byte characters, single-byte characters are used (for example, a DBCS data item moved to an alphanumeric group item).

External floating-point:

- For a floating-point sending item, the floating-point value is converted to the usage of the receiving external floating-point item (if different from the sending item's representation).
- For other sending items, the numeric value is treated as though that value were converted to internal floating-point and then converted to the usage of the receiving external floating-point item.

Internal floating-point:

- When the category of the sending operand is not internal floating-point, the numeric value of the sending item is converted to internal floating-point format.

National or national-edited:

- If the representation of the sending item is not national characters, the sending data is converted to national characters and treated as though it were moved to a temporary data item of category national of a length not to cause truncation or padding. The resulting category national data item is treated as the sending data item.
- If the representation of the sending item is national characters, the sending data is used without conversion.
- Alignment and any necessary space filling or truncation take place as described under “Alignment rules” on page 164. The programmer is responsible for ensuring that multiple encoding units that together form a graphic character are not split by truncation.
- If the sending item has an operational sign, the unsigned value is used. If the operational sign occupies a separate character, that character is not moved, and the size of the sending item is considered to be one less character than the actual size.

Numeric or numeric-edited:

- Except when zeros are replaced because of editing requirements, alignment by decimal point and any necessary zero filling take place, as described under “Alignment rules” on page 164.
- If the receiving item is signed, the sign of the sending item is placed in the receiving item, with any necessary sign conversion. If the sending item is unsigned, a positive operational sign is generated for the receiving item.
- If the receiving item is unsigned, no operational sign is generated for the receiving item and the absolute value of the sending item is used in the move.
- When the category of the sending item is alphanumeric, alphanumeric-edited, national, or national-edited, the data is moved as if the sending item were described as an unsigned integer.
- When the sending item is floating-point, the data is first converted to either a binary or internal decimal representation and is then moved.
- When the receiving item is numeric-edited, editing takes place as defined by the picture character string or BLANK WHEN ZERO clause associated with the receiving item.
- When the sending item is numeric-edited, the compiler de-edits the sending data to establish the unedited value of the numeric-edited item (this value can be signed). The unedited numeric value is used in the move to the receiving numeric or numeric-edited data item.

Usage notes:

1. If the receiving item is of category alphanumeric, alphanumeric-edited, numeric-edited, national, or national-edited and the sending field is numeric, any digit positions described with picture symbol P in the sending item are considered to have the value zero. Each P is counted in the size of the sending item.
2. If the receiving item is numeric and the sending field is an alphanumeric literal, a national literal, or an ALL literal, all characters of the literal must be numeric characters.

Valid and invalid elementary moves

The following table shows valid and invalid elementary moves for each category. In the table:

- YES = Move is valid.
- NO = Move is invalid.
- Column headings indicate receiving item categories; row headings indicate sending item categories.

Table 49. Valid and invalid elementary moves

	Alphabetic	Alphanumeric	Alphanumeric-edited	Numeric	Numeric-edited	External floating-point	Internal floating-point	DBCS ¹	National, national-edited
Alphabetic and SPACE sending item	Yes	Yes	Yes	No	No	No	No	No	Yes
Alphanumeric sending item ²	Yes	Yes	Yes	Yes ³	Yes ³	Yes ⁸	Yes ⁸	No	Yes
Alphanumeric-edited sending item	Yes	Yes	Yes	No	No	No	No	No	Yes

Table 49. Valid and invalid elementary moves (continued)

	Alpha- betic	Alpha- numeric	Alpha- numeric edited	Numeric	Numeric- edited	External floating- point	Internal floating- point	DBCS ¹	National, national- edited
Numeric integer and ZERO sending item ⁴	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Numeric noninteger sending item ⁵	No	No	No	Yes	Yes	Yes	Yes	No	No
Numeric-edited sending item	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Floating-point sending item ⁶	No	No	No	Yes	Yes	Yes	Yes	No	No
DBCS sending item ⁷	No	No	No	No	No	No	No	Yes	Yes
National sending item ⁹	No	No	No	Yes	Yes	Yes	Yes	No	Yes
National-edited sending item	No	No	No	No	No	No	No	No	Yes
<ol style="list-style-type: none"> 1. Includes DBCS data items. 2. Includes alphanumeric literals. 3. Figurative constants and alphanumeric literals must consist only of numeric characters and will be treated as numeric integer fields. 4. Includes integer numeric literals. 5. Includes noninteger numeric literals. 6. Includes floating-point literals, external floating-point data items (USAGE DISPLAY or USAGE NATIONAL), and internal floating-point data items (USAGE COMP-1 or USAGE COMP-2). 7. Includes DBCS data-items, DBCS literals, and figurative constant SPACE. 8. Figurative constants and alphanumeric literals must consist only of numeric characters and will be treated as numeric integer fields. The ALL literal cannot be used as a sending item. 9. Includes national data items, national literals, national functions, and figurative constants ZERO, SPACE, QUOTE, and ALL national literal. 									

Moves involving date fields

If the sending item is specified as a year-last date field, then all receiving fields must also be year-last date fields with the same date format as the sending item. If a year-last date field is specified as a receiving item, then the sending item must be either a nondate or a year-last date field with the same date format as the receiving item. In both cases, the move is then performed as if all items were nondates.

Table 50 on page 386 describes the behavior of moves involving non-year-last date fields. If the sending item is a date field, then the receiving item must be a compatible date field. If the sending and receiving items are both date fields, then they must be compatible; that is, they must have the same date format, except for the year part, which can be windowed or expanded.

This table uses the following terms to describe the moves:

Normal

The move is performed with no date-sensitive behavior, as if the sending and receiving items were both nondates.

Expanded

The windowed date field sending item is treated as if it were first converted to expanded form, as described under “Semantics of windowed date fields” on page 185.

Invalid

The move is not allowed.

Table 50. Moves involving date fields

	Nondate receiving item	Windowed date field receiving item	Expanded date field receiving item
Nondate sending item	Normal	Normal	Normal
Windowed date field sending item	Invalid	Normal	Expanded
Expanded date field sending item	Invalid	Normal ¹	Normal
1. A move from an expanded date field to a windowed date field is, in effect, a "windowed" move, because it truncates the century component of the expanded date field. If the move is alphanumeric, it treats the receiving windowed date field as if its data description specified JUSTIFIED RIGHT. This is true even if the receiving windowed date field is a group item, for which the JUSTIFIED clause cannot be specified.			

Moves involving file record areas

The successful execution of an OPEN statement for a given file makes the record area for that file available. You can move data to or from the record description entries associated with a file only when the file is in the open status. Execution of an implicit or explicit CLOSE statement removes a file from open status and makes the record area unavailable.

Group moves

A group move is any move in which an alphanumeric group item is a sending item or a receiving item, or both. The following are group moves:

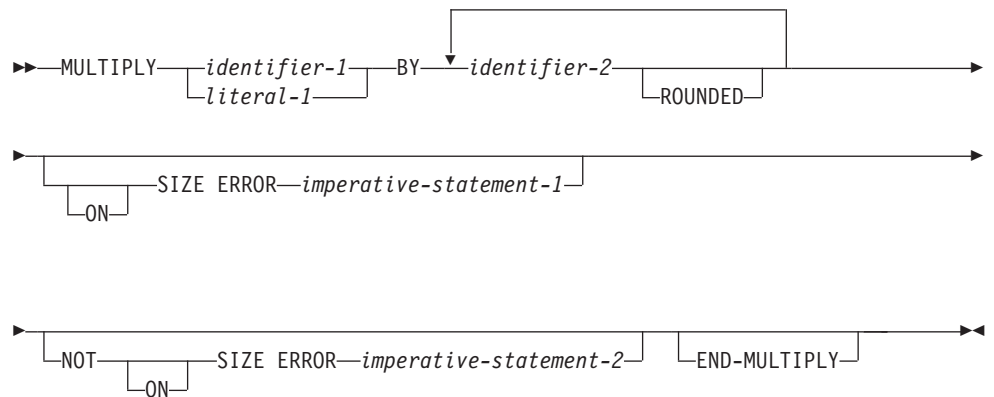
- A move to an alphanumeric group item from one of the following:
 - any elementary data item that is valid as a sending item in the MOVE statement
 - a national group item
 - a literal
 - a figurative constant
- A move from an alphanumeric group item to the following:
 - any elementary data item that is valid as a receiving item in the MOVE statement
 - a national group item
 - an alphanumeric group item

A group move is treated as though it were an alphanumeric-to-alphanumeric elementary move, except that there is no conversion of data from one form of internal representation to another. In a group move, the receiving area is filled without consideration for the individual elementary items contained within either the sending area or the receiving area, except as noted in the OCCURS clause. (See “OCCURS clause” on page 192.)

MULTIPLY statement

The MULTIPLY statement multiplies numeric items and sets the values of data items equal to the results.

Format 1: MULTIPLY statement



In format 1, the value of *identifier-1* or *literal-1* is multiplied by the value of *identifier-2*; the product is then placed in *identifier-2*. For each successive occurrence of *identifier-2*, the multiplication takes place in the left-to-right order in which *identifier-2* is specified.

details on arithmetic intermediate results, *Appendix A. Intermediate results and arithmetic precision* in the *COBOL for AIX Programming Guide*.

ROUNDED phrase

For formats 1 and 2, see “ROUNDED phrase” on page 292.

SIZE ERROR phrases

For formats 1 and 2, see “SIZE ERROR phrases” on page 293.

END-MULTIPLY phrase

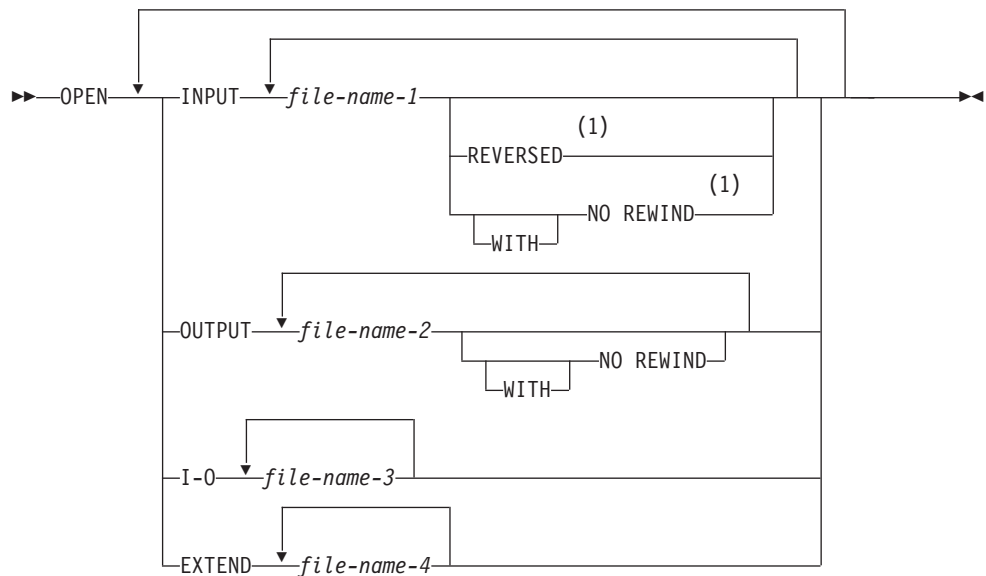
This explicit scope terminator serves to delimit the scope of the MULTIPLY statement. END-MULTIPLY permits a conditional MULTIPLY statement to be nested in another conditional statement. END-MULTIPLY can also be used with an imperative MULTIPLY statement.

For more information, see “Delimited scope statements” on page 289.

OPEN statement

The OPEN statement initiates the processing of files. It also checks or writes labels, or both.

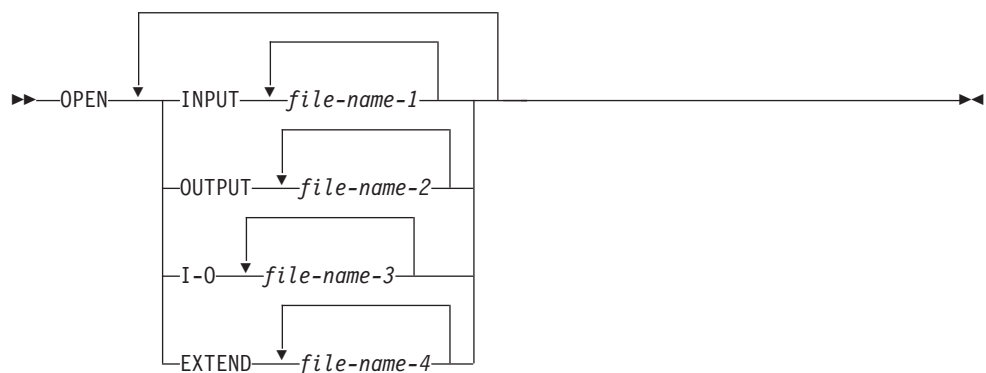
Format 1: OPEN statement for sequential files



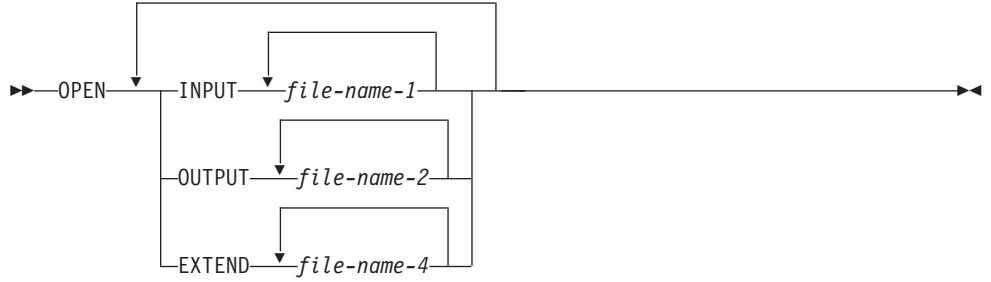
Notes:

- 1 The REVERSED and WITH NO REWIND phrases are syntax checked, but have no effect on the execution of the program.

Format 2: OPEN statement for indexed and relative files



Format 3: OPEN statement for line-sequential files



The phrases INPUT, OUTPUT, I-O, and EXTEND specify the mode to be used for opening the file. At least one of the phrases INPUT, OUTPUT, I-O, or EXTEND must be specified with the OPEN keyword. The INPUT, OUTPUT, I-O, and EXTEND phrases can appear in any order.

INPUT

Permits input operations.

OUTPUT

Permits output operations. This phrase can be specified when the file is being created.

Do not specify OUTPUT for files that contain records. The file will be replaced by new data.

If the OUTPUT phrase is specified for a file that already contains records and either the SdU or SFS file system is used, the file will be deleted and recreated.

I-O Permits both input and output operations. The I-O phrase can be specified only for files assigned to direct access devices.

The I-O phrase is not valid for line-sequential files.

EXTEND

Permits output operations that append to or create a file.

The EXTEND phrase is allowed for sequential access files only if the new data is written in ascending sequence. The EXTEND phrase is allowed for files that specify the LINAGE clause.

file-name-1, file-name-2, file-name-3, file-name-4

Designate a file upon which the OPEN statement is to operate. If more than one file is specified, the files need not have the same organization or access mode. Each file-name must be defined in an FD entry in the data division and must not name a sort or merge file. The FD entry must be equivalent to the information supplied when the file was defined.

REVERSED

The REVERSED phrase is syntax checked, but has no effect on the execution of the program.

NO REWIND

The NO REWIND phrase is syntax checked, but has no effect on the execution of the program.

For information on file sizes, see Appendix B, “Compiler limits,” on page 577.

General rules

- If a file opened with the INPUT phrase is an optional file that is not available, the OPEN statement sets the file position indicator to indicate that an optional input file is not available.
- Execution of an OPEN INPUT or OPEN I-O statement sets the file position indicator:
 - For indexed files, to the characters with the lowest ordinal position in the collating sequence associated with the file.
 - For sequential and relative files, to 1.
- When the EXTEND phrase is specified, the OPEN statement positions the file immediately after the last record written in the file. (The record with the highest prime record key value for indexed files or relative key value for relative files is considered the last record.) Subsequent WRITE statements add records as if the file were opened OUTPUT. The EXTEND phrase can be specified when a file is being created; it can also be specified for a file that contains records, or that has contained records that have been deleted. For more information, see note 1 in the “OPEN statement notes” and SELECT OPTIONAL in the “SELECT clause” on page 126.
- When the EXTEND phrase is not specified, the OPEN statement positions the file at its beginning.

Label records

Label processing is not supported. A warning message is issued if any of the following language elements are encountered:

- LABEL RECORDS IS data-name
- USE...AFTER...LABEL PROCEDURE
- GO TO MORE-LABELS

OPEN statement notes

1. The successful execution of an OPEN statement determines the availability of the file and results in that file being in open mode. A file is *available* if it is physically present and recognized by the input-output control system. The following table shows the results of opening available and unavailable files.

Table 51. Availability of a file

Opened as	File is available	File is unavailable
INPUT	Normal open	Open is unsuccessful. (file status 35)
INPUT (optional file)	Normal open	Normal open; the first read causes the at end condition or the invalid key condition. (file status 05)
I-O	Normal open	Open is unsuccessful. (file status 35)
I-O (optional file)	Normal open	Open causes the file to be created. (file status 05)
OUTPUT	Normal open; the file contains no records	Open causes the file to be created.
EXTEND	Normal open	Open is unsuccessful. (file status 35)

Table 51. Availability of a file (continued)

Opened as	File is available	File is unavailable
EXTEND (optional file)	Normal open	Open causes the file to be created. (file status 05)

2. The successful execution of the OPEN statement places the file in open status and makes the associated record area available to the program.
3. The OPEN statement does not obtain or release the first data record.
4. You can move data to or from the record area only when the file is in open status.
5. An OPEN statement must be successfully executed prior to the execution of any of the permissible input-output statements, except a SORT or MERGE statement with the USING or GIVING phrase. In the following table, an 'X' indicates that the specified statement can be used with the open mode given at the top of the column.

Table 52. Permissible statements for sequential files

Statement	Input open mode	Output open mode	I-O open mode	Extend open mode
READ	X		X	
WRITE		X		X
REWRITE			X	

In the following table, an 'X' indicates that the specified statement, used in the access mode given for that row, can be used with the open mode given at the top of the column.

Table 53. Permissible statements for indexed and relative files

File access mode	Statement	Input open mode	Output open mode	I-O open mode	Extend open mode
Sequential	READ	X		X	
	WRITE		X		X
	REWRITE			X	
	START	X		X	
	DELETE			X	
Random	READ	X		X	
	WRITE		X	X	
	REWRITE			X	
	START				
	DELETE			X	
Dynamic	READ	X		X	
	WRITE		X	X	
	REWRITE			X	
	START	X		X	
	DELETE			X	

In the following table, an 'X' indicates that the specified statement can be used with the open mode given at the top of the column.

Table 54. Permissible statements for line-sequential files

Statement	Input open mode	Output open mode	I-O open mode	Extend open mode
READ	X			
WRITE		X		X
REWRITE				

1. A file can be opened for INPUT, OUTPUT, I-O, or EXTEND (sequential and line-sequential files only) in the same program. After the first OPEN statement execution for a given file, each subsequent OPEN statement execution must be preceded by a successful CLOSE file statement execution without the LOCK phrase.
2. If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the OPEN statement is executed.
3. If an OPEN statement is issued for a file already in the open status, the EXCEPTION/ERROR procedure (if specified) for this file is executed.

PERFORM statement

The PERFORM statement transfers control explicitly to one or more procedures and implicitly returns control to the next executable statement after execution of the specified procedures is completed.

The PERFORM statement is:

An out-of-line PERFORM statement

When *procedure-name-1* is specified.

An in-line PERFORM statement

When *procedure-name-1* is omitted.

An in-line PERFORM must be delimited by the END-PERFORM phrase.

The in-line and out-of-line formats cannot be combined. For example, if *procedure-name-1* is specified, imperative statements and the END-PERFORM phrase must not be specified.

The PERFORM statement formats are:

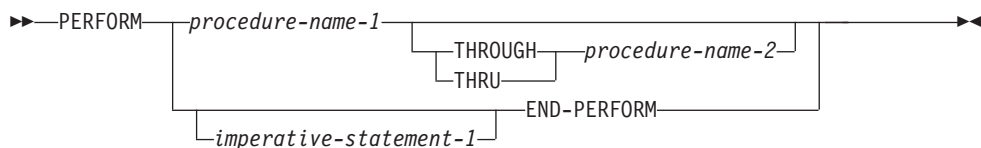
- Basic PERFORM
- TIMES phrase PERFORM
- UNTIL phrase PERFORM
- VARYING phrase PERFORM

Basic PERFORM statement

The procedures referenced in the basic PERFORM statement are executed once, and control then passes to the next executable statement following the PERFORM statement.

Note: A PERFORM statement must not cause itself to be executed. A recursive PERFORM statement can cause unpredictable results.

Format 1: Basic PERFORM statement



procedure-name-1 , *procedure-name-2*

Must name a section or paragraph in the procedure division.

When both *procedure-name-1* and *procedure-name-2* are specified, if either is a procedure-name in a declarative procedure, both must be procedure-names in the same declarative procedure.

If *procedure-name-1* is specified, *imperative-statement-1* and the END-PERFORM phrase must not be specified.

If *procedure-name-1* is omitted, *imperative-statement-1* and the END-PERFORM phrase must be specified.

imperative-statement-1

The statements to be executed for an in-line PERFORM

An in-line PERFORM statement functions according to the same general rules as an otherwise identical out-of-line PERFORM statement, except that statements contained within the in-line PERFORM are executed in place of the statements contained within the range of *procedure-name-1* (through *procedure-name-2*, if specified). Unless specifically qualified by the word *in-line* or the word *out-of-line*, all the rules that apply to the out-of-line PERFORM statement also apply to the in-line PERFORM.

Whenever an out-of-line PERFORM statement is executed, control is transferred to the first statement of the procedure named *procedure-name-1*. Control is always returned to the statement following the PERFORM statement. The point from which this control is returned is determined as follows:

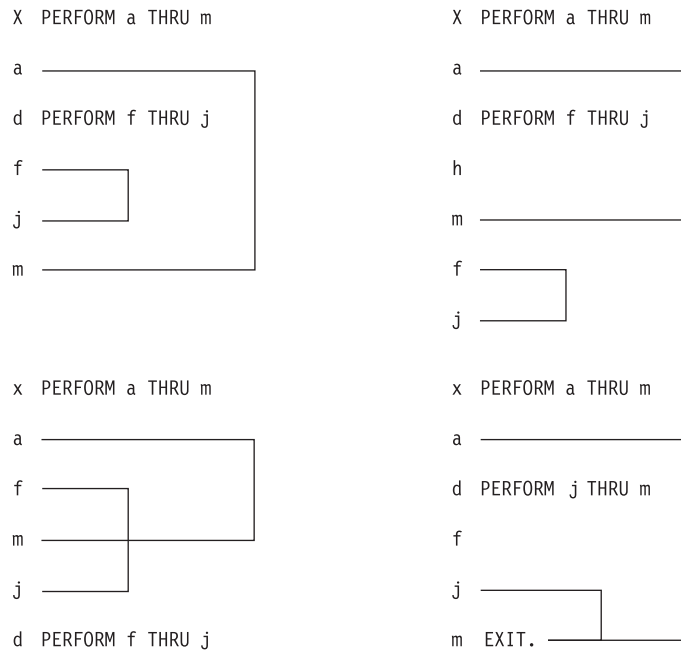
- If *procedure-name-1* is a paragraph name and *procedure-name-2* is not specified, the return is made after the execution of the last statement of the *procedure-name-1* paragraph.
- If *procedure-name-1* is a section name and *procedure-name-2* is not specified, the return is made after the execution of the last statement of the last paragraph in the *procedure-name-1* section.
- If *procedure-name-2* is specified and it is a paragraph name, the return is made after the execution of the last statement of the *procedure-name-2* paragraph.
- If *procedure-name-2* is specified and it is a section name, the return is made after the execution of the last statement of the last paragraph in the *procedure-name-2* section.

The only necessary relationship between *procedure-name-1* and *procedure-name-2* is that a consecutive sequence of operations is executed, beginning at the procedure named by *procedure-name-1* and ending with the execution of the procedure named by *procedure-name-2*.

PERFORM statements can be specified within the performed procedure. If there are two or more logical paths to the return point, then *procedure-name-2* can name a paragraph that consists only of an EXIT statement; all the paths to the return point must then lead to this paragraph.

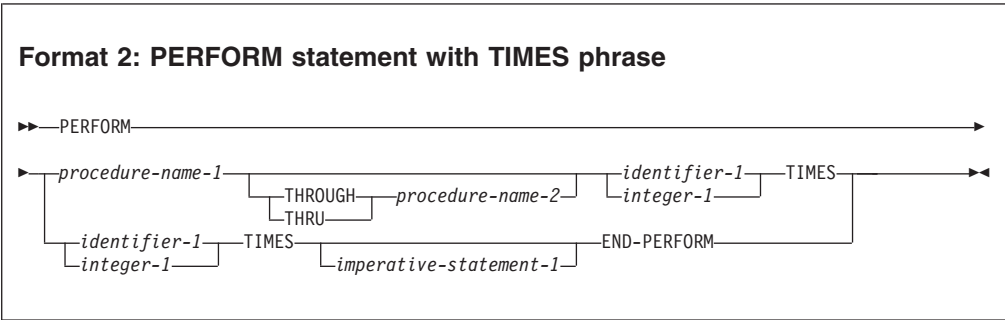
When the performed procedures include another PERFORM statement, the sequence of procedures associated with the embedded PERFORM statement must be totally included in or totally excluded from the performed procedures of the first PERFORM statement. That is, an active PERFORM statement whose execution point begins within the range of performed procedures of another active PERFORM statement must not allow control to pass through the exit point of the other active PERFORM statement. However, two or more active PERFORM statements can have a common exit.

The following figure illustrates valid sequences of execution for PERFORM statements.



END-PERFORM

PERFORM with TIMES phrase



identifier-1

Must name an integer item. *identifier-1* cannot be a windowed date field.

If *identifier-1* is zero or a negative number at the time the PERFORM statement is initiated, control passes to the statement following the PERFORM statement.

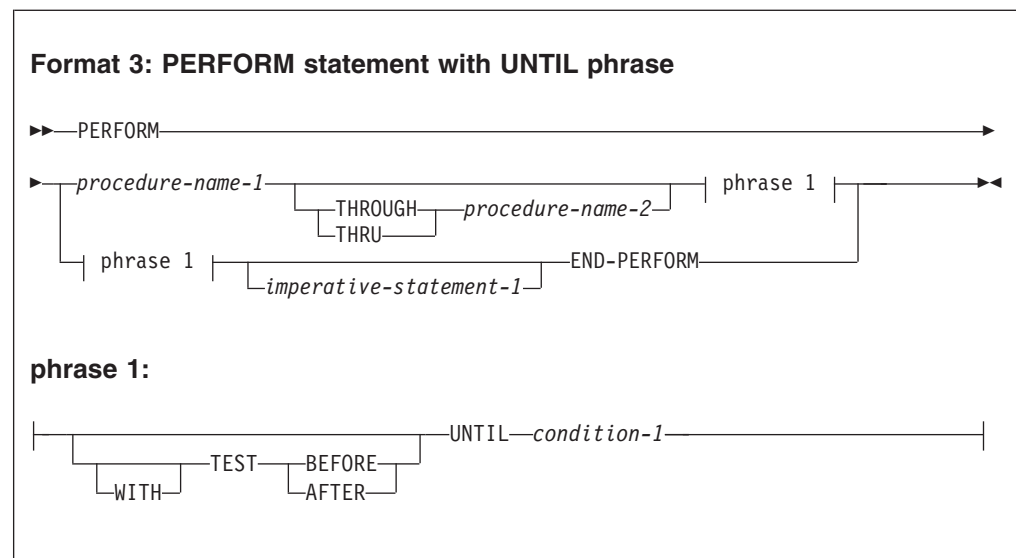
After the PERFORM statement has been initiated, any change to *identifier-1* has no effect in varying the number of times the procedures are initiated.

integer-1

Can be a positive signed integer.

PERFORM with UNTIL phrase

In the UNTIL phrase format, the procedures referred to are performed *until* the condition specified by the UNTIL phrase is true. Control is then passed to the next executable statement following the PERFORM statement.



If *procedure-name-1* is specified, *imperative-statement-1* and the END-PERFORM phrase must not be specified.

condition-1

Can be any condition described under “Conditional expressions” on page 262. If the condition is true at the time the PERFORM statement is initiated, the specified procedures are not executed.

Any subscripting associated with the operands specified in *condition-1* is evaluated each time the condition is tested.

If the TEST BEFORE phrase is specified or assumed, the condition is tested before any statements are executed (corresponds to DO WHILE).

If the TEST AFTER phrase is specified, the statements to be performed are executed at least once before the condition is tested (corresponds to DO UNTIL).

literal-1 through literal-4

Must represent a numeric literal.

condition-1, condition-2

Can be any condition described under “Conditional expressions” on page 262. If the condition is true at the time the PERFORM statement is initiated, the specified procedures are not executed.

After the conditions specified in the UNTIL phrase are satisfied, control is passed to the next executable statement following the PERFORM statement.

If any of the operands specified in *condition-1* or *condition-2* is subscripted, reference modified, or is a function-identifier, the subscript, reference-modifier, or function is evaluated each time the condition is tested.

Floating-point data items and literals can be used anywhere a numeric data item or literal can be specified.

When TEST BEFORE is indicated, all specified conditions are tested before the first execution, and the statements to be performed are executed, if at all, only when *all* specified tests fail. When TEST AFTER is indicated, the statements to be performed are executed at least once, before any condition is tested.

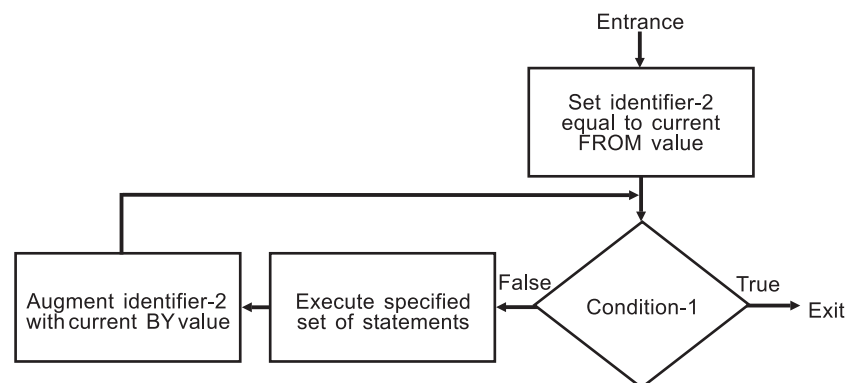
If neither the TEST BEFORE nor the TEST AFTER phrase is specified, the TEST BEFORE phrase is assumed.

Varying identifiers

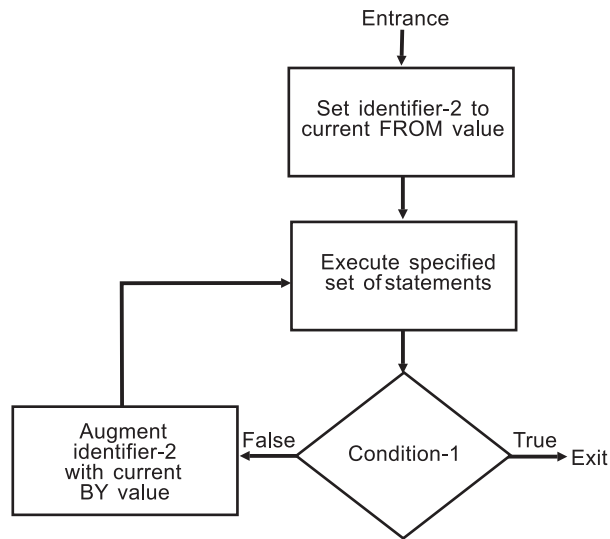
The way in which operands are increased or decreased depends on the number of variables specified. In the following discussion, every reference to *identifier-n* refers equally to *index-name-n* (except when *identifier-n* is the object of the BY phrase).

If *identifier-2* or *identifier-5* is subscripted, the subscripts are evaluated each time the content of the data item referenced by the identifier is set or augmented. If *identifier-3*, *identifier-4*, *identifier-6*, or *identifier-7* is subscripted, the subscripts are evaluated each time the content of the data item referenced by the identifier is used in a setting or an augmenting operation.

The following figure illustrates the logic of the PERFORM statement when an identifier is varied with TEST BEFORE.



The following figure illustrates the logic of the PERFORM statement when an identifier is varied with TEST AFTER.



Varying two identifiers

```

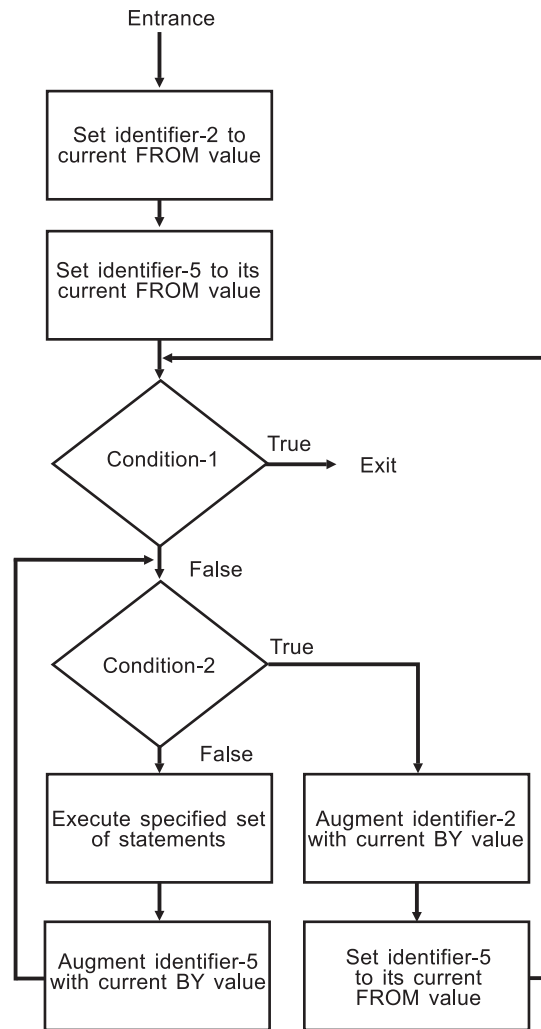
PERFORM PROCEDURE-NAME-1 THROUGH PROCEDURE-NAME-2
  VARYING IDENTIFIER-2 FROM IDENTIFIER-3
    BY IDENTIFIER-4 UNTIL CONDITION-1
  AFTER IDENTIFIER-5 FROM IDENTIFIER-6
    BY IDENTIFIER-7 UNTIL CONDITION-2
  
```

1. *identifier-2* and *identifier-5* are set to their initial values, *identifier-3* and *identifier-6*, respectively.
2. *condition-1* is evaluated as follows:
 - a. If it is false, steps 3 through 7 are executed.
 - b. If it is true, control passes directly to the statement following the PERFORM statement.
3. *condition-2* is evaluated as follows:
 - a. If it is false, steps 4 through 6 are executed.
 - b. If it is true, *identifier-2* is augmented by *identifier-4*, *identifier-5* is set to the current value of *identifier-6*, and step 2 is repeated.
4. *procedure-name-1* and *procedure-name-2* are executed once (if specified).
5. *identifier-5* is augmented by *identifier-7*.
6. Steps 3 through 5 are repeated until *condition-2* is true.
7. Steps 2 through 6 are repeated until *condition-1* is true.

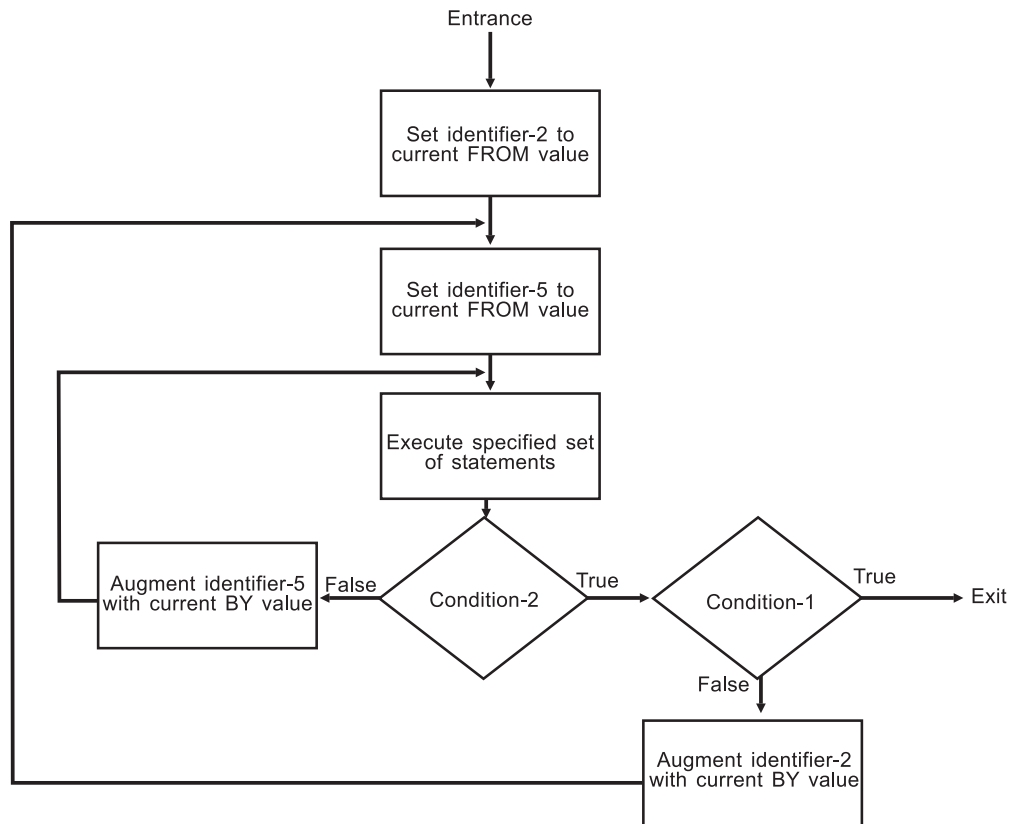
At the end of PERFORM statement execution:

- *identifier-5* contains the current value of *identifier-6*.
- *identifier-2* has a value that exceeds the last-used setting by the increment or decrement value (unless *condition-1* was true at the beginning of PERFORM statement execution, in which case, *identifier-2* contains the current value of *identifier-3*).

The following figure illustrates the logic of the PERFORM statement when two identifiers are varied with TEST BEFORE.



The following figure illustrates the logic of the PERFORM statement when two identifiers are varied with TEST AFTER.



Varying three identifiers

```

PERFORM PROCEDURE-NAME-1 THROUGH PROCEDURE-NAME-2
  VARYING IDENTIFIER-2 FROM IDENTIFIER-3
    BY IDENTIFIER-4 UNTIL CONDITION-1
  AFTER IDENTIFIER-5 FROM IDENTIFIER-6
    BY IDENTIFIER-7 UNTIL CONDITION-2
  AFTER IDENTIFIER-8 FROM IDENTIFIER-9
    BY IDENTIFIER-10 UNTIL CONDITION-3
  
```

The actions are the same as those for two identifiers, except that *identifier-8* goes through the complete cycle each time that *identifier-5* is augmented by *identifier-7*, which, in turn, goes through a complete cycle each time that *identifier-2* is varied.

At the end of PERFORM statement execution:

- *identifier-5* and *identifier-8* contain the current values of *identifier-6* and *identifier-9*, respectively.
- *identifier-2* has a value exceeding its last-used setting by one increment/decrement value (unless *condition-1* was true at the beginning of PERFORM statement execution, in which case *identifier-2* contains the current value of *identifier-3*).

Varying more than three identifiers

You can produce analogous PERFORM statement actions to the example above with the addition of up to four AFTER phrases.

Varying phrase rules

No matter how many variables are specified, the following rules apply:

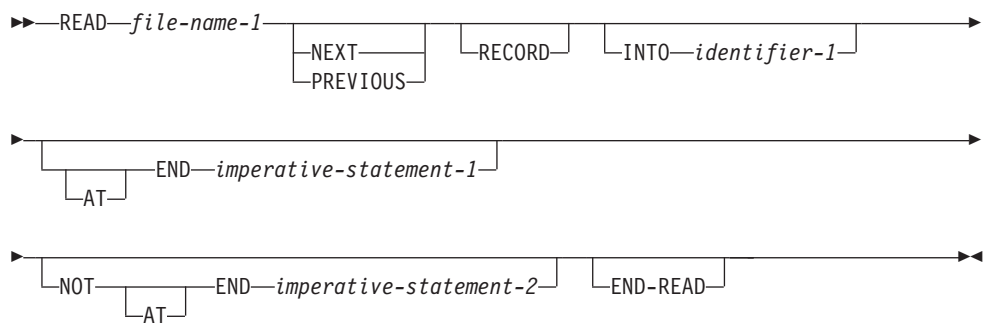
- In the VARYING or AFTER phrases, when an index-name is specified:
 - The index-name is initialized and incremented or decremented according to the rules under “INDEX phrase” on page 234. (See also “SET statement” on page 427.)
 - In the associated FROM phrase, an identifier must be described as an integer and have a positive value; a literal must be a positive integer.
 - In the associated BY phrase, an identifier must be described as an integer; a literal must be a nonzero integer.
- In the FROM phrase, when an index-name is specified:
 - In the associated VARYING or AFTER phrase, an identifier must be described as an integer. It is initialized as described in the SET statement.
 - In the associated BY phrase, an identifier must be described as an integer and have a nonzero value; a literal must be a nonzero integer.
- In the BY phrase, identifiers and literals must have nonzero values.
- Changing the values of identifiers or index-names in the VARYING, FROM, and BY phrases during execution changes the number of times the procedures are executed.

READ statement

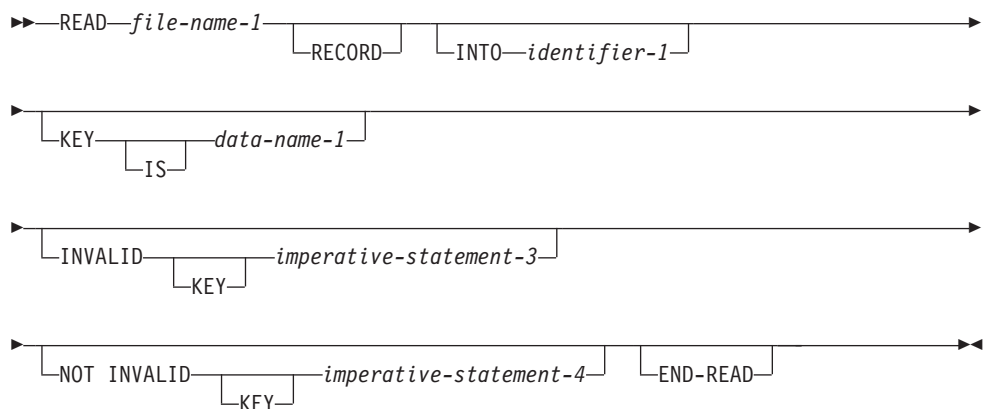
For sequential access, the READ statement makes the next logical record from a file available to the object program. For random access, the READ statement makes a specified record from a direct-access file available to the object program.

When the READ statement is executed, the associated file must be open in INPUT or I-O mode.

Format 1: READ statement for sequential retrieval



Format 2: READ statement for random retrieval



file-name-1

Must be defined in a data division FD entry.

NEXT RECORD

Reads the next record in the logical sequence of records. NEXT is optional when the access mode is sequential, and has no effect on READ statement execution.

You must specify either the NEXT phrase or the PREVIOUS phrase to retrieve records sequentially from files in dynamic access mode.

PREVIOUS RECORD

Reads the previous record in the logical sequence of records. PREVIOUS applies to indexed and relative files with DYNAMIC access mode.

To retrieve records sequentially, you must specify either the NEXT phrase or the PREVIOUS phrase for files in dynamic access mode.

If you specify READ...PREVIOUS and no previous logical record exists, the AT END condition occurs and the READ statement is unsuccessful.

When you specify READ...PREVIOUS, the setting of the file position indicator is used to determine which record to make available according to the following rules:

- If the file position indicator indicates that no valid previous record has been established, the READ is unsuccessful.
- If the file position indicator is positioned by the execution of an OPEN statement, the AT END condition occurs.
- If the file position indicator is established by a previous START statement, the first existing record in the file whose relative record number (if a relative file) or whose key value (if an indexed file) is less than or equal to the file position indicator is selected.
- If the file position indicator is established by a previous READ statement, the first existing record in the file whose relative record number (if a relative file) or whose key value (if an indexed file) is less than the file position indicator is selected.

INTO *identifier-1*

identifier-1 is the receiving field.

identifier-1 must be a valid receiving field for the selected sending record description entry in accordance with the rules of the MOVE statement.

The record areas associated with *file-name-1* and *identifier-1* must not be the same storage area.

When there is only one record description associated with *file-name-1* or all the records and the data item referenced by *identifier-1* describe an elementary alphanumeric item or an alphanumeric group item, the result of the execution of a READ statement with the INTO phrase is equivalent to the application of the following rules in the order specified:

- The execution of the same READ statement without the INTO phrase.
- The current record is moved from the record area to the area specified by *identifier-1* according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified for the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the READ statement was unsuccessful. Any subscripting or reference modification associated with *identifier-1* is evaluated after the record has been read and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by *identifier-1*.

If *identifier-1* is a date field, then the implied MOVE statement is performed according to the behavior described under “Moves involving date fields” on page 385.

When there are multiple record descriptions associated with *file-name-1* and they do not all describe an alphanumeric group item or elementary alphanumeric item, the following rules apply:

1. If the file referenced by *file-name-1* is described as containing variable-length records, a group move will take place.
2. If the file referenced by *file-name-1* is described as containing fixed-length records, a move will take place according to the rules for a MOVE statement using, as a sending field description, the record that specifies the largest number of character positions. If more than one such record exists, the sending field record selected will be the one among those records that appears first under the description of *file-name-1*.

KEY IS phrase

The KEY IS phrase can be specified only for indexed files. *data-name-1* must identify a record key associated with *file-name-1*. *data-name-1* can be qualified; it cannot be subscripted.

AT END phrases

For sequential access, both the AT END phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

For information about at-end condition processing, see “AT END condition” on page 409.

INVALID KEY phrases

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

For information about INVALID KEY phrase processing, see “Invalid key condition” on page 300.

END-READ phrase

This explicit scope terminator serves to delimit the scope of the READ statement. END-READ permits a conditional READ statement to be nested in another conditional statement. END-READ can also be used with an imperative READ statement. For more information, see “Delimited scope statements” on page 289.

Multiple record processing

If more than one record description entry is associated with *file-name-1*, those records automatically share the same storage area; that is, they are implicitly redefined. After a READ statement is executed, only those data items within the range of the current record are replaced; data items stored beyond that range are undefined. The following example illustrates this concept. If the range of the current record exceeds the record description entries for *file-name-1*, the record is truncated on the right to the maximum size. In either of these cases, the READ statement is successful and the I-O status is set to 04 indicating a record length conflict has occurred.

The following example shows two record areas of different sizes in an FD. When a shorter record is read, the content of the remaining record area is undefined.

```
FD INPUT-FILE LABEL RECORD OMITTED.  
01 RECORD-1 PICTURE X(30).  
01 RECORD-2 PICTURE X(20).
```

Content of input area when READ statement is executed:

ABCDEFGHIJKLMNOPQRSTUVWXYZ1234

Content of record being read in (RECORD-2):

01234567890123456789

Content of input area after READ statement is executed:

01234567890123456789??????????

The "?" characters are undefined characters in input area.

Sequential access mode

Format 1 must be used for all files in sequential access mode.

Execution of a format-1 READ statement retrieves the next logical record from the file. The next record accessed is determined by the file organization.

Sequential files

The NEXT RECORD is the next record in a logical sequence of records. The NEXT phrase need not be specified; it has no effect on READ statement execution.

If SELECT OPTIONAL is specified in the file-control entry for this file, and the file is unavailable during this execution of the object program, execution of the first READ statement causes an at-end condition; however, since no file is available, the system-defined end-of-file processing is not performed.

AT END condition:

If the file position indicator indicates that no next logical record exists, or that an optional input file is not available, the following occurs in the order specified:

1. A value derived from the setting of the file position indicator is placed into the I-O status associated with *file-name-1* to indicate the at-end condition.
2. If the AT END phrase is specified in the statement causing the condition, control is transferred to *imperative-statement-1* in the AT END phrase. Any USE AFTER STANDARD EXCEPTION procedure associated with *file-name-1* is not executed.
3. If the AT END phrase is not specified and an applicable USE AFTER STANDARD EXCEPTION procedure exists, the procedure is executed. Return from that procedure is to the next executable statement following the end of the READ statement.

Both the AT END phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

When the at-end condition occurs, execution of the READ statement is unsuccessful. The contents of the associated record area are undefined and the file position indicator is set to indicate that no valid next record has been established.

If an at-end condition does not occur during the execution of a READ statement, the AT END phrase is ignored, if specified, and the following actions occur:

1. The file position indicator is set and the I-O status associated with *file-name-1* is updated.
2. If an exception condition that is not an at-end condition exists, control is transferred to the end of the READ statement after the execution of any USE AFTER STANDARD EXCEPTION procedure applicable to *file-name-1*.
If no USE AFTER STANDARD EXCEPTION procedure is specified, control is transferred to the end of the READ statement or to *imperative-statement-2*, if specified.
3. If no exception condition exists, the record is made available in the record area and any implicit move resulting from the presence of an INTO phrase is executed. Control is transferred to the end of the READ statement or to *imperative-statement-2*, if specified. In the latter case, execution continues according to the rules for each statement specified in *imperative-statement-2*. If a procedure branching or conditional statement which causes explicit transfer of control is executed, control is transferred in accordance with the rules for that statement; otherwise, upon completion of the execution of *imperative-statement-2*, control is transferred to the end of the READ statement.

After the unsuccessful execution of a READ statement, the contents of the associated record area are undefined and the file position indicator is set to indicate that no valid next record has been established. Attempts to access or move data into the record area after an unsuccessful read can result in a segmentation violation.

Indexed or relative files

The NEXT RECORD is the next logical record in the key sequence.

The PREVIOUS RECORD is the preceding logical record in the key sequence.

For indexed files, the key sequence is the sequence of ascending values of the current key of reference. For relative files, the key sequence is the sequence of ascending values of relative record numbers for records that exist in the file.

Before the READ statement is executed, the file position indicator must have been set by a successful OPEN, START, or READ statement. When the READ statement is executed, the record indicated by the file position indicator is made available if it is still accessible through the path indicated by the file position indicator.

If the record is no longer accessible (because it has been deleted, for example), the file position indicator is updated to point to the next (or previous) existing record in the file, and that record is made available.

For files in sequential access mode, the NEXT phrase need not be specified.

For files in dynamic access mode, the NEXT phrase (or the PREVIOUS phrase) must be specified for sequential record retrieval.

AT END condition:

This condition exists when the file position indicator indicates that no next logical record exists (or no previous record exists) or that an optional input file is not available. See the discussion of PREVIOUS RECORD above.

If neither an at-end nor an invalid key condition occurs during the execution of a READ statement, the AT END or the INVALID KEY phrase is ignored, if specified. The same actions occur as when the at-end condition does not occur with sequential files (see “AT END condition” on page 409).

Sequentially accessed indexed files:

When an ALTERNATE RECORD KEY with DUPLICATES is the key of reference, file records with duplicate key values are made available in the order in which they were placed in the file.

Sequentially accessed relative files:

If the RELATIVE KEY clause is specified for this file, READ statement execution updates the RELATIVE KEY data item to indicate the relative record number of the record being made available.

Random access mode

Format 2 must be specified for indexed and relative files in random access mode, and also for files in the dynamic access mode when record retrieval is random.

Execution of the READ statement depends on the file organization, as explained in the following sections.

Indexed files

Execution of a format-2 READ statement causes the value of the key of reference to be compared with the value of the corresponding key data item in the file records, until the first record having an equal value is found. The file position indicator is positioned to this record, which is then made available. If no record can be so identified, an INVALID KEY condition exists, and READ statement execution is unsuccessful. (See “Invalid key condition” on page 300 for details of the invalid key condition.)

If the KEY phrase is not specified, the prime RECORD KEY becomes the key of reference for this request. When dynamic access is specified, the prime RECORD KEY is also used as the key of reference for subsequent executions of sequential READ statements, until a different key of reference is established.

When the KEY phrase is specified, *data-name-1* becomes the key of reference for this request. When dynamic access is specified, this key of reference is used for subsequent executions of sequential READ statements, until a different key of reference is established.

Relative files

Execution of a format-2 READ statement sets the file position indicator pointer to the record whose relative record number is contained in the RELATIVE KEY data item, and makes that record available.

If the file does not contain such a record, the INVALID KEY condition exists, and READ statement execution is unsuccessful. (See “Invalid key condition” on page 300 for details of the invalid key condition).

The KEY phrase must not be specified for relative files.

Dynamic access mode

For files with indexed or relative organization, dynamic access mode can be specified in the file-control entry. In dynamic access mode, either sequential or random record retrieval can be used, depending on the format used.

Format 1 with the NEXT phrase must be specified for sequential retrieval. All other rules for sequential access apply.

READ statement notes:

- If the FILE-STATUS clause is specified in the file-control entry, the associated file status key is updated when the READ statement is executed.
- After unsuccessful READ statement execution, the contents of the associated record area and the value of the file position indicator are undefined. Attempts to access or move data into the record area after an unsuccessful read can result in a segmentation violation.

RELEASE statement

The RELEASE statement transfers records from an input/output area to the initial phase of a sorting operation.

The RELEASE statement can be used only within the range of an INPUT PROCEDURE associated with a SORT statement.

Format: RELEASE

►►—RELEASE—*record-name-1*—┐
└FROM—*identifier-1*—┘◄◄

Within an INPUT PROCEDURE, at least one RELEASE statement must be specified.

When the RELEASE statement is executed, the current contents of *record-name-1* are placed in the sort file. This makes the record available to the initial phase of the sorting operation.

record-name-1

Must specify the name of a logical record in a sort-merge file description entry (SD). *record-name-1* can be qualified.

FROM phrase

The result of the execution of the RELEASE statement with the FROM *identifier-1* phrase is equivalent to the execution of the following statements in the order specified.

```
MOVE identifier-1 to record-name-1.  
RELEASE record-name-1.
```

The MOVE is performed according to the rules for the MOVE statement without the CORRESPONDING phrase.

identifier-1

identifier-1 must reference one of the following:

- An entry in the working-storage section, the local-storage section, or the linkage section
- A record description for another previously opened file
- An alphanumeric or national function.

identifier-1 must be a valid sending item with *record-name-1* as the receiving item in accordance with the rules of the MOVE statement.

identifier-1 and *record-name-1* must not refer to the same storage area.

After the RELEASE statement is executed, the information is still available in *identifier-1*. (See "INTO and FROM phrases" on page 301 under "Common processing facilities".)

If the RELEASE statement is executed without specifying the SD entry for *file-name-1* in a SAME RECORD AREA clause, the information in *record-name-1* is no longer available.

If the SD entry is specified in a SAME RECORD AREA clause, *record-name-1* is still available as a record of the other files named in that clause.

When FROM *identifier-1* is specified, the information is still available in *identifier-1*.

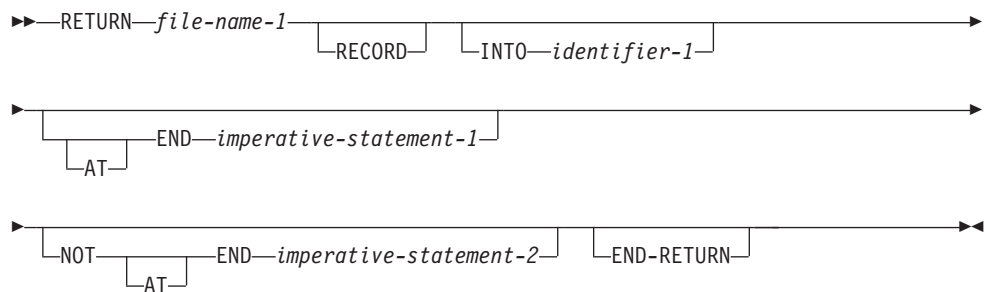
When control passes from the INPUT PROCEDURE, the sort file consists of all those records placed in it by execution of RELEASE statements.

RETURN statement

The RETURN statement transfers records from the final phase of a sorting or merging operation to an OUTPUT PROCEDURE.

The RETURN statement can be used only within the range of an OUTPUT PROCEDURE associated with a SORT or MERGE statement.

Format: RETURN statement



Within an OUTPUT PROCEDURE, at least one RETURN statement must be specified.

When the RETURN statement is executed, the next record from *file-name-1* is made available for processing by the OUTPUT PROCEDURE.

file-name-1

Must be described in a data division SD entry.

If more than one record description is associated with *file-name-1*, those records automatically share the same storage; that is, the area is implicitly redefined. After RETURN statement execution, only the contents of the current record are available. If any data items lie beyond the length of the current record, their contents are undefined.

INTO phrase

When there is only one record description associated with *file-name-1* or all the records and the data item referenced by *identifier-1* describe an elementary alphanumeric item or an alphanumeric group item, the result of the execution of a RETURN statement with the INTO phrase is equivalent to the application of the following rules in the order specified:

- The execution of the same RETURN statement without the INTO phrase.
- The current record is moved from the record area to the area specified by *identifier-1* according to the rules for the MOVE statement without the CORRESPONDING phrase. The size of the current record is determined by rules specified for the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement does not occur if the execution of the RETURN statement was unsuccessful. Any subscripting or reference modification associated with *identifier-1* is evaluated after the record has

been read and immediately before it is moved to the data item. The record is available in both the record area and the data item referenced by *identifier-1*.

When there are multiple record descriptions associated with *file-name-1* and they do not all describe an alphanumeric group item or elementary alphanumeric item, the following rules apply:

1. If the file referenced by *file-name-1* contains variable-length records, a group move takes place.
2. If the file referenced by *file-name-1* contains fixed-length records, a move takes place according to the rules for a MOVE statement using, as a sending field description, the record that specifies the largest number of character positions. If more than one such record exists, the sending field record selected will be the one among those records that appears first under the description of *file-name-1*.

identifier-1 must be a valid receiving field for the selected sending record description entry in accordance with the rules of the MOVE statement.

The record areas associated with *file-name-1* and *identifier-1* must not be the same storage area.

AT END phrases

The imperative-statement specified on the AT END phrase executes after all records have been returned from *file-name-1*. No more RETURN statements can be executed as part of the current output procedure.

If an at-end condition does not occur during the execution of a RETURN statement, then after the record is made available and after executing any implicit move resulting from the presence of an INTO phrase, control is transferred to the imperative statement specified by the NOT AT END phrase. If an at-end condition does occur, control is transferred to the end of the RETURN statement.

END-RETURN phrase

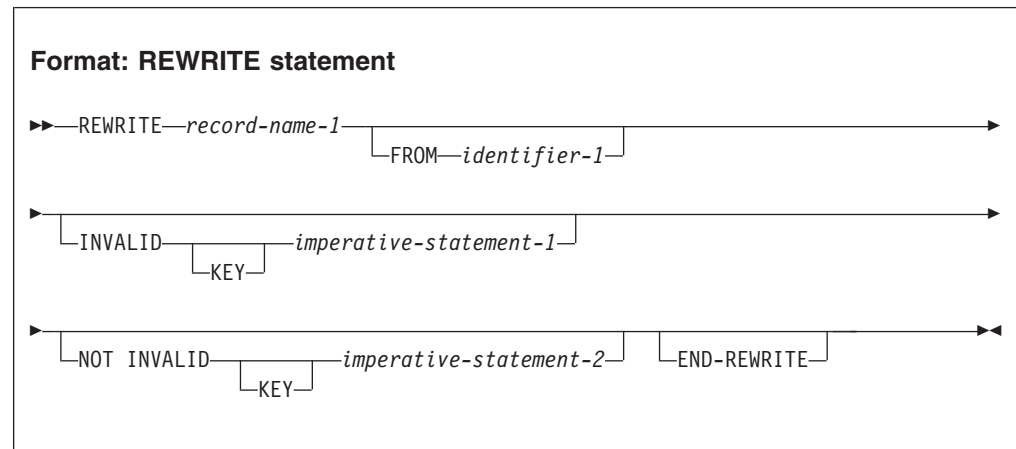
This explicit scope terminator serves to delimit the scope of the RETURN statement. END-RETURN permits a conditional RETURN statement to be nested in another conditional statement. END-RETURN can also be used with an imperative RETURN statement.

For more information, see “Delimited scope statements” on page 289.

REWRITE statement

The REWRITE statement logically replaces an existing record in a direct-access file. When the REWRITE statement is executed, the associated direct-access file must be open in I-O mode.

The REWRITE statement is not supported for line-sequential files.



record-name-1

Must be the name of a logical record in a data division FD entry. The record-name can be qualified.

FROM phrase

The result of the execution of the REWRITE statement with the FROM *identifier-1* phrase is equivalent to the execution of the following statements in the order specified.

```
MOVE identifier-1 TO record-name-1.
REWRITE record-name-1
```

The MOVE is performed according to the rules for the MOVE statement without the CORRESPONDING phrase.

identifier-1

identifier-1 can reference one of the following:

- A record description for another previously opened file
- An alphanumeric or national function
- A data item defined in the working-storage section, the local-storage section, or the linkage section

identifier-1 must be a valid sending item with *record-name-1* as the receiving item in accordance with the rules of the MOVE statement.

identifier-1 and *record-name-1* must not refer to the same storage area.

After the REWRITE statement is executed, the information is still available in *identifier-1* ("INTO and FROM phrases" on page 301 under "Common processing facilities").

INVALID KEY phrases

An INVALID KEY condition exists when:

- The access mode is sequential, and the value contained in the prime RECORD KEY of the record to be replaced does not equal the value of the prime RECORD KEY data item of the last-retrieved record from the file
- The value contained in the prime RECORD KEY does not equal that of any record in the file
- The value of an ALTERNATE RECORD KEY data item for which DUPLICATES is not specified is equal to that of a record already in the file

For details of invalid key processing, see “Invalid key condition” on page 300.

END-REWRITE phrase

This explicit scope terminator serves to delimit the scope of the REWRITE statement. END-REWRITE permits a conditional REWRITE statement to be nested in another conditional statement. END-REWRITE can also be used with an imperative REWRITE statement.

For more information, see “Delimited scope statements” on page 289.

Reusing a logical record

After successful execution of a REWRITE statement, the logical record is no longer available in *record-name-1* unless the associated file is named in a SAME RECORD AREA clause (in which case, the record is also available as a record of the other files named in the SAME RECORD AREA clause).

The file position indicator is not affected by execution of the REWRITE statement.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the REWRITE statement is executed.

Sequential files

For files in the sequential access mode, the last prior input/output statement executed for this file must be a successfully executed READ statement. When the REWRITE statement is executed, the record retrieved by that READ statement is logically replaced.

The number of character positions in *record-name-1* must equal the number of character positions in the record being replaced.

The INVALID KEY phrase must not be specified for a file with sequential organization. An EXCEPTION/ERROR procedure can be specified.

Indexed files

The number of character positions in *record-name-1* can be different from the number of character positions in the record being replaced.

When the access mode is sequential, the record to be replaced is specified by the value contained in the prime RECORD KEY. When the REWRITE statement is executed, this value must equal the value of the prime record key data item in the last record read from this file.

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

When the access mode is random or dynamic, the record to be replaced is specified by the value contained in the prime RECORD KEY.

Values of ALTERNATE RECORD KEY data items in the rewritten record can differ from those in the record being replaced. The system ensures that later access to the record can be based upon any of the record keys.

If an invalid key condition exists, the execution of the REWRITE statement is unsuccessful, the updating operation does not take place, and the data in *record-name-1* is unaffected. (See “Invalid key condition” on page 300 under “Common processing facilities”.)

Relative files

The number of character positions in *record-name-1* can be different from the number of character positions in the record being replaced.

For relative files in sequential access mode, the INVALID KEY phrase must not be specified. An EXCEPTION/ERROR procedure can be specified.

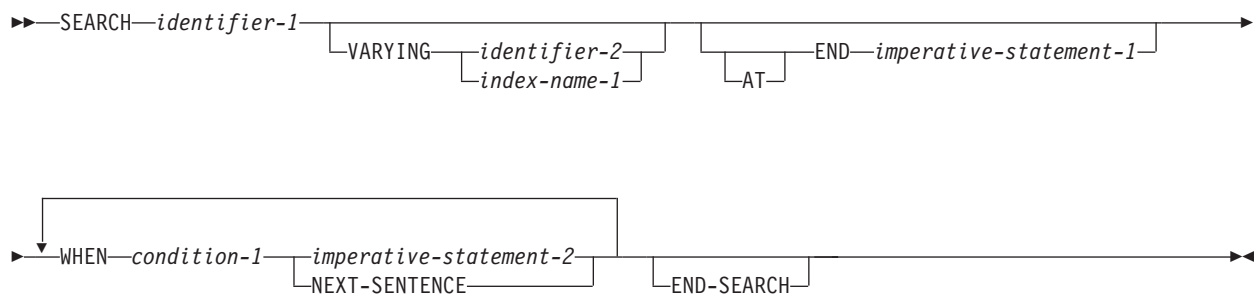
For relative files in random or dynamic access mode, the INVALID KEY phrase or an applicable EXCEPTION/ERROR procedure can be specified. Both can be omitted.

When the access mode is random or dynamic, the record to be replaced is specified in the RELATIVE KEY data item. If the file does not contain the record specified, an invalid key condition exists, and, if specified, the INVALID KEY imperative-statement is executed. (See “Invalid key condition” on page 300 under “Common processing facilities”.) The updating operation does not take place, and the data in *record-name* is unaffected.

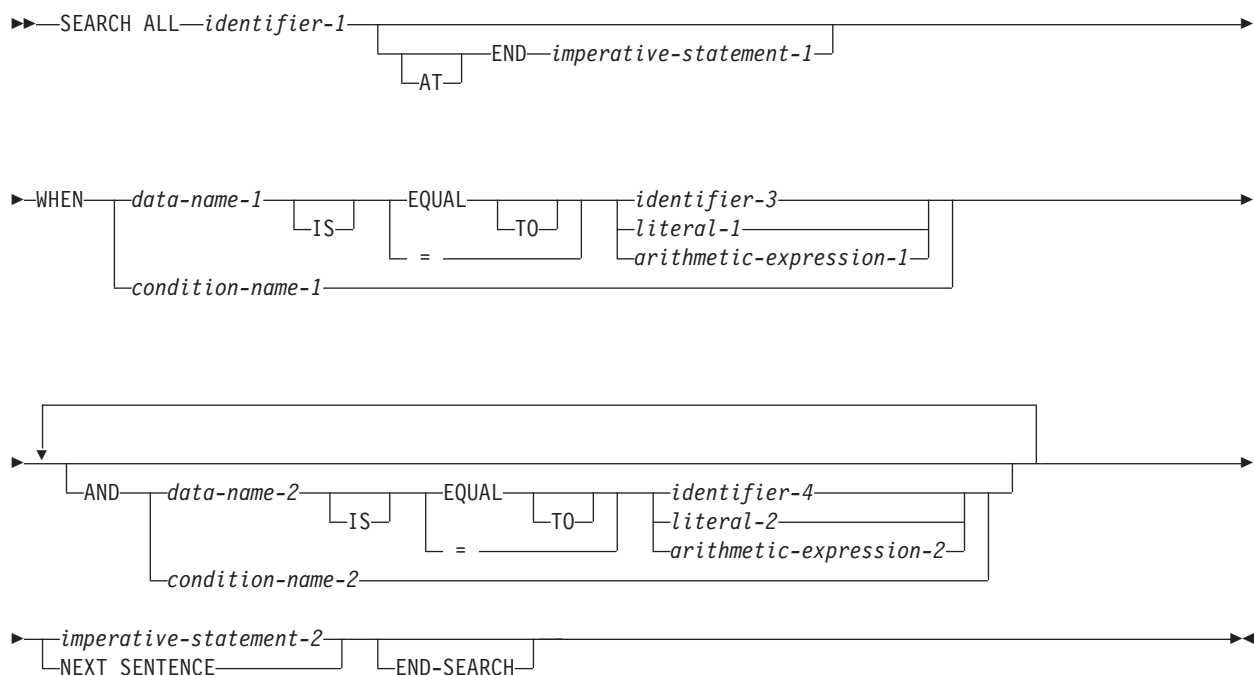
SEARCH statement

The SEARCH statement searches a table for an element that satisfies the specified condition and adjusts the associated index to indicate that element.

Format 1: SEARCH statement for serial search



Format 2: SEARCH statement for binary search



Use format 1 (serial search) when the table that you want to search has not been sorted. Use format 1 to search a sorted table when you want to search serially through the table or you want to control subscripts or indexes.

Use format 2 (binary search) when you want to efficiently search across all occurrences in a table. The table must previously have been sorted.

Serial search

identifier-1 (serial search)

identifier-1 identifies the table that is to be searched. *identifier-1* references all occurrences within that table.

The data description entry for *identifier-1* must contain an OCCURS clause.

The data description entry for *identifier-1* should contain an OCCURS clause with the INDEXED BY phrase, but a table can be searched using an index defined for an appropriately described different table.

identifier-1 can reference a data item that is subordinate to a data item that is described with an OCCURS clause (that is, *identifier-1* can be a subordinate table within a multidimensional table). In this case, the data description entries must specify an INDEXED BY phrase for each dimension of the table.

identifier-1 must not be subscripted or reference-modified.

AT END

The condition that exists when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

Before executing a serial search, you must set the value of the first (or only) index associated with *identifier-1* (the search index) to indicate the starting occurrence for the search.

Before using a serial search on a multidimensional table, you must also set the value of the index for each superordinate dimension.

The SEARCH statement modifies only the value in the search index, and, if the VARYING phrase is specified, the value in *index-name-1* or *identifier-2*. Therefore, to search an entire two-dimensional to seven-dimensional table, you must execute a SEARCH statement for each dimension. In the WHEN phrases, you must specify the indexes for all dimensions. Before the execution of each SEARCH statement, you must initialize the associated indexes with SET statements.

The SEARCH statement executes a serial search beginning at the current setting of the search index.

When the search begins, if the value of the index associated with *identifier-1* is not greater than the highest possible occurrence number, the following actions take place:

- The conditions in the WHEN phrase are evaluated in the order in which they are written.
- If none of the conditions is satisfied, the index for *identifier-1* is increased to correspond to the next table element, and step 1 is repeated.
- If upon evaluation one of the WHEN conditions is satisfied, the search is terminated immediately, and the *imperative-statement-2* associated with that condition is executed. The index points to the table element that satisfied the condition. If NEXT SENTENCE is specified, control passes to the statement following the closest period.

- If the end of the table is reached (that is, the value of the incremented index is greater than the highest possible occurrence number) without the WHEN condition being satisfied, the search is terminated.

If, when the search begins, the value of the index-name associated with *identifier-1* is greater than the highest possible occurrence number, the search terminates immediately.

When the search terminates, if the AT END phrase is specified, *imperative-statement-1* is executed. If the AT END phrase is omitted, control passes to the next statement after the SEARCH statement.

Example: multidimensional serial search

The following code fragment shows a search of the inner dimension (table C) in the third occurrence within the superordinate table (table R):

```

. . .
Working-storage section.
1 G.
  2 R occurs 10 indexed by Rindex.
    3 C occurs 10 ascending key X indexed by Cindex.
      4 X pic 99.
1 Arg pic 99 value 34.
Procedure division.
. . .
* To search within occurrence 3 of table R, set its index to 3
* To search table C beginning at occurrence 1, set its index to 1
  Set Rindex to 3
  Set Cindex to 1
* In the SEARCH statement, specify C without indexes
  Search C
* Specify indexes for both dimensions in the WHEN phrase
  when X(Rindex Cindex) = Arg
    display "Found " X(Rindex Cindex)
  End-search
. . .

```

VARYING phrase

index-name-1

One of the following actions applies:

- If *index-name-1* is an index for *identifier-1*, this index is used for the search. Otherwise, the first (or only) index-name is used.
- If *index-name-1* is an index for another table element, then the first (or only) index-name for *identifier-1* is used for the search; the occurrence number represented by *index-name-1* is increased by the same amount as the search index-name and at the same time.

When the VARYING *index-name-1* phrase is omitted, the first (or only) index-name for *identifier-1* is used for the search.

If indexing is used to search a table without an INDEXED BY phrase, correct results are ensured only if both the table defined with the index and the table defined without the index have table elements of the same length and with the same number of occurrences.

When the object of the VARYING phrase is an index-name for another table element, one serial SEARCH statement steps through two table elements at once.

identifier-2

Must be either an index data item or an elementary integer item. *identifier-2* cannot be a windowed date field. *identifier-2* cannot be subscripted by the first (or only) index-name specified for *identifier-1*. During the search, one of the following actions applies:

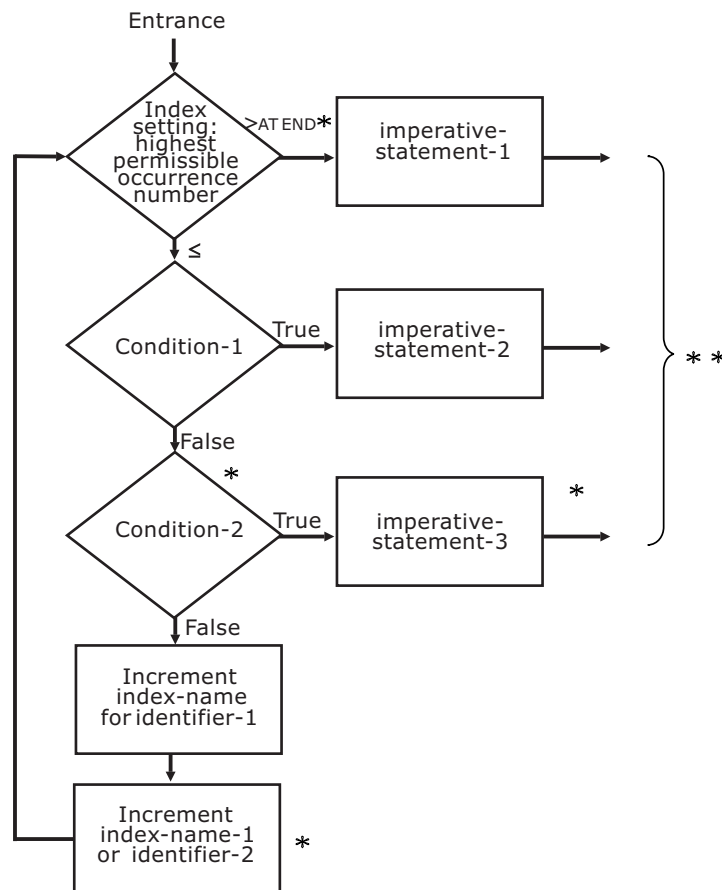
- If *identifier-2* is an index data item, then, whenever the search index is increased, the specified index data item is simultaneously increased by the same amount.
- If *identifier-2* is an integer data item, then, whenever the search index is increased, the specified data item is simultaneously increased by 1.

WHEN phrase (serial search)

condition-1

Can be any condition described under “Conditional expressions” on page 262.

The following figure illustrates a format-1 SEARCH operation containing two WHEN phrases.



- *These operations are included only when called for in the statement.
- * *Control transfers to the next sentence, unless the imperative statement ends with a GO TO statement.

Binary search

identifier-1 (binary search)

identifier-1 identifies the table that is to be searched. *identifier-1* references all occurrences within that table.

The data description entry for *identifier-1* must contain an OCCURS clause with the INDEXED BY and KEY IS phrases.

identifier-1 can reference a data item that is subordinate to a data item that contains an OCCURS clause (that is, *identifier-1* can be a subordinate table within a multidimensional table). In this case, the data description entry must specify an INDEXED BY phrase for each dimension of the table.

identifier-1 must not be subscripted or reference-modified.

AT END

The condition that exists when the search operation terminates without satisfying the conditions specified in the WHEN phrase.

The SEARCH ALL statement executes a binary search. The index associated with *identifier-1* (the search index) need not be initialized by SET statements. The search index is varied during the search operation so that its value is at no time less than the value of the first table element, nor ever greater than the value of the last table element. The index used is always that associated with the first index-name specified in the OCCURS clause.

Before using a binary search on a multidimensional table, you must execute SET statements to set the value of the index for each superordinate dimension.

The SEARCH statement modifies only the value in the search index. Therefore, to search an entire two-dimensional to seven-dimensional table, you must execute a SEARCH statement for each dimension. In the WHEN phrases, you must specify the indexes for all dimensions.

If the search ends without the WHEN condition being satisfied and the AT END phrase is specified, *imperative-statement-1* is executed. If the AT END phrase is omitted, control passes to the next statement after the SEARCH statement.

The results of a SEARCH ALL operation are predictable only when:

- The data in the table is ordered in ASCENDING KEY or DESCENDING KEY order
- The contents of the ASCENDING or DESCENDING keys specified in the WHEN clause provide a unique table reference.

WHEN phrase (binary search)

If a relation condition is specified in the WHEN phrase, the evaluation of the relation is based on the USAGE of the data item referenced by *data-name-1*. The search argument is moved to a temporary data item with the same USAGE as *data-name-1*, and this temporary data item is used for the compare operations associated with the SEARCH. If *data-name-1* is a numeric item, the temporary data item is signed or unsigned consistent with the presence or absence of a sign in the data description of *data-name-1*. If the search argument is signed and *data-name-1* is unsigned, the sign will be removed from the search argument before the comparison is done.

If the WHEN phrase cannot be satisfied for any setting of the index within this range, the search is unsuccessful. Control is passed to *imperative-statement-1* of the AT END phrase, when specified, or to the next statement after the SEARCH statement. In either case, the final setting of the index is not predictable.

If the WHEN phrase can be satisfied, control passes to *imperative-statement-2*, if specified, or to the next executable sentence if the NEXT SENTENCE phrase is specified. The index contains the value indicating the occurrence that allowed the WHEN conditions to be satisfied.

After *imperative-statement-2* is executed, control passes to the end of the SEARCH statement, unless *imperative-statement-2* ends with a GO TO statement.

condition-name-1, condition-name-2

Each condition-name specified must have only a single value, and each must be associated with an ASCENDING KEY or DESCENDING KEY data item for this table element.

data-name-1, data-name-2

Must specify an ASCENDING KEY or DESCENDING KEY data item in the table element referenced by *identifier-1* and must be subscripted by the first index-name associated with *identifier-1*. Each data-name can be qualified.

data-name-1 must be a valid operand for comparison with *identifier-3*, *literal-1*, or *arithmetic-expression-1* according to the rules of comparison.

data-name-2 must be a valid operand for comparison with *identifier-4*, *literal-2*, or *arithmetic-expression-2* according to the rules of comparison.

data-name-1 and *data-name-2* cannot reference:

- Floating-point data items
- Group items containing variable-occurrence data items
- Windowed date fields

identifier-3, identifier-4

Must not be an ASCENDING KEY or DESCENDING KEY data item for *identifier-1* or an item that is subscripted by the first index-name for *identifier-1*.

identifier-3 and *identifier-4* cannot be data items defined with any of the usages POINTER, FUNCTION-POINTER, PROCEDURE-POINTER, or OBJECT REFERENCE.

identifier-3 and *identifier-4* cannot be windowed date fields.

If *identifier-3* or *literal-1* is of class national, *data-name-1* must be of class national.

If *identifier-4* or *literal-2* is of class national, *data-name-2* must be of class national.

literal-1, literal-2

literal-1 or *literal-2* must be a valid operand for comparison with *data-name-1* or *data-name-2*, respectively.

arithmetic-expression

Can be any of the expressions defined under “Arithmetic expressions” on page 257, with the following restriction: Any identifier in *arithmetic-expression* must not be an ASCENDING KEY or DESCENDING KEY data item for *identifier-1* or an item that is subscripted by the first index-name for *identifier-1*.

When an ASCENDING KEY or DESCENDING KEY data item is specified, explicitly or implicitly, in the WHEN phrase, all preceding ASCENDING KEY or DESCENDING KEY data-names for *identifier-1* must also be specified.

Search statement considerations

Index data items cannot be used as subscripts, because of the restrictions on direct reference to them.

To ensure correct execution of a SEARCH statement for a variable-length table, make sure the object of the OCCURS DEPENDING ON clause (*data-name-1*) contains a value that specifies the current length of the table.

The scope of a SEARCH statement can be terminated by any of the following:

- An END-SEARCH phrase at the same level of nesting
- A separator period
- An ELSE or END-IF phrase associated with a previous IF statement

AT END and WHEN phrases

After *imperative-statement-1* or *imperative-statement-2* is executed, control passes to the end of the SEARCH statement, unless *imperative-statement-1* or *imperative-statement-2* ends with a GO TO statement.

The function of the AT END phrase is the same for a serial search and a binary search.

NEXT SENTENCE

NEXT SENTENCE transfers control to the first statement following the closest separator period.

When NEXT SENTENCE is specified with END-SEARCH, control does not pass to the statement following the END-SEARCH. Instead, control passes to the statement after the closest following period.

For the format-2 SEARCH ALL statement, neither *imperative-statement-2* nor NEXT SENTENCE is required. Without them, the SEARCH statement sets the index to the value in the table that matched the condition.

The function of the NEXT SENTENCE phrase is the same for a serial search and a binary search.

END-SEARCH phrase

This explicit scope terminator delimits the scope of the SEARCH statement. END-SEARCH permits a conditional SEARCH statement to be nested in another conditional statement.

For more information, see “Delimited scope statements” on page 289.

The function of END-SEARCH is the same for a serial search and a binary search.

SET statement

The SET statement is used to perform one of the following operations:

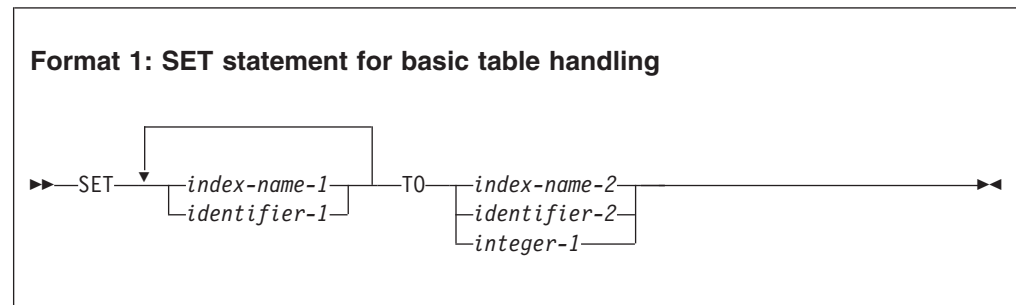
- Placing values associated with table elements into indexes associated with index-names
- Incrementing or decrementing an occurrence number
- Setting the status of an external switch to ON or OFF
- Moving data to condition names to make conditions true
- Setting USAGE POINTER data items to a data address
- Setting USAGE PROCEDURE-POINTER data items to an entry address
- Setting USAGE FUNCTION-POINTER data items to an entry address
- Setting USAGE OBJECT REFERENCE data items to refer to an object instance

Index-names are related to a given table through the INDEXED BY phrase of the OCCURS clause; they are not further defined in the program.

When the sending and receiving fields in a SET statement share part of their storage (that is, the operands overlap), the result of the execution of that SET statement is undefined.

Format 1: SET for basic table handling

When this form of the SET statement is executed, the current value of the receiving field is replaced by the value of the sending field (with conversion).



index-name-1

Receiving field.

Must name an index that is specified in the INDEXED BY phrase of an OCCURS clause.

identifier-1

Receiving field.

Must name either an index data item or an elementary numeric integer item. A receiving field cannot be a windowed date field.

index-name-2

Sending field.

Must name an index that is specified in the INDEXED BY phrase of an OCCURS clause. The value of the index before the SET statement is executed must correspond to an occurrence number of its associated table.

identifier-2

Sending field.

Must name either an index data item or an elementary numeric integer item. A sending field cannot be a windowed date field.

integer-1

Sending field.

Must be a positive integer.

The following table shows valid combinations of sending and receiving fields in a format-1 SET statement.

Table 55. Sending and receiving fields for format-1 SET statement

Sending field	Index-name receiving field	Index data item receiving field	Integer data item receiving field
Index-name*	Valid	Valid**	Valid
Index data item*	Valid**	Valid**	Invalid
Integer data item	Valid	Invalid	Invalid
Integer literal	Valid	Invalid	Invalid
*An index-name refers to an index named in the INDEXED BY phrase of an OCCURS clause. An index data item is defined with the USAGE IS INDEX clause.			
**No conversion takes place.			

Receiving fields are acted upon in the left-to-right order in which they are specified. Any subscripting or indexing associated with *identifier-1* is evaluated immediately before that receiving field is acted upon.

The value used for the sending field is the value at the beginning of SET statement execution.

The value of an index after execution of a SEARCH or PERFORM statement can be undefined; therefore, use a format-1 SET statement to reinitialize such indexes before you attempt other table-handling operations.

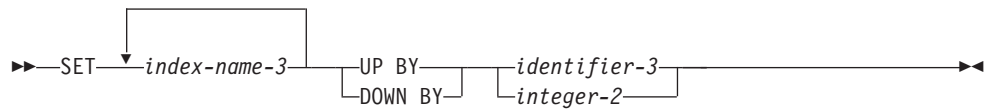
If *index-name-2* is for a table that has a subordinate item that contains an OCCURS DEPENDING ON clause, then undefined values can be received into *identifier-1*.

For more information about complex OCCURS DEPENDING ON, see *Appendix B. Complex OCCURS DEPENDING ON* in the *COBOL for AIX Programming Guide*.

Format 2: SET for adjusting indexes

When this form of the SET statement is executed, the value of the receiving index is increased (UP BY) or decreased (DOWN BY) by a value that corresponds to the value in the sending field.

Format 2: SET statement for adjusting indexes



The *receiving field* is an index specified by *index-name-3*. The index value both before and after the SET statement execution must correspond to an occurrence number in an associated table.

The *sending field* can be specified as *identifier-3*, which must be an elementary integer data item, or as *integer-2*, which must be a nonzero integer. *identifier-3* cannot be a windowed date field.

When the format-2 SET statement is executed, the contents of the receiving field are increased (UP BY) or decreased (DOWN BY) by a value that corresponds to the number of occurrences represented by the value of *identifier-3* or *integer-2*. Receiving fields are acted upon in the left-to-right order in which they are specified. The value of the incrementing or decrementing field at the beginning of SET statement execution is used for all receiving fields.

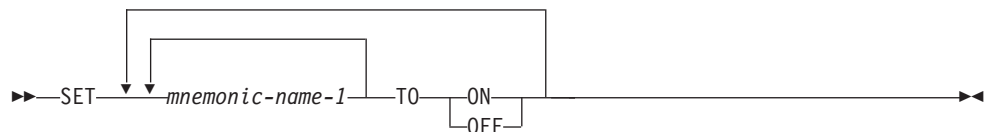
If *index-name-3* is for a table that has a subordinate item that contains an OCCURS DEPENDING ON clause, and if the ODO object is changed before executing a format-2 SET Statement, then *index-name-3* cannot contain a value that corresponds to an occurrence number of its associated table.

For more information about complex OCCURS DEPENDING ON, see *Appendix B. Complex OCCURS DEPENDING ON* in the *COBOL for AIX Programming Guide*.

Format 3: SET for external switches

When this form of the SET statement is executed, the status of each external switch associated with the specified mnemonic-name is turned ON or OFF.

Format 3: SET statement for external switches



mnemonic-name-1

Must be associated with an external switch, the status of which can be altered.

Format 4: SET for condition-names

When this form of the SET statement is executed, the value associated with a condition-name is placed in its conditional variable according to the rules of the VALUE clause.

Format 4: SET statement for condition-names

Diagram illustrating the syntax for Format 4: SET statement for condition-names. The statement is: `SET condition-name-1 TO TRUE`. The diagram shows the statement enclosed in a box with arrows at both ends. A bracket above the statement indicates that `condition-name-1` is a single identifier.

condition-name-1

Must be associated with a conditional variable.

If more than one literal is specified in the VALUE clause of *condition-name-1*, its associated conditional variable is set equal to the first literal.

If multiple condition-names are specified, the results are the same as if a separate SET statement had been written for each condition-name in the same order in which they are specified in the SET statement.

Format 5: SET for USAGE IS POINTER data items

When this form of the SET statement is executed, the current value of the receiving field is replaced by the address value contained in the sending field.

Format 5: SET statement for data-pointers

Diagram illustrating the syntax for Format 5: SET statement for data-pointers. The statement is: `SET identifier-4 ADDRESS OF identifier-5 TO identifier-6 ADDRESS OF identifier-7 NULL NULLS`. The diagram shows the statement enclosed in a box with arrows at both ends. Brackets above the statement indicate that `identifier-4` and `identifier-5` are single identifiers, while `identifier-6`, `identifier-7`, `NULL`, and `NULLS` are part of a list of identifiers.

identifier-4

Receiving fields.

Must be described as USAGE IS POINTER.

ADDRESS OF *identifier-5*

Receiving fields.

identifier-5 must be level-01 or level-77 items defined in the linkage section. The addresses of these items are set to the value of the operand specified in the TO phrase.

identifier-5 must not be reference-modified.

identifier-6

Sending field.

Must be described as USAGE IS POINTER.

Cannot contain an address within the program's own working-storage section, file section, or local-storage section.

ADDRESS OF *identifier-7*

Sending field. *identifier-7* must name an item of any level except 66 or 88 in the linkage section, the working-storage section, or the local-storage section. ADDRESS OF *identifier-7* contains the address of the identifier, and not the content of the identifier.

NULL, NULLS

Sending field.

Sets the receiving field to contain the value of an invalid address.

The following table shows valid combinations of sending and receiving fields in a format-5 SET statement.

Table 56. Sending and receiving fields for format-5 SET statement

Sending field	USAGE IS POINTER receiving field	ADDRESS OF receiving field	NULL/NULLS receiving field
USAGE IS POINTER	Valid	Valid	Invalid
ADDRESS OF	Valid	Valid	Invalid
NULL/NULLS	Valid	Valid	Invalid

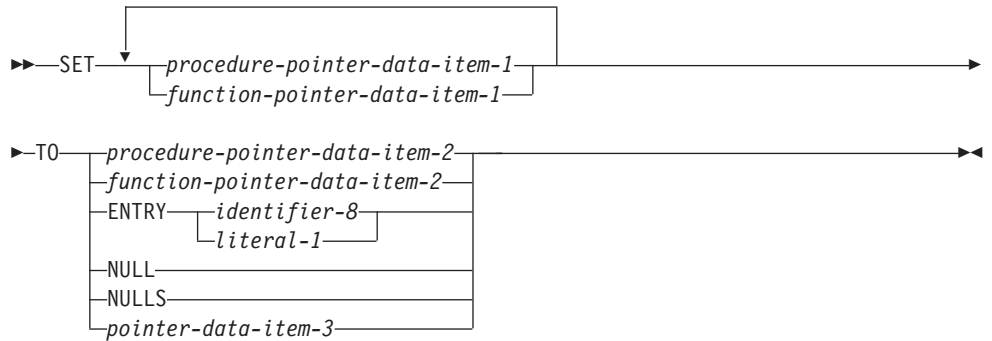
Format 6: SET for procedure-pointer and function-pointer data items

When this format of the SET statement is executed, the current value of the receiving field is replaced by the address value specified by the sending field.

At run time, function-pointers and procedure-pointers can reference the address of the primary entry point of a COBOL program, an alternate entry point in a COBOL program, or an entry point in a non-COBOL program; or they can be NULL.

COBOL function-pointers are more easily used than procedure-pointers for interoperation with C functions.

Format 6: SET statement for procedure-pointers and function-pointers



procedure-pointer-data-item-1, procedure-pointer-data-item-2

Must be described as USAGE IS PROCEDURE-POINTER.

procedure-pointer-data-item-1 is a receiving field; *procedure-pointer-data-item-2* is a sending field.

function-pointer-data-item-1, function-pointer-data-item-2

Must be described as USAGE IS FUNCTION-POINTER.

function-pointer-data-item-1 is a receiving field; *function-pointer-data-item-2* is a sending field.

identifier-8

Must be defined as an alphabetic or alphanumeric item such that the value can be a program name. For more information, see “PROGRAM-ID paragraph” on page 98. For entry points in non-COBOL programs, *identifier-8* can contain the characters @, #, and, \$.

literal-1

Must be alphanumeric and must conform to the rules for formation of program-names. For details on formation rules, see the discussion of program-name under “PROGRAM-ID paragraph” on page 98.

identifier-8 or *literal-1* must refer to one of the following types of entry points:

- The primary entry point of a COBOL program as defined by the PROGRAM-ID paragraph. The PROGRAM-ID must reference the outermost program of a compilation unit; it must not reference a nested program.
- An alternate entry point of a COBOL program as defined by a COBOL ENTRY statement.
- An entry point in a non-COBOL program.

The program-name referenced by the SET ... TO ENTRY statement can be affected by the PGMNAME compiler option. For details, see PGMNAME in the *COBOL for AIX Programming Guide*.

NULL, NULLS

Sets the receiving field to contain the value of an invalid address.

pointer-data-item-3

Must be defined with USAGE POINTER. You must set *pointer-data-item-3* in a non-COBOL program to point to a valid program entry point.

Example of COBOL/C interoperability

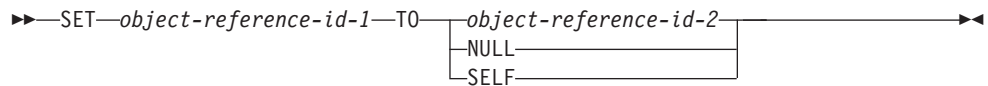
The following example demonstrates a COBOL CALL to a C function that returns a function-pointer to a service, followed by a COBOL CALL to the service:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID DEMO.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 FP USAGE FUNCTION-POINTER.  
PROCEDURE DIVISION.  
    CALL "c-function" RETURNING FP.  
    CALL FP.
```

Format 7: SET for USAGE OBJECT REFERENCE data items

When this format of the SET statement is executed the value in the receiving item is replaced by the value in the sending item.

Format 7: SET statement for object references



object-reference-id-1 and *object-reference-id-2* must be defined as USAGE OBJECT REFERENCE. *object-reference-id-1* is the receiving item and *object-reference-id-2* is the sending item. If *object-reference-id-1* is defined as an object reference of a certain class (defined as "USAGE OBJECT REFERENCE class-name"), *object-reference-id-2* must be an object reference of the same class or a class derived from that class.

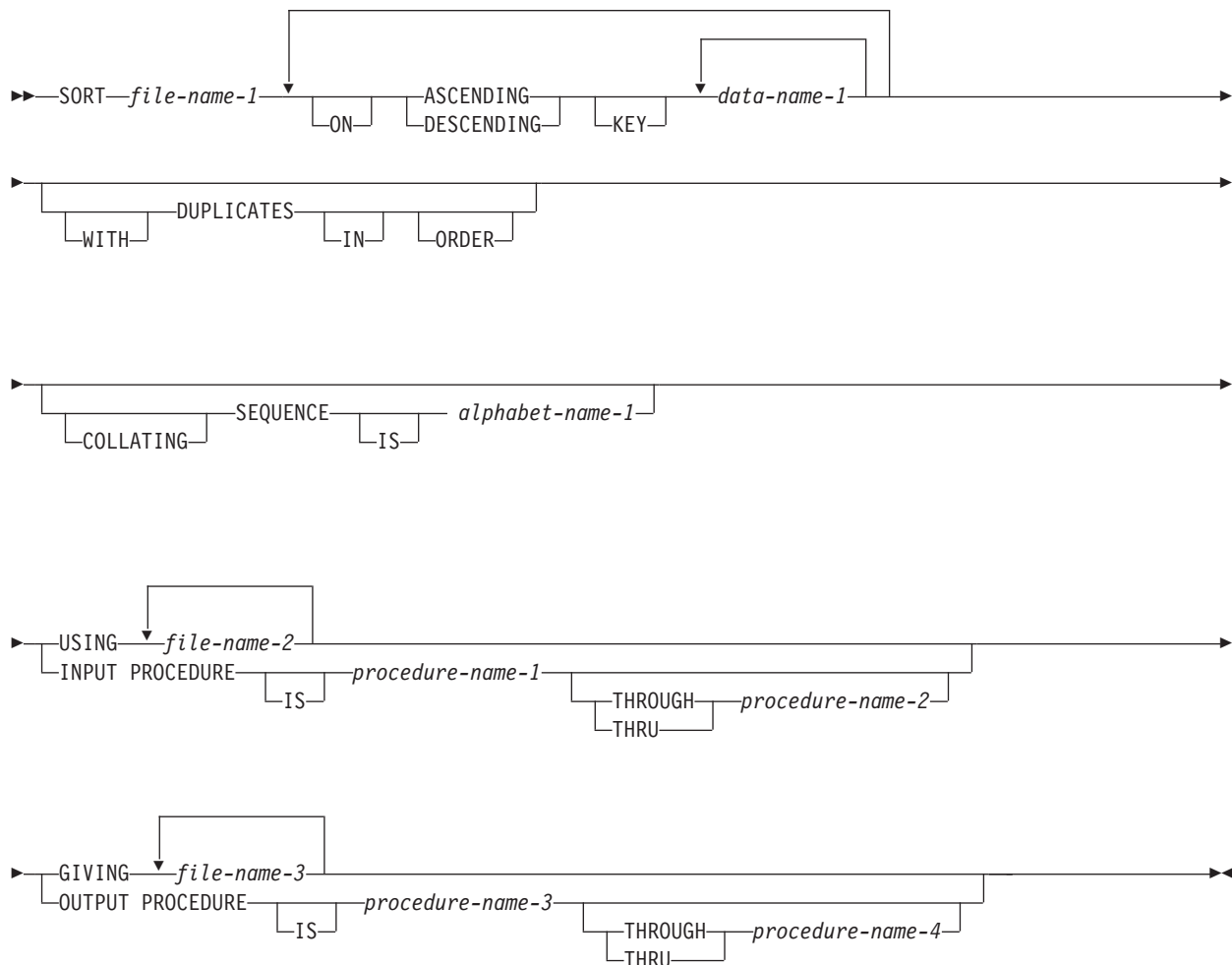
If the figurative constant NULL is specified, the receiving *object-reference-id-1* is set to the NULL value.

If SELF is specified, the SET statement must appear in the procedure division of a method. *object-reference-id-1* is set to reference the object upon which the currently executing method was invoked.

SORT statement

The SORT statement accepts records from one or more files, sorts them according to the specified keys, and makes the sorted records available either through an output procedure or in an output file. See also “MERGE statement” on page 374. The SORT statement can appear anywhere in the procedure division except in the declarative portion.

Format



file-name-1

The name given in the SD entry that describes the records to be sorted.

No pair of file-names in a SORT statement can be specified in the same SAME SORT AREA clause or the SAME SORT-MERGE AREA clause. File-names associated with the GIVING clause (*file-name-3*, ...) cannot be specified in the SAME AREA clause; however, they can be associated with the SAME RECORD AREA clause.

ASCENDING KEY and DESCENDING KEY phrases

This phrase specifies that records are to be processed in ascending or descending sequence (depending on the phrase specified), based on the specified sort keys.

data-name-1

Specifies a KEY data item on which the SORT statement will be based.

Each such data-name must identify a data item in a record associated with *file-name-1*. The data-names following the word KEY are listed from left to right in the SORT statement in order of decreasing significance without regard to how they are divided into KEY phrases. The leftmost data-name is the major key, the next data-name is the next most significant key, and so forth. The following rules apply:

- A specific KEY data item must be physically located in the same position and have the same data format in each input file. However, it need not have the same data-name.
- If *file-name-1* has more than one record description, the KEY data items need be described in only one of the record descriptions.
- If *file-name-1* contains variable-length records, all of the KEY data-items must be contained within the first *n* character positions of the record, where *n* equals the minimum records size specified for *file-name-1*.
- KEY data items must not contain an OCCURS clause or be subordinate to an item that contains an OCCURS clause.
- KEY data items cannot be:
 - Variably located
 - Group items that contain variable-occurrence data items
 - Windowed date fields
 - Category numeric described with usage NATIONAL (national decimal item)
 - Category external floating-point described with usage NATIONAL (national floating-point item)
 - Category DBCS
- KEY data items can be qualified.
- KEY data items can be any of the following data categories:
 - Alphabetic, alphanumeric, alphanumeric-edited
 - Numeric (except numeric with usage NATIONAL)
 - Numeric-edited (with usage DISPLAY or NATIONAL)
 - Internal floating-point or display floating-point
 - National or national-edited when the NCOLLSEQ(BINARY) compiler option is in effect. Binary collating sequence is applied to national keys.

If *file-name-3* references an indexed file, the first specification of *data-name-1* must be associated with an ASCENDING phrase and the data item referenced by that *data-name-1* must occupy the same character positions in this record as the data item associated with the major record key for that file.

The direction of the sorting operation depends on the specification of the ASCENDING or DESCENDING keywords as follows:

- When ASCENDING is specified, the sequence is from the lowest key value to the highest key value.

- When DESCENDING is specified, the sequence is from the highest key value to the lowest.
- If the KEY data item is alphabetic, alphanumeric, alphanumeric-edited, or numeric-edited, the sequence of key values depends on the collating sequence used (see “COLLATING SEQUENCE phrase”).
- If the KEY data item is described with usage NATIONAL, the sequence of the KEY values is based on the binary values of the national characters.
- If the KEY is a display floating-point item, the compiler treats the data item as character data of the same size as the key, rather than as numeric data. The sequence in which the records are sorted depends on the collating sequence used.
- If the KEY data item is internal floating point, the sequence of key values will be in numeric order.
- When the COLLATING SEQUENCE phrase is not specified, the key comparisons are performed according to the rules for comparison of operands in a relation condition. See “General relation conditions” on page 267).
- When the COLLATING SEQUENCE phrase is specified, the indicated collating sequence is used for key data items of alphabetic, alphanumeric, alphanumeric-edited, external floating-point, and numeric-edited categories. For all other key data items, the comparisons are performed according to the rules for comparison of operands in a relation condition.

DUPLICATES phrase

If the DUPLICATES phrase is specified, and the contents of all the key elements associated with one record are equal to the corresponding key elements in one or more other records, the order of return of these records is as follows:

- The order of the associated input files as specified in the SORT statement. Within a given file the order is that in which the records are accessed from that file.
- The order in which these records are released by an input procedure, when an input procedure is specified.

If the DUPLICATES phrase is not specified, the order of these records is undefined.

COLLATING SEQUENCE phrase

This phrase specifies the collating sequence to be used in alphanumeric comparisons for the KEY data items in this sorting operation.

The COLLATING SEQUENCE phrase has no effect for keys that are not alphabetic or alphanumeric.

The COLLATING SEQUENCE phrase is valid only when a single-byte ASCII code page is in effect.

alphabet-name-1

Must be specified in the ALPHABET clause of the SPECIAL-NAMES paragraph. *alphabet-name-1* can be associated with any one of the ALPHABET clause phrases, with the following results:

STANDARD-1

The collating sequence is based on the character's hex value order.

STANDARD-2

The collating sequence is based on the character's hex value order.

NATIVE

The collating sequence indicated by the locale is selected.

EBCDIC

The EBCDIC collating sequence is used for all alphanumeric comparisons. (The EBCDIC collating sequence is shown in Appendix D, "EBCDIC and ASCII collating sequences," on page 585.)

literal The collating sequence established by the specification of literals in the alphabet-name clause is used for all alphanumeric comparisons.

When the COLLATING SEQUENCE phrase is omitted, the PROGRAM COLLATING SEQUENCE clause (if specified) in the OBJECT-COMPUTER paragraph specifies the collating sequence to be used. When both the COLLATING SEQUENCE phrase and the PROGRAM COLLATING SEQUENCE clause are omitted, the collating sequence indicated by the COLLSEQ compiler option is used.

USING phrase

file-name-2, ...

The input files.

When the USING phrase is specified, all the records in *file-name-2, ...*, (that is, the input files) are transferred automatically to *file-name-1*. At the time the SORT statement is executed, these files must not be open. The compiler opens, reads, makes records available, and closes these files automatically. If EXCEPTION/ERROR procedures are specified for these files, the compiler makes the necessary linkage to these procedures.

All input files must be described in FD entries in the data division.

If the USING phrase is specified and if *file-name-1* contains variable-length records, the size of the records contained in the input files (*file-name-2, ...*) must be neither less than the smallest record nor greater than the largest record described for *file-name-1*. If *file-name-1* contains fixed-length records, the size of the records contained in the input files must not be greater than the largest record described for *file-name-1*. For more information, see *Describing the input to sorting or merging* in the *COBOL for AIX Programming Guide*.

INPUT PROCEDURE phrase

This phrase specifies the name of a procedure that is to select or modify input records before the sorting operation begins.

procedure-name-1

Specifies the first (or only) section or paragraph in the input procedure.

procedure-name-2

Identifies the last section or paragraph of the input procedure.

The input procedure can consist of any procedure needed to select, modify, or copy the records that are made available one at a time by the RELEASE statement to the file referenced by *file-name-1*. The range includes all

statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, PERFORM, and XML PARSE statements in the range of the input procedure, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the input procedure. The range of the input procedure must not cause the execution of any MERGE, RETURN, or SORT statement.

If an input procedure is specified, control is passed to the input procedure before the file referenced by *file-name-1* is sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last statement in the input procedure. When control passes the last statement in the input procedure, the records that have been released to the file referenced by *file-name-1* are sorted.

GIVING phrase

file-name-3, ...

The output files.

When the GIVING phrase is specified, all the sorted records in *file-name-1* are automatically transferred to the output files (*file-name-3, ...*).

All output files must be described in FD entries in the data division.

If the output files (*file-name-3, ...*) contain variable-length records, the size of the records contained in *file-name-1* must be neither less than the smallest record nor greater than the largest record described for the output files. If the output files contain fixed-length records, the size of the records contained in *file-name-1* must not be greater than the largest record described for the output files. For more information, see *Describing the output to sorting or merging* in the *COBOL for AIX Programming Guide*.

At the time the SORT statement is executed, the output files (*file-name-3, ...*) must not be open. For each of the output files, the execution of the SORT statement causes the following actions to be taken:

- The processing of the file is initiated. The initiation is performed as if an OPEN statement with the OUTPUT phrase had been executed.
- The sorted logical records are returned and written onto the file. Each record is written as if a WRITE statement without any optional phrases had been executed.

For a relative file, the relative key data item for the first record returned contains the value '1'; for the second record returned, the value '2'. After execution of the SORT statement, the content of the relative key data item indicates the last record returned to the file.

- The processing of the file is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

These implicit functions are performed such that any associated USE AFTER EXCEPTION/ERROR procedures are executed; however, the execution of such a USE procedure must not cause the execution of any statement manipulating the file referenced by, or accessing the record area associated with, *file-name-3*. On the first attempt to write beyond the externally defined boundaries of the file, any USE AFTER STANDARD EXCEPTION/ERROR procedure specified for the file is executed. If control is returned from that USE procedure or if no such USE procedure is specified, the processing of the file is terminated.

OUTPUT PROCEDURE phrase

This phrase specifies the name of a procedure that is to select or modify output records from the sorting operation.

procedure-name-3

Specifies the first (or only) section or paragraph in the output procedure.

procedure-name-4

Identifies the last section or paragraph of the output procedure.

The output procedure can consist of any procedure needed to select, modify, or copy the records that are made available one at a time by the RETURN statement in sorted order from the file referenced by *file-name-1*. The range includes all statements that are executed as the result of a transfer of control by CALL, EXIT, GO TO, PERFORM, and XML PARSE statements in the range of the output procedure. The range also includes all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure. The range of the output procedure must not cause the execution of any MERGE, RELEASE, or SORT statement.

If an output procedure is specified, control passes to it after the file referenced by *file-name-1* has been sequenced by the SORT statement. The compiler inserts a return mechanism at the end of the last statement in the output procedure and when control passes the last statement in the output procedure, the return mechanism provides the termination of the sort and then passes control to the next executable statement after the SORT statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record in sorted order when requested. The RETURN statements in the output procedure are the requests for the next record.

The INPUT PROCEDURE and OUTPUT PROCEDURE phrases are similar to those for a basic PERFORM statement. For example, if you name a procedure in an output procedure, that procedure is executed during the sorting operation just as if it were named in a PERFORM statement. As with the PERFORM statement, execution of the procedure is terminated after the last statement completes execution. The last statement in an input or output procedure can be the EXIT statement (see “EXIT statement” on page 345).

SORT special registers

The special registers, SORT-CORE-SIZE, SORT-MESSAGE, and SORT-MODE-SIZE, are equivalent to option control statement keywords in the sort control file. You define the sort control file with the SORT-CONTROL special register.

Usage note: If you use a sort control file to specify control statements, the values specified in the sort control file take precedence over those in the special register.

SORT-MESSAGE special register

See “SORT-MESSAGE” on page 23.

SORT-CORE-SIZE special register

See “SORT-CORE-SIZE” on page 22.

SORT-FILE-SIZE special register

See “SORT-FILE-SIZE” on page 23.

SORT-MODE-SIZE special register

See "SORT-MODE-SIZE" on page 23.

SORT-CONTROL special register

See "SORT-CONTROL" on page 22.

SORT-RETURN special register

See "SORT-RETURN" on page 24.

Segmentation considerations

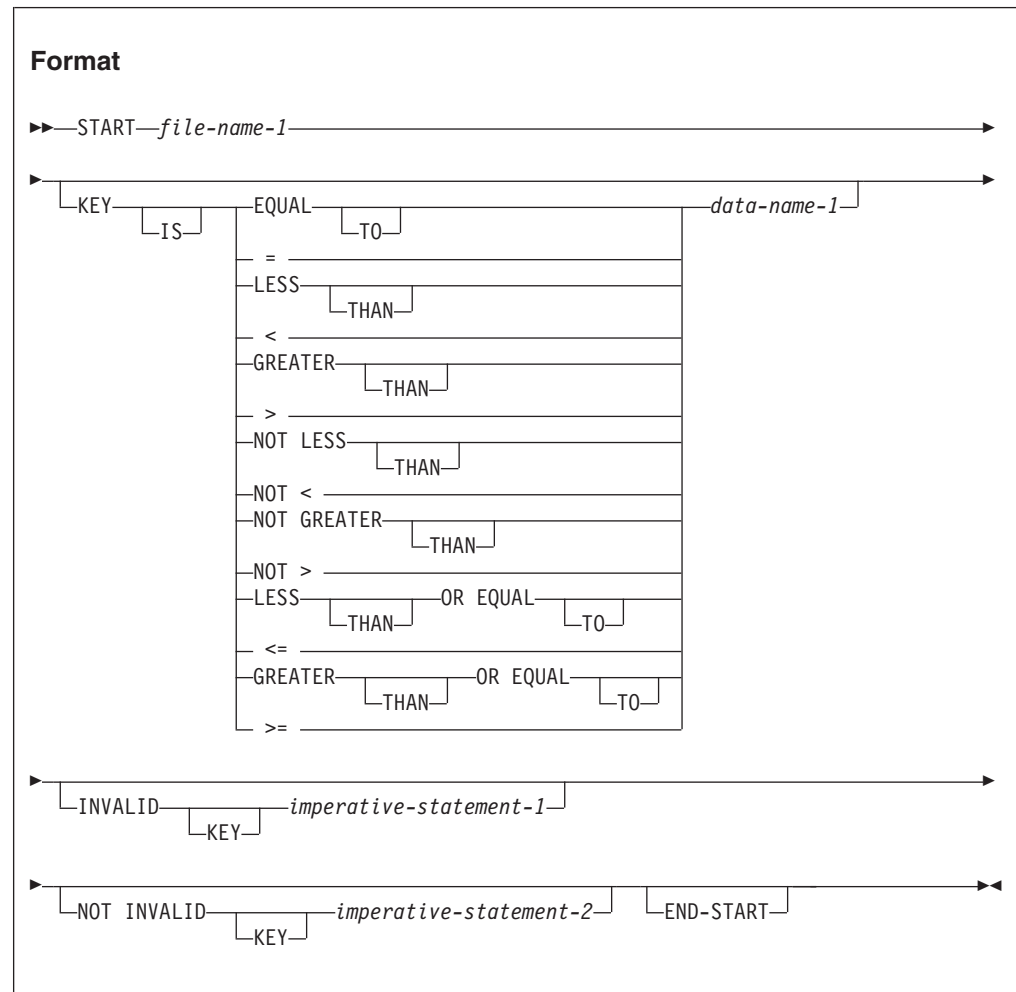
If a SORT statement is coded in a fixed segment, any input or output procedure referenced by that SORT statement must be either totally within a fixed segment or wholly contained in a single independent segment.

If a SORT statement is coded in an independent segment, any input or output procedure referenced by that SORT statement must be either totally within a fixed segment or wholly contained within the same independent segment as that SORT statement.

START statement

The START statement provides a means of positioning within an indexed or relative file for subsequent sequential record retrieval.

When the START statement is executed, the associated indexed or relative file must be open in either INPUT or I-O mode.



file-name-1

Must name a file with sequential or dynamic access. *file-name-1* must be defined in an FD entry in the data division and must not name a sort file.

KEY phrase

When the KEY phrase is specified, the file position indicator is positioned at the logical record in the file whose key field satisfies the comparison.

When the KEY phrase is not specified, KEY IS EQUAL (to the prime record key) is implied.

If you specify the KEY to be 'less than' or 'less than or equal to' the data item, the file position indicator is positioned to the last logical record that currently exists in the file that satisfies the comparison.

For an indexed file, if the key that satisfies the comparison has duplicate entries, the file position indicator is positioned to the last of these entries.

data-name-1

Can be qualified; it cannot be subscripted.

When the START statement is executed, a comparison is made between the current value in the key *data-name* and the corresponding key field in the file's index.

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the START statement is executed (See "File status key" on page 297).

INVALID KEY phrases

If the comparison is not satisfied by any record in the file, an invalid key condition exists; the position of the file position indicator is undefined, and (if specified) the INVALID KEY imperative-statement is executed. (See "INTO and FROM phrases" on page 301 under "Common processing facilities".)

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

END-START phrase

This explicit scope terminator serves to delimit the scope of the START statement. END-START permits a conditional START statement to be nested in another conditional statement. END-START can also be used with an imperative START statement.

For more information, see "Delimited scope statements" on page 289.

Indexed files

When the KEY phrase is specified, the key data item used for the comparison is *data-name-1*.

When the KEY phrase is not specified, the key data item used for the EQUAL TO comparison is the prime RECORD KEY.

When START statement execution is successful, the RECORD KEY or ALTERNATE RECORD KEY with which *data-name-1* is associated becomes the key of reference for subsequent READ statements.

data-name-1

Can be any of the following:

- The prime RECORD KEY.
- Any ALTERNATE RECORD KEY.
- A data item within a record description for a file whose leftmost character position corresponds to the leftmost character position of that

record key; it can be qualified. The size of the data item must be less than or equal to the length of the record key for the file.

Regardless of its category, *data-name-1* is treated as an alphanumeric item for purposes of the comparison operation.

The file position indicator points to the first record in the file whose key field satisfies the comparison. If the operands in the comparison are of unequal lengths, the comparison proceeds as if the longer field were truncated on the right to the length of the shorter field. All other numeric and alphanumeric comparison rules apply, except that the PROGRAM COLLATING SEQUENCE clause, if specified, has no effect.

When START statement execution is successful, the RECORD KEY with which *data-name-1* is associated becomes the key of reference for subsequent READ statements.

When START statement execution is unsuccessful, the key of reference is undefined.

Relative files

When the KEY phrase is specified, *data-name-1* must specify the RELATIVE KEY.

Whether or not the KEY phrase is specified, the key data item used in the comparison is the RELATIVE KEY data item. Numeric comparison rules apply.

The file position indicator points to the logical record in the file whose key satisfies the specified comparison.

STOP statement

The STOP statement halts execution of the object program either permanently or temporarily.

Format

►► STOP RUN
 └ *literal* ─┘

literal

Can be a fixed-point numeric literal (signed or unsigned) or an alphanumeric literal. It can be any figurative constant except ALL *literal*.

When STOP *literal* is specified, the literal is communicated to the operator, and object program execution is suspended. Program execution is resumed only after operator intervention, and continues at the next executable statement in sequence.

The STOP *literal* statement is useful for special situations when operator intervention is needed during program execution; for example, when a special tape or disk must be mounted or a specific daily code must be entered. However, the ACCEPT and DISPLAY statements are preferred when operator intervention is needed.

Do not use STOP RUN or STOP *literal* in programs compiled with the THREAD compiler option.

When STOP RUN is specified, execution is terminated and control is returned to the system. When STOP RUN is not the last or only statement in a sequence of imperative statements within a sentence, the statements following STOP RUN are not executed.

The STOP RUN statement closes *all* files defined in any of the programs in the run unit.

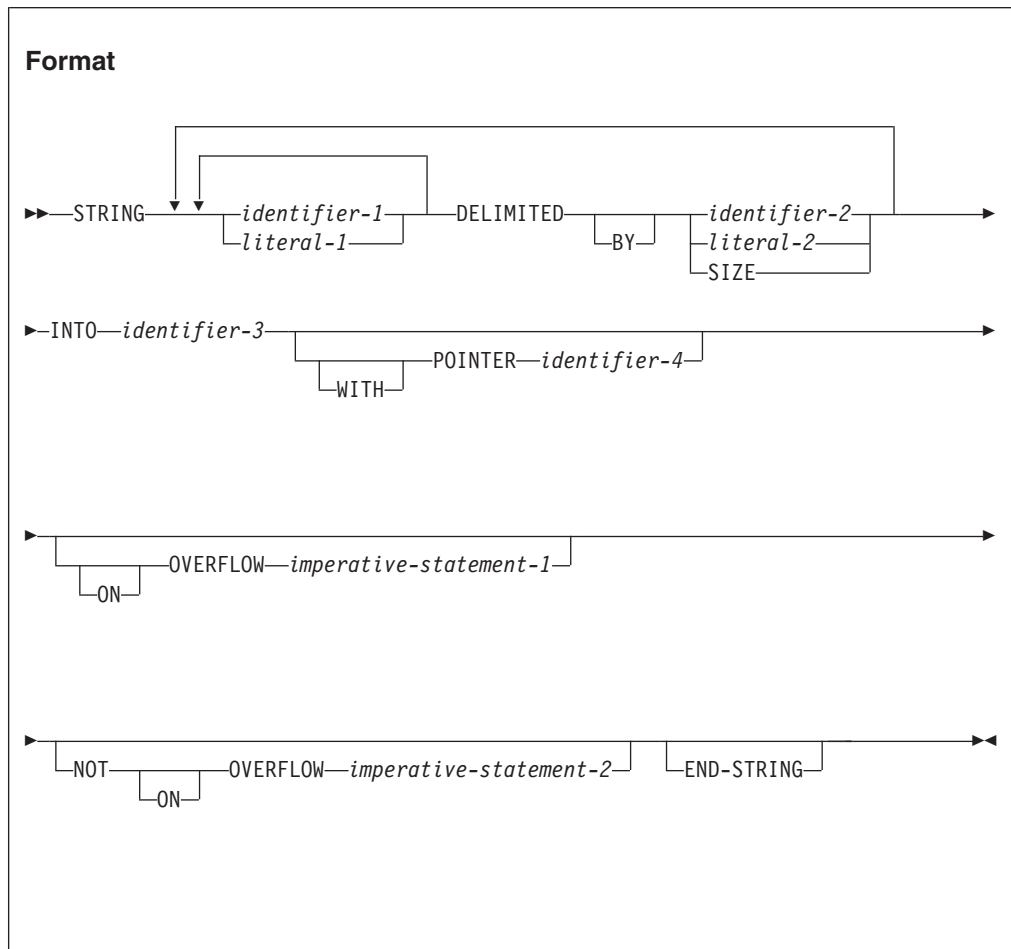
For use of the STOP RUN statement in calling and called programs, see the following table.

Termination statement	Main program	Subprogram
STOP RUN	Returns to the calling program. (Can be the system, which causes the application to end.)	Returns directly to the program that called the main program. (Can be the system, which causes the application to end.)

STRING statement

The STRING statement strings together the partial or complete contents of two or more data items or literals into one single data item.

One STRING statement can be written instead of a series of MOVE statements.



identifier-1, literal-1

Represents the *sending fields*.

DELIMITED BY phrase

Sets the limits of the string.

identifier-2, literal-2

Are delimiters; that is, characters that delimit the data to be transferred.

SIZE Transfers the complete sending area.

INTO phrase

Identifies the receiving field.

identifier-3

Represents the *receiving field*.

POINTER phrase

Points to a character position in the receiving field. The pointer field indicates a relative alphanumeric character position, DBCS character position, or national character position when the receiving field is of usage DISPLAY, DISPLAY-1, or NATIONAL, respectively.

identifier-4

Represents the *pointer field*. *identifier-4* must be large enough to contain a value equal to the length of the receiving field plus 1. You must initialize *identifier-4* to a nonzero value before execution of the STRING statement begins.

The following rules apply:

- All identifiers except *identifier-4* must reference data items described explicitly or implicitly as usage DISPLAY, DISPLAY-1, or NATIONAL.
- *literal-1* or *literal-2* must be of category alphanumeric, DBCS, or national and can be any figurative constant that does not begin with the word ALL (except NULL).
- If *identifier-1* or *identifier-2* references a data item of category numeric, each numeric item must be described as an integer without the symbol 'P' in its PICTURE character-string.
- *identifier-3* must not reference a data item of category numeric-edited, alphanumeric-edited, or national-edited; an external floating-point data item of usage DISPLAY, or an external floating-point data item of usage NATIONAL.
- *identifier-3* must not be described with the JUSTIFIED clause.
- If *identifier-3* is of usage DISPLAY, *identifier-1* and *identifier-2* must be of usage DISPLAY and all literals must be alphanumeric literals. Any figurative constant can be specified except one that begins with the word ALL. Each figurative constant represents a 1-character alphanumeric literal.
- If *identifier-3* is of usage DISPLAY-1, *identifier-1* and *identifier-2* must be of usage DISPLAY-1 and all literals must be DBCS literals. The only figurative constant that can be specified is SPACE, which represents a 1-character DBCS literal. ALL DBCS-literal must not be specified.
- If *identifier-3* is of usage NATIONAL, *identifier-1* and *identifier-2* must be of usage NATIONAL and all literals must be national literals. Any figurative constant can be specified except *symbolic-character* and one that begins with the word ALL. Each figurative constant represents a 1-character national literal.
- If *identifier-1* or *identifier-2* references an elementary data item of usage DISPLAY that is described as category numeric, numeric-edited, or alphanumeric-edited, the item is treated as if it were redefined as category alphanumeric.
- If *identifier-1* or *identifier-2* references an elementary data item of usage NATIONAL that is described as category numeric, numeric-edited, or national-edited item, the item is treated as if it were redefined as category national.
- *identifier-4* must not be described with the symbol P in its PICTURE character-string.
- None of the identifiers in a STRING statement can be a windowed date field.

Evaluation of subscripts, reference modification, variable-lengths, variable locations, and function-identifiers is performed only once, at the beginning of the execution of the STRING statement. Therefore, if *identifier-3* or *identifier-4* is used as a subscript, reference-modifier, or function argument in the STRING statement, or affects the length or location of any of the identifiers in the STRING statement, the

values calculated for those subscripts, reference-modifiers, variable lengths, variable locations, and functions are not affected by any results of the STRING statement.

If *identifier-3* and *identifier-4* occupy the same storage area, undefined results will occur, even if the identifiers are defined by the same data description entry.

If *identifier-1* or *identifier-2* occupies the same storage area as *identifier-3* or *identifier-4*, undefined results will occur, even if the identifiers are defined by the same data description entry.

See “Data flow” on page 448 below for details of STRING statement processing.

ON OVERFLOW phrases

imperative-statement-1

Executed when the pointer value (explicit or implicit):

- Is less than 1
- Exceeds a value equal to the length of the receiving field

When either of the above conditions occurs, an overflow condition exists, and no more data is transferred. Then the STRING operation is terminated, the NOT ON OVERFLOW phrase, if specified, is ignored, and control is transferred to the end of the STRING statement or, if the ON OVERFLOW phrase is specified, to *imperative-statement-1*.

If control is transferred to *imperative-statement-1*, execution continues according to the rules for each statement specified in *imperative-statement-1*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement; otherwise, upon completion of the execution of *imperative-statement-1*, control is transferred to the end of the STRING statement.

If at the time of execution of a STRING statement, conditions that would cause an overflow condition are not encountered, then after completion of the transfer of data, the ON OVERFLOW phrase, if specified, is ignored. Control is then transferred to the end of the STRING statement, or if the NOT ON OVERFLOW phrase is specified, to *imperative-statement-2*.

If control is transferred to *imperative-statement-2*, execution continues according to the rules for each statement specified in *imperative-statement-2*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of *imperative-statement-2*, control is transferred to the end of the STRING statement.

END-STRING phrase

This explicit scope terminator serves to delimit the scope of the STRING statement. END-STRING permits a conditional STRING statement to be nested in another conditional statement. END-STRING can also be used with an imperative STRING statement.

For more information, see “Delimited scope statements” on page 289.

Data flow

When the STRING statement is executed, characters are transferred from the sending fields to the receiving field. The order in which sending fields are processed is the order in which they are specified. The following rules apply:

- Characters from the sending fields are transferred to the receiving fields in the following manner:
 - For national sending fields, data is transferred using the rules of the MOVE statement for elementary national-to-national moves, except that no space filling takes place.
 - For DBCS sending fields, data is transferred using the rules of the MOVE statement for elementary DBCS-to-DBCS moves, except that no space filling takes place.
 - Otherwise, data is transferred to the receiving fields using the rules of the MOVE statement for elementary alphanumeric-to-alphanumeric moves, except that no space filling takes place (see “MOVE statement” on page 380).
- When DELIMITED BY *identifier-2* or *literal-2* is specified, the contents of each sending item are transferred, character-by-character, beginning with the leftmost character position and continuing until either:
 - A delimiter for this sending field is reached (the delimiter itself is not transferred).
 - The rightmost character of this sending field has been transferred.
- When DELIMITED BY SIZE is specified, each entire sending field is transferred to the receiving field.
- When the receiving field is filled, or when all the sending fields have been processed, the operation is ended.
- When the POINTER phrase is specified, an explicit pointer field is available to the COBOL user to control placement of data in the receiving field. The user must set the explicit pointer's initial value, which must not be less than 1 and not more than the character position count of the receiving field.

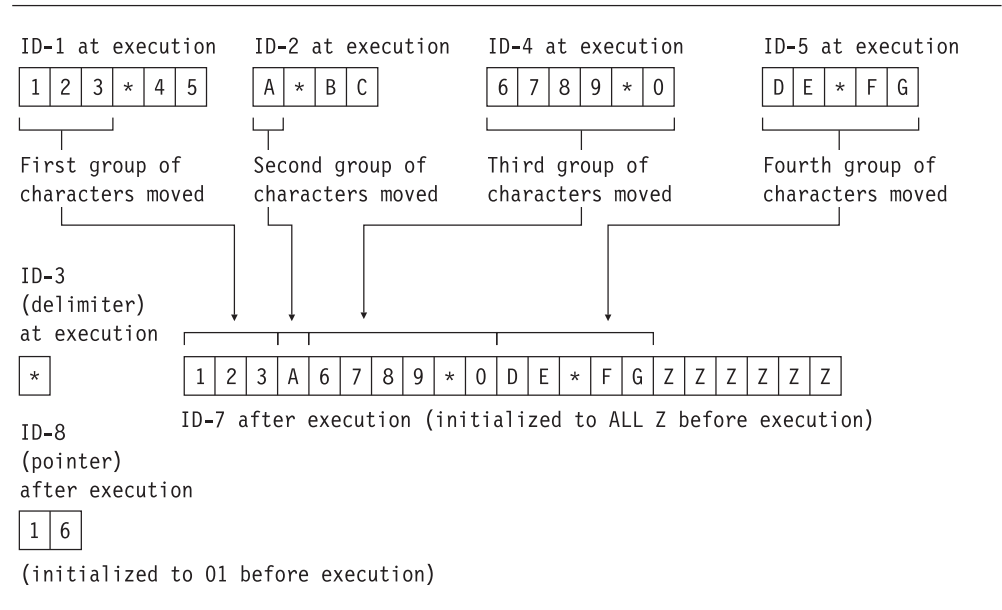
Usage note: The pointer field must be defined as a field large enough to contain a value equal to the length of the receiving field plus 1; this precludes arithmetic overflow when the system updates the pointer at the end of the transfer.

- When the POINTER phrase is not specified, no pointer is available to the user. However, a conceptual implicit pointer with an initial value of 1 is used by the system.
- Conceptually, when the STRING statement is executed, the initial pointer value (explicit or implicit) is the first character position within the receiving field into which data is to be transferred. Beginning at that position, data is then positioned, character-by-character, from left to right. After each character is positioned, the explicit or implicit pointer is increased by 1. The value in the pointer field is changed only in this manner. At the end of processing, the pointer value always indicates a value equal to one character position beyond the last character transferred into the receiving field.

After STRING statement execution is completed, only that part of the receiving field into which data was transferred is changed. The rest of the receiving field contains the data that was present before this execution of the STRING statement.

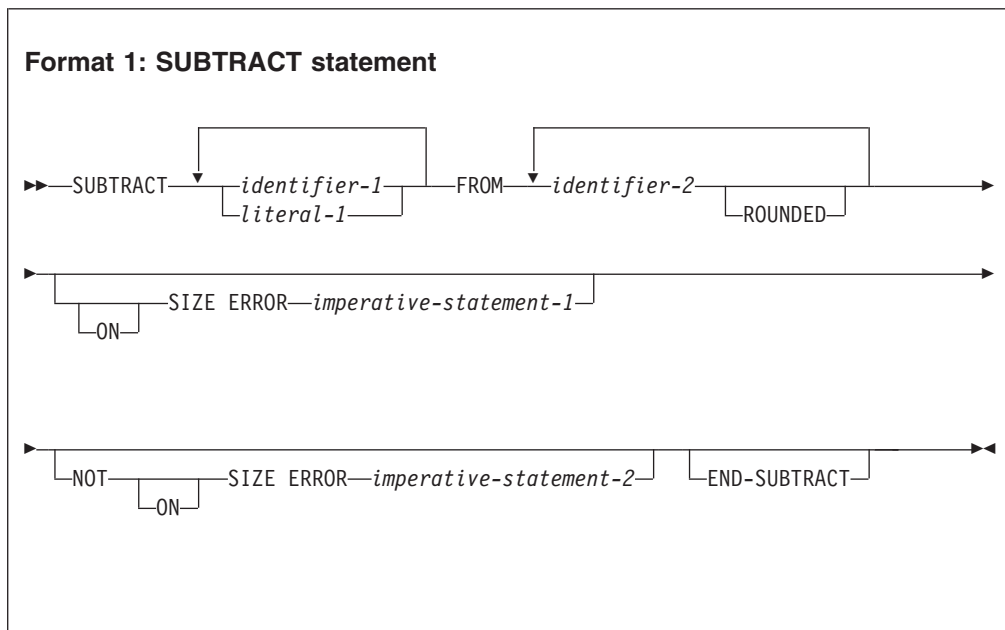
When the following STRING statement is executed, the results obtained will be like those illustrated in the figure after the statement.

```
STRING ID-1 ID-2 DELIMITED BY ID-3
      ID-4 ID-5 DELIMITED BY SIZE
      INTO ID-7 WITH POINTER ID-8
END-STRING
```



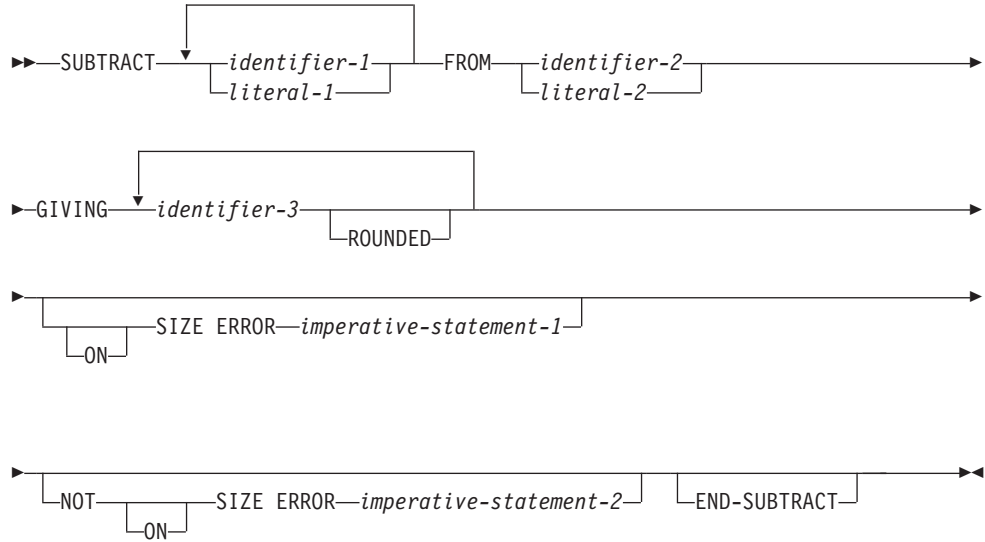
SUBTRACT statement

The SUBTRACT statement subtracts one numeric item, or the sum of two or more numeric items, from one or more numeric items, and stores the result.



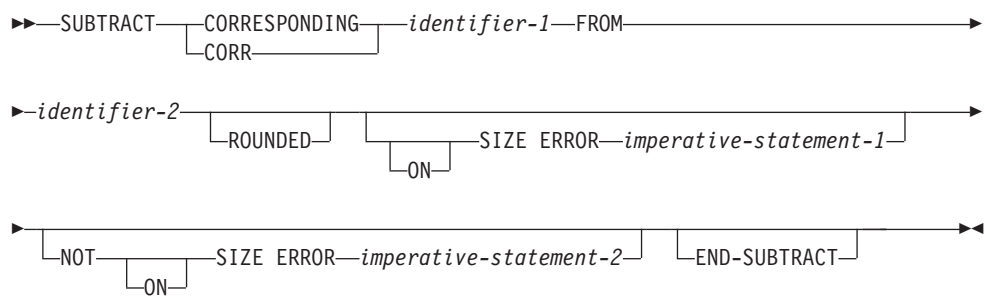
All identifiers or literals preceding the keyword **FROM** are added together and their sum is subtracted from and stored immediately in *identifier-2*. This process is repeated for each successive occurrence of *identifier-2*, in the left-to-right order in which *identifier-2* is specified.

Format 2: SUBTRACT statement with GIVING phrase



All identifiers or literals preceding the keyword FROM are added together and their sum is subtracted from *identifier-2* or *literal-2*. The result of the subtraction is stored as the new value of each data item referenced by *identifier-3*.

Format 3: SUBTRACT statement with CORRESPONDING phrase



Elementary data items within *identifier-1* are subtracted from, and the results are stored in, the corresponding elementary data items within *identifier-2*.

When the ARITH(COMPAT) compiler option is in effect, the composite of operands can contain a maximum of 30 digits. When the ARITH(EXTEND) compiler option is in effect, the composite of operands can contain a maximum of 31 digits. For more information about arithmetic intermediate results, see *Appendix A. Intermediate results and arithmetic precision in the COBOL for AIX Programming Guide*.

For all formats:

identifier

In format 1, must name an elementary numeric data item.

In format 2, must name an elementary numeric data item, unless the identifier follows the word GIVING. Each identifier following the word GIVING must name a numeric or numeric-edited elementary data item.

In format 3, must name an alphanumeric group item or a national group item.

The following restrictions apply to date fields:

- In format 1, *identifier-1* can specify at most one date field. If *identifier-1* specifies a date field, then every instance of *identifier-2* must specify a date field that is compatible with the date field specified by *identifier-1*. If *identifier-1* does not specify a date field, then *identifier-2* can specify one or more date fields, with no restriction on their DATE FORMAT clauses.
- In format 2, *identifier-1* and *identifier-2* can each specify at most one date field. If *identifier-1* specifies a date field, then the FROM *identifier-2* must be a date field that is compatible with the date field specified by *identifier-1*. *identifier-3* can specify one or more date fields. If *identifier-2* specifies a date field and *identifier-1* does not, then every instance of *identifier-3* must specify a date field that is compatible with the date field specified by *identifier-2*.
- In format 3, if an item within *identifier-1* is a date field, then the corresponding item within *identifier-2* must be a compatible date field.
- A year-last date field is allowed in a SUBTRACT statement only as *identifier-1* and when the result of the subtraction is a nondate.

There are two steps to determining the result of a SUBTRACT statement that involves one or more date fields:

1. Subtraction: determine the result of the subtraction operation, as described under “Subtraction that involves date fields” on page 261.
2. Storage: determine how the result is stored in the receiving field. (In formats 1 and 3, the receiving field is *identifier-2*; in format 3, the receiving field is the GIVING *identifier-3*.) For details, see “Storing arithmetic results that involve date fields” on page 261.

literal Must be a numeric literal.

Floating-point data items and literals can be used anywhere numeric data items and literals can be specified.

ROUNDED phrase

For information about the ROUNDED phrase, and for operand considerations, see “ROUNDED phrase” on page 292.

SIZE ERROR phrases

For information about the SIZE ERROR phrases, and for operand considerations, see “SIZE ERROR phrases” on page 293.

CORRESPONDING phrase (format 3)

See “CORRESPONDING phrase” on page 291.

END-SUBTRACT phrase

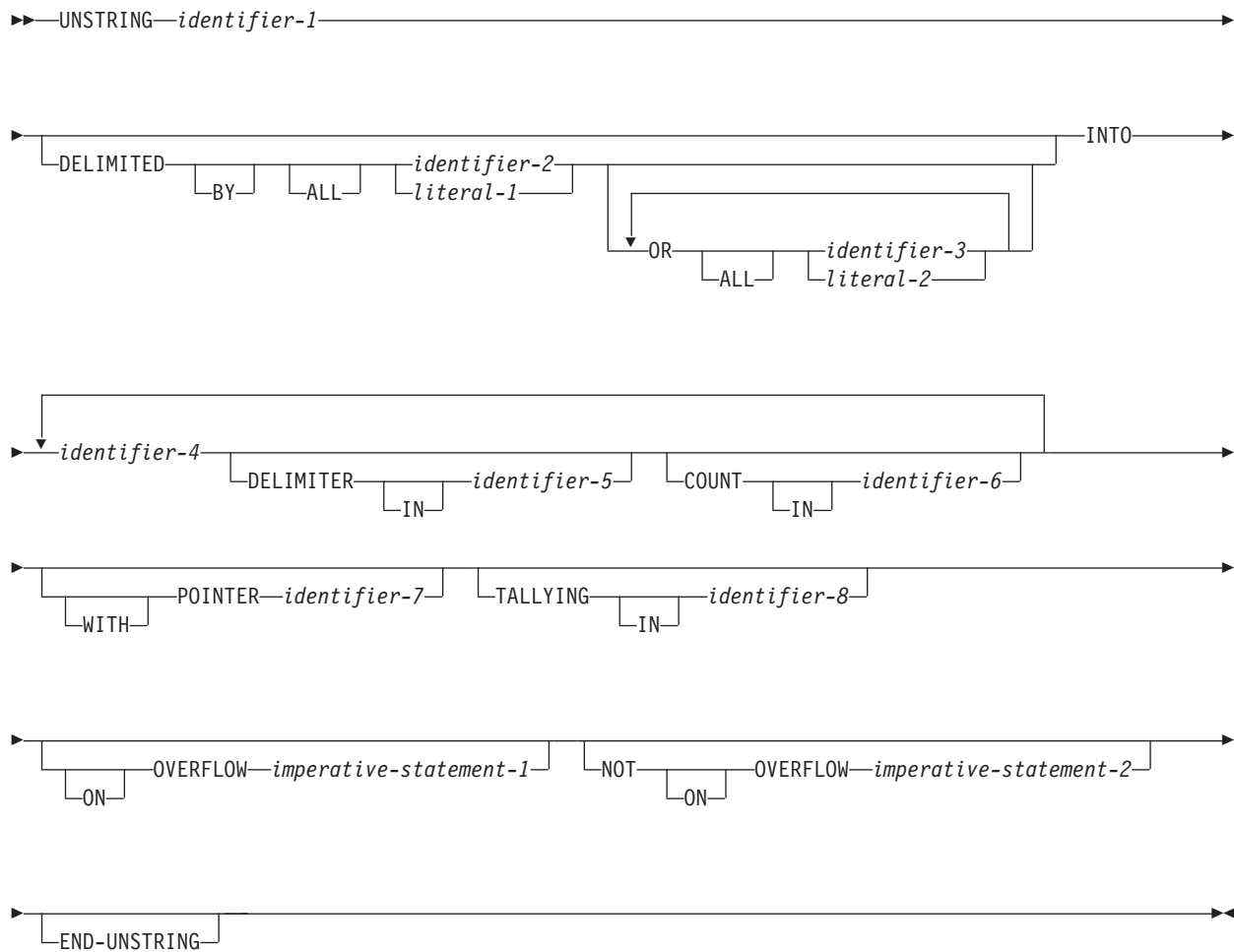
This explicit scope terminator serves to delimit the scope of the SUBTRACT statement. END-SUBTRACT permits a conditional SUBTRACT statement to be nested in another conditional statement. END-SUBTRACT can also be used with an imperative SUBTRACT statement.

For more information, see “Delimited scope statements” on page 289.

UNSTRING statement

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

Format



identifier-1

Represents the *sending field*. Data is transferred from this field to the data receiving fields (*identifier-4*).

identifier-1 must reference a data item of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

identifier-2, *literal-1*, *identifier-3*, *literal-2*

Specifies one or more delimiters.

identifier-2 and *identifier-3* must reference data items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, national, or national-edited.

literal-1 or *literal-2* must be of category alphanumeric, DBCS, or national and must not be a figurative constant that begins with the word ALL.

identifier-4

Specifies one or more receiving fields.

identifier-4 must reference a data item of category alphabetic, alphanumeric, numeric, DBCS, or national. If the referenced data item is of category numeric, its picture character-string must not contain the picture symbol P, and its usage must be DISPLAY or NATIONAL.

identifier-5

Specifies a field to receive the delimiter associated with *identifier-4*.

Identifier-5 must reference a data item of category alphabetic, alphanumeric, DBCS, or national.

identifier-6

Specifies a field to hold the count of characters that are transferred to *identifier-4*.

identifier-6 must be an integer data item defined without the symbol P in its PICTURE character-string.

identifier-7

Specifies a field to hold a relative character position during UNSTRING processing.

identifier-7 must be an integer data item defined without the symbol P in the PICTURE string. *identifier-7* must be described as a data item of sufficient size to contain a value equal to 1 plus the number of character positions in the data item referenced by *identifier-1*.

identifier-8

Specifies a field that is incremented by the number of delimited fields processed.

identifier-8 must be an integer data item defined without the symbol P in its PICTURE character-string.

The following rules apply

- If *identifier-4* references a data item of usage DISPLAY, *identifier-1*, *identifier-2*, *identifier-3*, and *identifier-5* must also reference data items of usage DISPLAY and all literals must be alphanumeric literals. Any figurative constant can be specified except NULL or one that begins with the word ALL. Each figurative constant represents a 1-character alphanumeric literal.
- If *identifier-4* references a data item of usage DISPLAY-1, *identifier-1*, *identifier-2*, *identifier-3*, and *identifier-5* must also reference data items of usage DISPLAY-1 and all literals must be DBCS literals. Figurative constant SPACE is the only figurative constant that can be specified. Each figurative constant represents a 1-character DBCS literal.
- If *identifier-4* references a data item of usage NATIONAL, *identifier-1*, *identifier-2*, *identifier-3*, and *identifier-5* must also reference data items of usage NATIONAL and all literals must be national literals. Any figurative constant can be specified except NULL or one that begins with the word ALL. Each figurative constant represents a 1-character national literal.
- None of the identifiers in an UNSTRING statement can be windowed date fields.

Count fields (*identifier-6*) and pointer fields (*identifier-7*) are incremented by number of character positions (alphanumeric, DBCS, or national), not by number of bytes.

One UNSTRING statement can take the place of a series of MOVE statements, except that evaluation or calculation of certain elements is performed only once, at the beginning of the execution of the UNSTRING statement. For more information, see "Values at the end of execution of the UNSTRING statement" on page 460.

The rules for moving are the same as those for a MOVE statement for an elementary sending item of the category of *identifier-1*, with the appropriate *identifier-4* as the receiving item (see "MOVE statement" on page 380). For example, rules for moving a DBCS item are used when *identifier-1* is a DBCS item.

DELIMITED BY phrase

This phrase specifies delimiters within the data that control the data transfer.

Each *identifier-2*, *identifier-3*, *literal-1*, or *literal-2* represents one delimiter.

If the DELIMITED BY phrase is *not* specified, the DELIMITER IN and COUNT IN phrases must *not* be specified.

ALL Multiple contiguous occurrences of any delimiters are treated as if there were only one occurrence; this one occurrence is moved to the delimiter receiving field (*identifier-5*), if specified. The delimiting characters in the sending field are treated as an elementary item of the same usage and category as *identifier-1* and are moved into the current delimiter receiving field according to the rules of the MOVE statement.

When DELIMITED BY ALL is *not* specified, and two or more contiguous occurrences of any delimiter are encountered, the current data receiving field (*identifier-4*) is filled with spaces or zeros, according to the description of the data receiving field.

Delimiter with two or more characters

A delimiter that contains two or more characters is recognized as a delimiter only if the delimiting characters are both of the following:

- Contiguous
- In the sequence specified in the sending field

Two or more delimiters

When two or more delimiters are specified, an OR condition exists, and each nonoverlapping occurrence of any one of the delimiters is recognized in the sending field in the sequence specified.

For example:

DELIMITED BY "AB" or "BC"

An occurrence of either AB or BC in the sending field is considered a delimiter. An occurrence of ABC is considered an occurrence of AB.

INTO phrase

This phrase specifies the fields where the data is to be moved.

identifier-4 represents the *data receiving fields*.

DELIMITER IN

This phrase specifies the fields where the delimiters are to be moved.

identifier-5 represents the *delimiter receiving fields*.

The DELIMITER IN phrase must *not* be specified if the DELIMITED BY phrase is *not* specified.

COUNT IN

This phrase specifies the field where the count of examined character positions is held.

identifier-6 is the *data count field* for each data transfer. Each field holds the count of examined character positions in the sending field, terminated by the delimiters or the end of the sending field, for the move to this receiving field. The delimiters are not included in this count.

The COUNT IN phrase must *not* be specified if the DELIMITED BY phrase is *not* specified.

POINTER phrase

When the POINTER phrase is specified, the value of the pointer field, *identifier-7*, behaves as if it were increased by 1 for each examined character position in the sending field. When execution of the UNSTRING statement is completed, the pointer field contains a value equal to its initial value plus the number of character positions examined in the sending field.

When this phrase is specified, the user must initialize the pointer field before execution of the UNSTRING statement begins.

TALLYING IN phrase

When the TALLYING phrase is specified, the area count field, *identifier-8*, contains (at the end of execution of the UNSTRING statement) a value equal to the initial value plus the number of data receiving areas acted upon.

When this phrase is specified, the user must initialize the area count field before execution of the UNSTRING statement begins.

ON OVERFLOW phrases

An overflow condition exists when:

- The pointer value (explicit or implicit) is less than 1.
- The pointer value (explicit or implicit) exceeds a value equal to the length of the sending field.
- All data receiving fields have been acted upon and the sending field still contains unexamined character positions.

When an overflow condition occurs

An overflow condition results in the following actions:

1. No more data is transferred.
2. The UNSTRING operation is terminated.
3. The NOT ON OVERFLOW phrase, if specified, is ignored.
4. Control is transferred to the end of the UNSTRING statement or, if the ON OVERFLOW phrase is specified, to *imperative-statement-1*.

imperative-statement-1

Statement or statements for dealing with an overflow condition.

If control is transferred to *imperative-statement-1*, execution continues according to the rules for each statement specified in *imperative-statement-1*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of *imperative-statement-1*, control is transferred to the end of the UNSTRING statement.

When an overflow condition does not occur

When, during execution of an UNSTRING statement, conditions that would cause an overflow condition are not encountered, then:

1. The transfer of data is completed.
2. The ON OVERFLOW phrase, if specified, is ignored.
3. Control is transferred to the end of the UNSTRING statement or, if the NOT ON OVERFLOW phrase is specified, to *imperative-statement-2*.

imperative-statement-2

Statement or statements for dealing with an overflow condition that does not occur.

If control is transferred to *imperative-statement-2*, execution continues according to the rules for each statement specified in *imperative-statement-2*. If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of *imperative-statement-2*, control is transferred to the end of the UNSTRING statement.

END-UNSTRING phrase

This explicit scope terminator serves to delimit the scope of the UNSTRING statement. END-UNSTRING permits a conditional UNSTRING statement to be nested in another conditional statement. END-UNSTRING can also be used with an imperative UNSTRING statement.

For more information, see “Delimited scope statements” on page 289.

Data flow

When the UNSTRING statement is initiated, data is transferred from the sending field to the current data receiving field, according to the following rules:

Stage 1: Examine

1. If the POINTER phrase is specified, the field is examined, beginning at the relative character position specified by the value in the pointer field.
If the POINTER phrase is *not* specified, the sending field character-string is examined, beginning with the leftmost character position.
2. If the DELIMITED BY phrase is specified, the examination proceeds from left to right, examining character positions one-by-one until a delimiter is encountered. If the end of the sending field is reached before a delimiter is found, the examination ends with the last character position in the sending field. If there are more receiving fields, the next one is selected; otherwise, an overflow condition occurs.
If the DELIMITED BY phrase is *not* specified, the number of character positions examined is equal to the size of the current data receiving field, as described in the table below. The size depends on the category treatment of the receiving field, as shown in Table 44 on page 362.

Table 57. Character positions examined when DELIMITED BY is not specified

If the receiving field is ...	The number of character positions examined is ...
Alphanumeric or alphabetic	Equal to the number of alphanumeric character positions in the current receiving field
DBCS	Equal to the number of DBCS character positions in the current receiving field
National	Equal to the number of national character positions in the current receiving field
Numeric	Equal to the number of character positions in the integer portion of the current receiving field
Described with the SIGN IS SEPARATE clause	1 less than the size of the current receiving field
Described as a variable-length data item	Determined by the size of the current receiving field at the beginning of the UNSTRING operation

Stage 2: Move

3. The examined character positions (excluding any delimiter characters) are treated as an elementary data item of the same data category as the sending field except for the cases shown in the table below.

Category of <i>identifier-1</i> (sending-field)	Category of elementary data item
Alphanumeric-edited	Alphanumeric
National-edited	National

That elementary data item is moved to the current data receiving field according to the rules for the MOVE statement for the categories of the sending and receiving fields as described in “MOVE statement” on page 380.

4. If the DELIMITER IN phrase is specified, the delimiting characters in the sending field are treated as an elementary alphanumeric item and are moved to the current delimiter receiving field, according to the rules for the MOVE statement. If the delimiting condition is the end of the sending field, the current delimiter receiving field is filled with spaces.

5. If the COUNT IN phrase is specified, a value equal to the number of examined character positions (excluding any delimiters) is moved into the data count field, according to the rules for an elementary move.

Stage 3: Successive iterations

6. If the DELIMITED BY phrase is specified, the sending field is further examined, beginning with the first character position to the right of the delimiter.
If the DELIMITED BY phrase is *not* specified, the sending field is further examined, beginning with the first character position to the right of the last character position examined.
7. For each succeeding data receiving field, this process of examining and moving is repeated until either of the following occurs:
 - All the characters in the sending field have been transferred.
 - There are no more unfilled data receiving fields.

Values at the end of execution of the UNSTRING statement

The following operations are performed only once, at the beginning of the execution of the UNSTRING statement:

- Calculations of subscripts, reference modifications, variable-lengths, variable locations
- Evaluations of functions

Therefore, if *identifier-4*, *identifier-5*, *identifier-6*, *identifier-7*, or *identifier-8* is used as a subscript, reference-modifier, or function argument in the UNSTRING statement, or affects the length or location of any of the identifiers in the UNSTRING statement, these values are determined at the beginning of the UNSTRING statement, and are *not* affected by any results of the UNSTRING statement.

Example of the UNSTRING statement

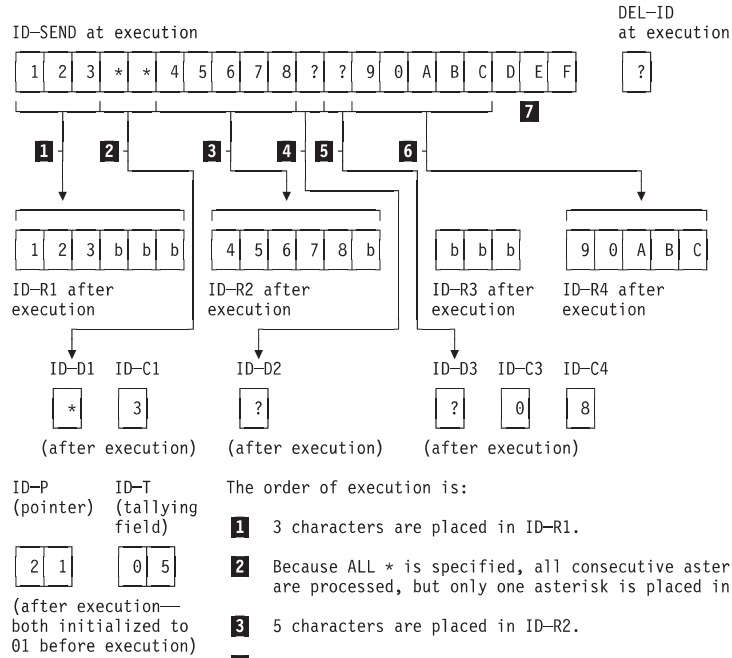
The following figure shows the execution results for an example of the UNSTRING statement.

```

UNSTRING ID-SEND DELIMITED BY DEL-ID OR ALL "*"
  INTO ID-R1 DELIMITER IN ID-D1 COUNT IN ID-C1
  ID-R2 DELIMITER IN ID-D2
  ID-R3 DELIMITER IN ID-D3 COUNT IN ID-C3
  ID-R4 COUNT IN ID-C4
WITH POINTER ID-P
TALLYING IN ID-T
ON OVERFLOW GO TO OFLOW-EXIT.

```

(All the data receiving fields are defined as alphanumeric)



- 3 characters are placed in ID-R1.
- Because ALL * is specified, all consecutive asterisks are processed, but only one asterisk is placed in ID-D1.
- 5 characters are placed in ID-R2.
- A ? is placed in ID-D2. The current receiving field is now ID-R3.
- A ? is placed in ID-D3; ID-R3 is filled with spaces; no characters are transferred, so 0 is placed in ID-C3.
- No delimiter is encountered before 5 characters fill ID-R4; 8 is placed in ID-C4, representing the number of characters examined since the last delimiter.
- ID-P is updated to 21, the total length of the sending field + 1; ID-T is updated to 5, the number of fields acted upon + 1. Since there are no unexamined characters in the ID-SEND, the OVERFLOW EXIT is not taken.

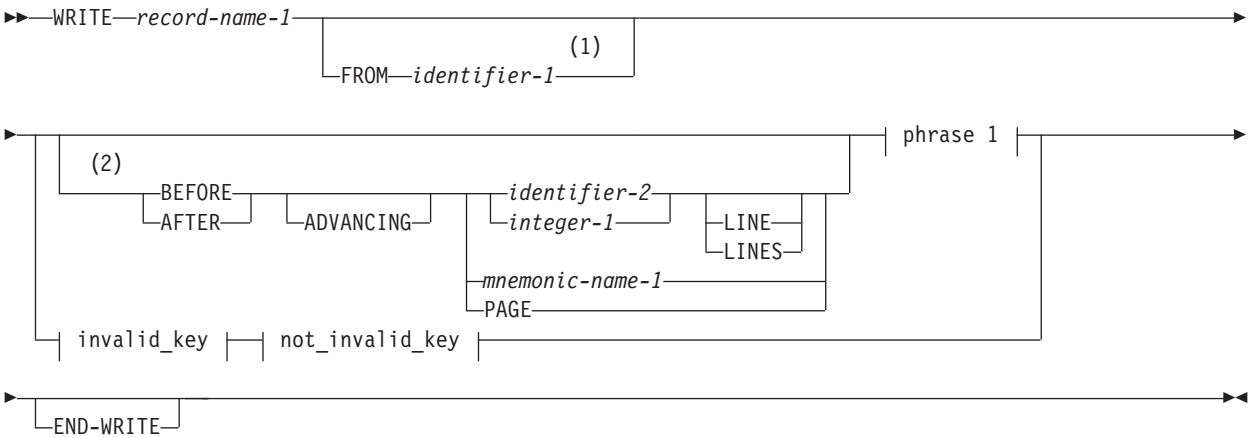
WRITE statement

The WRITE statement releases a logical record to an output or input/output file.

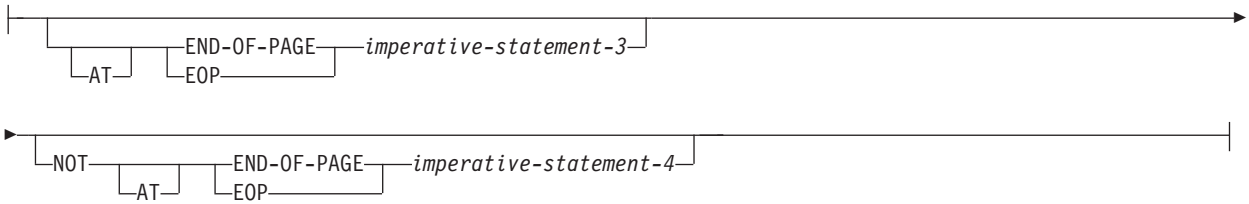
When the WRITE statement is executed:

- The associated sequential file must be open in OUTPUT or EXTEND mode.
- The associated indexed or relative file must be open in OUTPUT, I-O, or EXTEND mode.

Format 1: WRITE statement for sequential files



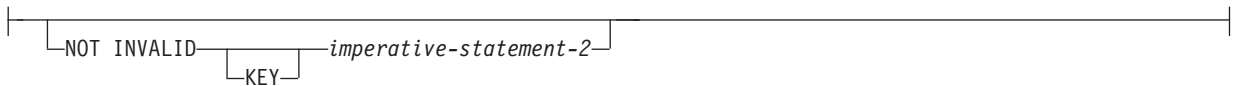
phrase 1:



invalid_key:



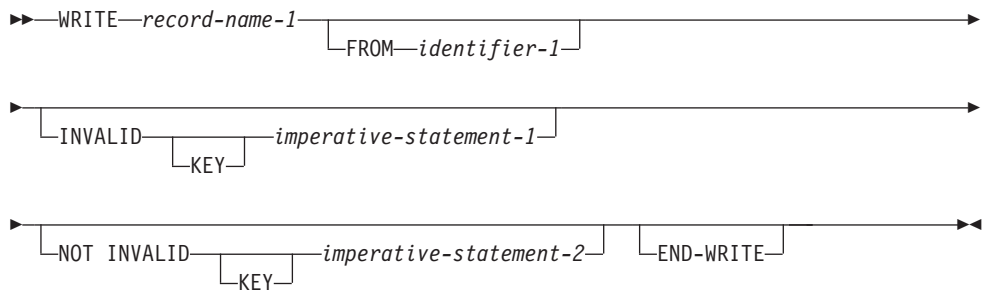
not_invalid_key:



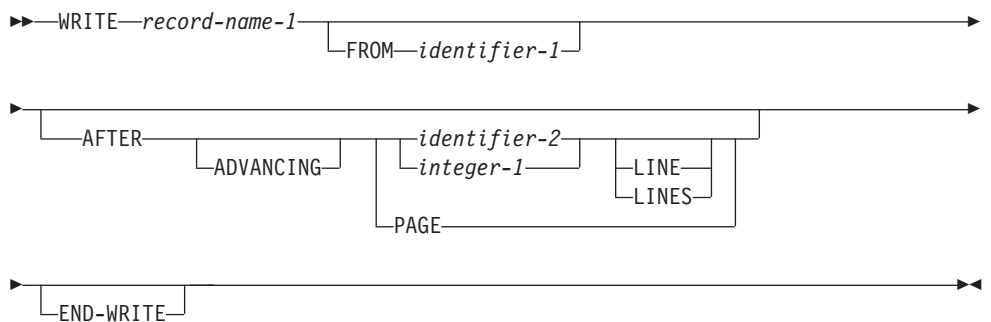
Notes:

- 1 The BEFORE, AFTER, INVALID KEY, and AT END OF PAGE phrases are not valid for STL files.
- 2 The BEFORE and AFTER phrases are not valid for RSD files.

Format 2: WRITE statement for indexed and relative files



Format 3: WRITE statement for line-sequential files



record-name-1

Must be defined in a data division FD entry. *record-name-1* can be qualified. It must not be associated with a sort or merge file.

For relative files, the number of character positions in the record being written can be different from the number of character positions in the record being replaced.

FROM phrase

The result of the execution of the WRITE statement with the FROM *identifier-1* phrase is equivalent to the execution of the following statements in the order specified:

```
MOVE identifier-1 TO record-name-1.  
WRITE record-name-1.
```

The MOVE is performed according to the rules for a MOVE statement without the CORRESPONDING phrase.

identifier-1

identifier-1 can reference any of the following:

- A data item defined in the working-storage section, the local-storage section, or the linkage section
- A record description for another previously opened file
- An alphanumeric function

- A national function

identifier-1 must be a valid sending item for a MOVE statement with *record-name-1* as the receiving item.

identifier-1 and *record-name-1* must not refer to the same storage area.

After the WRITE statement is executed, the information is still available in *identifier-1*. (See "INTO and FROM phrases" on page 301 under "Common processing facilities".)

identifier-2

Must be an integer data item.

ADVANCING phrase

The ADVANCING phrase controls positioning of the output record on the page.

When you use WRITE ADVANCING with environment names C01-C012 or S01-S05, one line is advanced.

ADVANCING phrase rules

When the ADVANCING phrase is specified, the following rules apply:

1. When BEFORE ADVANCING is specified, the line is printed before the page is advanced.
2. When AFTER ADVANCING is specified, the page is advanced before the line is printed.
3. When *identifier-2* is specified, the page is advanced the number of lines equal to the current value in *identifier-2*. *identifier-2* must name an elementary integer data item. *identifier-2* cannot name a windowed date field.
4. When integer is specified, the page is advanced the number of lines equal to the value of integer.
5. Integer or the value in *identifier-2* can be zero.
6. When PAGE is specified, the record is printed on the logical page BEFORE or AFTER (depending on the phrase used) the device is positioned to the next logical page. If PAGE has no meaning for the device used, then BEFORE or AFTER (depending on the phrase specified) ADVANCING 1 LINE is provided.
If the FD entry contains a LINAGE clause, the repositioning is to the first printable line of the next page, as specified in that clause. If the LINAGE clause is omitted, the repositioning is to line 1 of the next succeeding page.
7. The BEFORE ADVANCING and AFTER ADVANCING phrases are not supported for RSD files.

When the ADVANCING phrase is omitted, automatic line advancing is provided, as if AFTER ADVANCING 1 LINE had been specified.

LINAGE-COUNTER rules

If the LINAGE clause is specified for this file, the associated LINAGE-COUNTER special register is modified during the execution of the WRITE statement, according to the following rules:

1. If ADVANCING PAGE is specified, LINAGE-COUNTER is reset to 1.
2. If ADVANCING *identifier-2* or *integer* is specified, LINAGE-COUNTER is increased by the value in *identifier-2* or *integer*.

3. If the ADVANCING phrase is omitted, LINAGE-COUNTER is increased by 1.
4. When the device is repositioned to the first available line of a new page, LINAGE-COUNTER is reset to 1.

END-OF-PAGE phrases

When END-OF-PAGE is specified, and the logical end of the printed page is reached during execution of the WRITE statement, the END-OF-PAGE imperative-statement is executed. When the END-OF-PAGE phrase is specified, the FD entry for this file must contain a LINAGE clause.

The logical end of the printed page is specified in the associated LINAGE clause.

An END-OF-PAGE condition is reached when execution of a WRITE END-OF-PAGE statement causes printing or spacing within the footing area of a page body. This occurs when execution of such a WRITE statement causes the value in the LINAGE-COUNTER special register to equal or exceed the value specified in the WITH FOOTING phrase of the LINAGE clause. The WRITE statement is executed, and then the END-OF-PAGE imperative-statement is executed.

An automatic page overflow condition is reached whenever the execution of any given WRITE statement (with or without the END-OF-PAGE phrase) cannot be completely executed within the current page body. This occurs when a WRITE statement, if executed, would cause the value in the LINAGE-COUNTER to exceed the number of lines for the page body specified in the LINAGE clause. In this case, the line is printed BEFORE or AFTER (depending on the option specified) the device is repositioned to the first printable line on the next logical page, as specified in the LINAGE clause. If the END-OF-PAGE phrase is specified, the END-OF-PAGE imperative-statement is then executed.

If the WITH FOOTING phrase of the LINAGE clause is not specified, the automatic page overflow condition exists because no end-of-page condition (as distinct from the page overflow condition) can be detected.

If the WITH FOOTING phrase is specified, but the execution of a given WRITE statement would cause the LINAGE-COUNTER to exceed both the footing value and the page body value specified in the LINAGE clause, then both the end-of-page condition and the automatic page overflow condition occur simultaneously.

The keywords END-OF-PAGE and EOP are equivalent.

You can specify both the ADVANCING PAGE phrase and the END-OF-PAGE phrase in a single WRITE statement.

INVALID KEY phrases

An invalid key condition is caused by the following:

- For sequential files, an attempt is made to write beyond the externally defined boundary of the file.
- For indexed files:
 - An attempt is made to write beyond the externally defined boundary of the file.

- ACCESS SEQUENTIAL is specified and the file is opened OUTPUT, and the value of the prime record key is not greater than that of the previous record.
- The file is opened OUTPUT or I-O and the value of the prime record key equals that of an already existing record.
- For relative files:
 - An attempt is made to write beyond the externally defined boundary of the file.
 - When the access mode is random or dynamic and the RELATIVE KEY data item specifies a record that already exists in the file.
 - The number of significant digits in the relative record number is larger than the size of the relative key data item for the file.

When an invalid key condition occurs:

- If the INVALID KEY phrase is specified, *imperative-statement-1* is executed. For details of invalid key processing, see “Invalid key condition” on page 300.
- Otherwise, the WRITE statement is unsuccessful and the contents of *record-name* are unaffected and the following occurs:
 - For sequential files, the file status key, if specified, is updated and an EXCEPTION/ERROR condition exists.
 If an explicit or implicit EXCEPTION/ERROR procedure is specified for the file, the procedure is executed. If no such procedure is specified, the results are unpredictable.
 - For relative and indexed files, program execution proceeds according to the rules described by “Invalid key condition” on page 300 under “Common processing facilities”.
 The INVALID KEY conditions that apply to a relative file in OPEN OUTPUT mode also apply to one in OPEN EXTEND mode.
- If the NOT INVALID KEY phrase is specified and a valid key condition exists at the end of the execution of the WRITE statement, control is passed to *imperative-statement-4*.

Both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure can be omitted.

END-WRITE phrase

This explicit scope terminator serves to delimit the scope of the WRITE statement. END-WRITE permits a conditional WRITE statement to be nested in another conditional statement. END-WRITE can also be used with an imperative WRITE statement.

For more information, see “Delimited scope statements” on page 289.

WRITE for sequential files

The maximum record size for sequential files is established at the time the file is created and cannot subsequently be changed.

After the WRITE statement is executed, the logical record is no longer available in *record-name-1* unless either:

- The associated file is named in a SAME RECORD AREA clause (in which case, the record is also available as a record of the other files named in the SAME RECORD AREA clause)
- The WRITE statement is unsuccessful because of a boundary violation.

In either of these two cases, the logical record is still available in *record-name-1*.

The file position indicator is not affected by execution of the WRITE statement.

The number of character positions required to store the record in a file might or might not be the same as the number of character positions defined by the logical description of that record in the COBOL program. (See “PICTURE clause editing” on page 211 and “USAGE clause” on page 229.)

If the FILE STATUS clause is specified in the file-control entry, the associated file status key is updated when the WRITE statement is executed, whether or not execution is successful.

The WRITE statement can only be executed for a sequential file opened in OUTPUT mode.

WRITE for indexed files

Before the WRITE statement is executed for indexed files, you must set the prime record key (the RECORD KEY data item, as defined in the file-control entry) to the desired value. Note that RECORD KEY values must be unique within a file.

If the ALTERNATE RECORD KEY clause is also specified in the file-control entry, each alternate record key must be unique, unless the DUPLICATES phrase is specified. If the DUPLICATES phrase is specified, alternate record key values might not be unique. In this case, the system stores the records so that later sequential access to the records allows retrieval in the same order in which they were stored.

When ACCESS IS SEQUENTIAL is specified in the file-control entry, records must be released in ascending order of RECORD KEY values.

When ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified in the file-control entry, records can be released in any programmer-specified order.

WRITE for relative files

For relative record OUTPUT files, the WRITE statement causes the following actions:

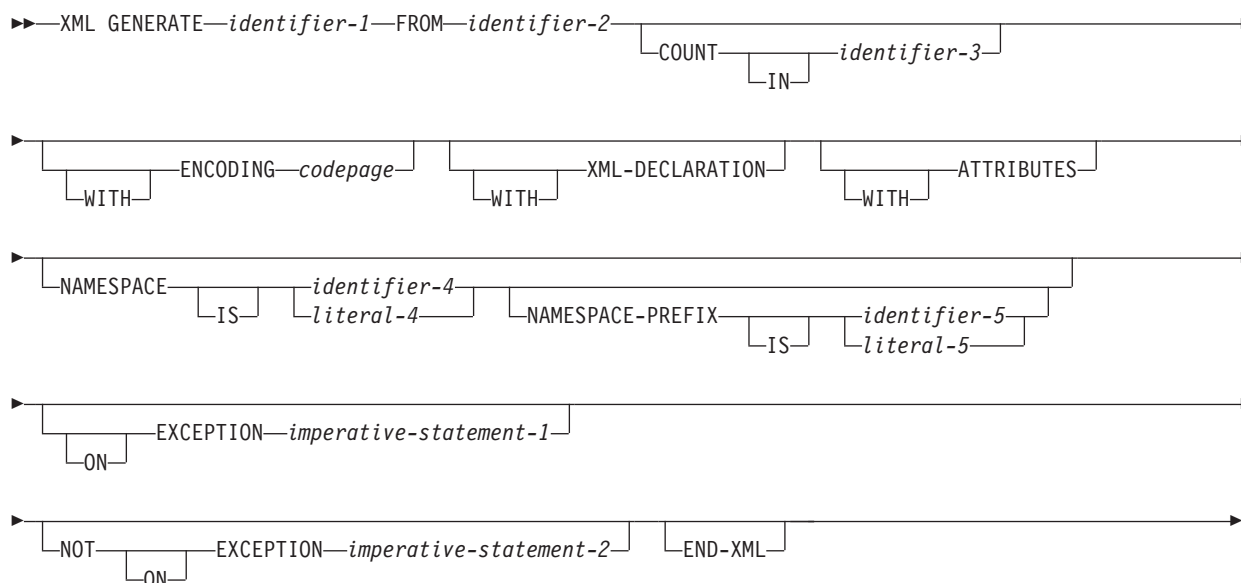
- If ACCESS IS SEQUENTIAL is specified:
The first record released has relative record number 1, the second record released has relative record number 2, the third number 3, and so on.
If the RELATIVE KEY is specified in the file-control entry, the relative record number of the record just released is placed in the RELATIVE KEY during execution of the WRITE statement.
- If ACCESS IS RANDOM or ACCESS IS DYNAMIC is specified, the RELATIVE KEY must contain the desired relative record number for this record before the WRITE statement is issued. When the WRITE statement is executed, this record is placed at the specified relative record number position in the file.

For I-O files, either ACCESS IS RANDOM or ACCESS IS DYNAMIC must be specified; the WRITE statement inserts new records into the file. The RELATIVE KEY must contain the desired relative record number for this record before the WRITE statement is issued. When the WRITE statement is executed, this record is placed at the specified relative record number position in the file.

XML GENERATE statement

The XML GENERATE statement converts data to XML format.

Format



identifier-1

The receiving area for a generated XML document. *identifier-1* must reference one of the following:

- An elementary data item of category alphanumeric
- An alphanumeric group item
- An elementary data item of category national
- A national group item

When *identifier-1* references a national group item, *identifier-1* is processed as an elementary data item of category national. When *identifier-1* references an alphanumeric group item, *identifier-1* is treated as though it were an elementary data item of category alphanumeric.

identifier-1 must not be described with the JUSTIFIED clause, and cannot be a function identifier. *identifier-1* can be subscripted or reference modified.

identifier-1 must not overlap *identifier-2*, *identifier-3*, *codepage* (if an identifier), *identifier-4*, or *identifier-5*.

Either *identifier-1* must reference a data item of category national, or if *identifier-1* is category alphanumeric, the document encoding must be Unicode UTF-8, if any of the following statements are true:

- *identifier-4* or *identifier-5* references a data item of category national.
- *literal-4* or *literal-5* is of category national.
- *codepage* is a national literal or references a data item of category national.

- The generated XML includes data from *identifier-2* for:
 - Any data item of class national or class DBCS
 - Any data item with a multibyte name (that is, a data item whose name consists of multibyte characters)
 - Any data item of class alphanumeric that contains multibyte characters

The encoding of the generated XML output is described further in the documentation of the ENCODING phrase below.

identifier-1 must be large enough to contain the generated XML document. Typically, it should be from five to 10 times the size of *identifier-2*, depending on the length of the data-name or data-names within *identifier-2*. If *identifier-1* is not large enough, an error condition exists at the end of the XML GENERATE statement.

identifier-2

The group or elementary data item to be converted to XML format.

If *identifier-2* references a national group item, *identifier-2* is processed as a group item. When *identifier-2* includes a subordinate national group item, that subordinate item is processed as a group item.

identifier-2 cannot be a function identifier or be reference modified, but it can be subscripted.

identifier-2 must not overlap *identifier-1* or *identifier-3*.

identifier-2 must not specify the RENAMES clause.

The following data items specified by *identifier-2* are ignored by the XML GENERATE statement:

- Any subordinate unnamed elementary data items or elementary FILLER data items
- Any slack bytes inserted for SYNCHRONIZED items
- Any data item subordinate to *identifier-2* that is described with the REDEFINES clause or that is subordinate to such a redefining item
- Any data item subordinate to *identifier-2* that is described with the RENAMES clause
- Any group data item all of whose subordinate data items are ignored

All data items specified by *identifier-2* that are not ignored according to the rules above must satisfy the following conditions:

- Each elementary data item must either have class alphabetic, alphanumeric, numeric, or national, or be an index data item. (That is, no elementary data item can be described with the USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE phrase.)
- There must be at least one such elementary data item.
- Each non-FILLER data-name must be unique within any immediately superordinate group data item.
- Any multibyte data-names, when converted to Unicode, must be legal as names in the XML specification, version 1.0. For details about the XML specification, see *XML specification*.
- The data items must not specify the DATE FORMAT clause, or the DATEPROC compiler option must not be in effect.

For example, consider the following data declaration:

```

01 STRUCT.
  02 STAT PIC X(4).
  02 IN-AREA PIC X(100).
  02 OK-AREA REDEFINES IN-AREA.
    03 FLAGS PIC X.
    03 PIC X(3).
    03 COUNTER USAGE COMP-5 PIC S9(9).
    03 ASFPTR REDEFINES COUNTER USAGE FUNCTION-POINTER.
    03 UNREFERENCED PIC X(92).
  02 NG-AREA1 REDEFINES IN-AREA.
    03 FLAGS PIC X.
    03 PIC X(3).
    03 PTR USAGE POINTER.
    03 ASNUM REDEFINES PTR USAGE COMP-5 PIC S9(9).
    03 PIC X(92).
  02 NG-AREA2 REDEFINES IN-AREA.
    03 FN-CODE PIC X.
    03 UNREFERENCED PIC X(3).
    03 QTYONHAND USAGE BINARY PIC 9(5).
    03 DESC USAGE NATIONAL PIC N(40).
    03 UNREFERENCED PIC X(12).

```

The following data items from the example above can be specified as *identifier-2*:

- STRUCT, of which subordinate data items STAT and IN-AREA would be converted to XML format. (OK-AREA, NG-AREA1, and NG-AREA2 are ignored because they specify the REDEFINES clause.)
- OK-AREA, of which subordinate data items FLAGS, COUNTER, and UNREFERENCED would be converted. (The item whose data description entry specifies 03 PIC X(3) is ignored because it is an elementary FILLER data item. ASFPTR is ignored because it specifies the REDEFINES clause.)
- Any of the elementary data items that are subordinate to STRUCT except:
 - ASFPTR or PTR (disallowed usage)
 - UNREFERENCED OF NG-AREA2 (nonunique names for data items that are otherwise eligible)
 - Any FILLER data items

The following data items cannot be specified as *identifier-2*:

- NG-AREA1, because subordinate data item PTR specifies USAGE POINTER but does not specify the REDEFINES clause. (PTR would be ignored if it specified the REDEFINES clause.)
- NG-AREA2, because subordinate elementary data items have the nonunique name UNREFERENCED.

COUNT IN phrase

If the COUNT IN phrase is specified, *identifier-3* contains (after execution of the XML GENERATE statement) the count of generated XML character encoding units. If *identifier-1* (the receiver) has category national, the count is in UTF-16 character encoding units. For all other encodings (including UTF-8), the count is in bytes.

identifier-3

The data count field. Must be an integer data item defined without the symbol P in its picture string.

identifier-3 must not overlap *identifier-1*, *identifier-2*, *codepage* (if an identifier), *identifier-4*, or *identifier-5*.

ENCODING phrase

The ENCODING phrase, if specified, determines the encoding of the generated XML document.

codepage

Must be an unsigned integer data item, an unsigned integer literal, an alphanumeric literal, a national literal, or reference a data item of category alphanumeric or national. If *codepage* is a literal, it must not be a figurative constant.

If *codepage* is of class alphanumeric or national, it must identify a primary or alias code-page name that is supported by ICU conversion libraries (see *International Components for Unicode: Converter Explorer*). If *codepage* is an integer, the integer must be a valid CCSID number.

If *identifier-1* references a data item of category national, *codepage* must identify UTF-16 in big-endian format.

If *identifier-1* references a data item of category alphanumeric, *codepage* must identify UTF-8 or a single-byte ASCII or EBCDIC code page:

- If the CHAR(EBCDIC) compiler option is not in effect or the data description entry for *identifier-1* contains the NATIVE phrase, *codepage* must identify UTF-8 or a single-byte ASCII code page.
- If CHAR(EBCDIC) is in effect and the data description entry for *identifier-1* does not contain the NATIVE phrase, *codepage* must identify a single-byte EBCDIC code page.

If *codepage* is an identifier, it must not overlap *identifier-1* or *identifier-3*.

If the ENCODING phrase is omitted and *identifier-1* is of category national, the document encoding is Unicode UTF-16 in big-endian format.

If the ENCODING phrase is omitted and *identifier-1* is of category alphanumeric, then:

- If CHAR(EBCDIC) is not in effect or the data description entry for *identifier-1* contains the NATIVE phrase, the XML document is encoded using the code page from the runtime locale.
- If CHAR(EBCDIC) option is in effect and the data description entry for *identifier-1* does not contain the NATIVE phrase, the XML document is encoded using the code page from the EBCDIC_CODEPAGE environment variable. If EBCDIC_CODEPAGE is not set, the encoding is the default EBCDIC code page associated with the runtime locale.

XML-DECLARATION phrase

If the XML-DECLARATION phrase is specified, the generated XML document starts with an XML declaration that includes the XML version information and an encoding declaration.

If *identifier-1* is of category national, the encoding declaration has the value UTF-16 (encoding="UTF-16").

If *identifier-1* is of category alphanumeric, the encoding declaration is derived from the ENCODING phrase, if specified, or from the runtime locale or the EBCDIC_CODEPAGE environment variable if the ENCODING phrase is not specified. See the description of the ENCODING phrase for further details.

For an example of the effect of coding the XML-DECLARATION phrase, see *Generating XML output* in the *COBOL for AIX Programming Guide*.

If the XML-DECLARATION phrase is omitted, the generated XML document does not include an XML declaration.

ATTRIBUTES phrase

If the ATTRIBUTES phrase is specified, each eligible item included in the generated XML document is expressed as an attribute of the XML element that corresponds to the data item immediately superordinate to that eligible item, rather than as a child element of the XML element. To be eligible, a data item must be elementary, must have a name other than FILLER, and must not specify an OCCURS clause in its data description entry.

For an example of the effect of the ATTRIBUTES phrase, see *Generating XML output* in the *COBOL for AIX Programming Guide*.

NAMESPACE and NAMESPACE-PREFIX phrases

Use the NAMESPACE phrase to identify a *namespace* for the generated XML document. If the NAMESPACE phrase is not specified, or if *identifier-4* has length zero or contains all spaces, the element names of XML documents produced by the XML GENERATE statement are not in any namespace.

Use the NAMESPACE-PREFIX phrase to qualify the start and end tag of each element in the generated XML document with a prefix.

If the NAMESPACE-PREFIX phrase is not specified, or if *identifier-5* is of length zero or contains all spaces, the namespace specified by the NAMESPACE phrase specifies the default namespace for the document. In this case, the namespace declared on the root element applies by default to each element name in the document, including that of the root element. (Default namespace declarations do not apply directly to attribute names.)

If the NAMESPACE-PREFIX phrase is specified, and *identifier-5* is not of length zero and does not contain all spaces, then the start and end tag of each element in the generated document is qualified with the specified prefix. The prefix should therefore preferably be short. When the XML GENERATE statement is executed, the prefix must be a valid XML name, but without the colon (:), as defined in *Namespaces in XML 1.0*. The prefix can have trailing spaces, which are removed before use.

identifier-4, literal-4; identifier-5, literal-5

identifier-4, literal-4: The namespace identifier, which must be a valid *Uniform Resource Identifier (URI)* as defined in *Uniform Resource Identifier (URI): Generic Syntax*.

identifier-5, literal-5: The namespace prefix, which serves as an alias for the namespace identifier.

identifier-4 and *identifier-5* must reference data items of category alphanumeric or national.

identifier-4 and *identifier-5* must not overlap *identifier-1* or *identifier-3*.

literal-4 and *literal-5* must be of category alphanumeric or national, and must not be figurative constants.

For full details about namespaces, see *Namespaces in XML 1.0*.

For examples that show the use of the NAMESPACE and NAMESPACE-PREFIX phrases, see *Generating XML output* in the *COBOL for AIX Programming Guide*.

ON EXCEPTION phrase

An exception condition exists when an error occurs during generation of the XML document, for example if *identifier-1* is not large enough to contain the generated XML document. In this case, XML generation stops and the content of the receiver, *identifier-1*, is undefined. If the COUNT IN phrase is specified, *identifier-3* contains the number of character positions that were generated, which can range from 0 to the length of *identifier-1*.

If the ON EXCEPTION phrase is specified, control is transferred to *imperative-statement-1*. If the ON EXCEPTION phrase is not specified, the NOT ON EXCEPTION phrase, if any, is ignored, and control is transferred to the end of the XML GENERATE statement. Special register XML-CODE contains an exception code, as detailed in *Handling XML GENERATE exceptions* in the *COBOL for AIX Programming Guide*.

NOT ON EXCEPTION phrase

If an exception condition does not occur during generation of the XML document, control is passed to *imperative-statement-2*, if specified, otherwise to the end of the XML GENERATE statement. The ON EXCEPTION phrase, if specified, is ignored. Special register XML-CODE contains zero after execution of the XML GENERATE statement.

END-XML phrase

This explicit scope terminator delimits the scope of XML GENERATE or XML PARSE statements. END-XML permits a conditional XML GENERATE or XML PARSE statement (that is, an XML GENERATE or XML PARSE statement that specifies the ON EXCEPTION or NOT ON EXCEPTION phrase) to be nested in another conditional statement.

The scope of a conditional XML GENERATE or XML PARSE statement can be terminated by:

- An END-XML phrase at the same level of nesting
- A separator period

END-XML can also be used with an XML GENERATE or XML PARSE statement that does not specify either the ON EXCEPTION or the NOT ON EXCEPTION phrase.

For more information on explicit scope terminators, see “Delimited scope statements” on page 289.

Nested XML GENERATE or XML PARSE statements

When a given XML GENERATE or XML PARSE statement appears as *imperative-statement-1* or *imperative-statement-2*, or as part of *imperative-statement-1* or *imperative-statement-2* of another XML GENERATE or XML PARSE statement, that given XML GENERATE or XML PARSE statement is a *nested* XML GENERATE or XML PARSE statement.

Nested XML GENERATE or XML PARSE statements are considered to be matched XML GENERATE and END-XML combinations, or XML PARSE and END-XML combinations, proceeding from left to right. Thus, any END-XML phrase that is encountered is matched with the nearest preceding XML GENERATE or XML PARSE statement that has not been implicitly or explicitly terminated.

Operation of XML GENERATE

The content of each eligible elementary data item within *identifier-2* is converted to character format as described under “Format conversion of elementary data” on page 477 and “Trimming of generated XML data” on page 478. Only the first definition of each storage area is processed. Redefinitions of data items are not included. Data items that are effectively defined by the RENAMEs clause are also not included.

The converted content is then inserted as element character content, or, if the ATTRIBUTES phrase is specified and the data item is eligible to be expressed as an attribute, as the value of the attribute, in the generated XML document.

The XML element names and attribute names are derived from the data-names within *identifier-2* as described under “XML element name and attribute name formation” on page 479. The names of group items that contain the selected elementary items are retained as parent elements. If the NAMESPACE-PREFIX phrase is specified, the prefix value, minus any trailing spaces, is used to qualify the start and end tag of each element.

No extra white space (new lines, indentation, and so forth) is inserted to make the generated XML more readable. An XML declaration is generated if the XML-DECLARATION phrase is specified.

If the receiving area specified by *identifier-1* is not large enough to contain the resulting XML document, an error condition exists. See the description of the ON EXCEPTION phrase above for details.

If *identifier-1* is longer than the generated XML document, only that part of *identifier-1* in which XML is generated is changed. The rest of *identifier-1* contains the data that was present before this execution of the XML GENERATE statement. To avoid referring to that data, either initialize *identifier-1* to spaces before the XML GENERATE statement or specify the COUNT IN phrase.

If the COUNT IN phrase is specified, *identifier-3* contains (after execution of the XML GENERATE statement) the total number of character positions (UTF-16 encoding units or bytes) that were generated. You can use *identifier-3* as a reference modification length field to refer to the part of *identifier-2* that contains the generated XML document.

After execution of the XML GENERATE statement, special register XML-CODE contains either zero, which indicates successful completion, or a nonzero exception code. For details, see *Handling XML GENERATE exceptions* in the *COBOL for AIX Programming Guide*.

The XML PARSE statement also uses special register XML-CODE. Therefore if you code an XML GENERATE statement in the processing procedure of an XML PARSE statement, save the value of XML-CODE before that XML GENERATE statement executes and restore the saved value after the XML GENERATE statement terminates.

A byte order mark is not generated for XML documents that have Unicode encoding.

Format conversion of elementary data

Elementary data items within *identifier-2* are converted in a sequence of several steps, some of them optional, as described below.

Conversion to character format:

Elementary data items are converted to character format depending on the type of the data item:

- Data items of category alphabetic, alphanumeric, alphanumeric-edited, DBCS, external floating-point, national, national-edited, and numeric-edited are not converted.
- Fixed-point numeric data items other than COMPUTATIONAL-5 (COMP-5) binary data items or binary data items compiled with the TRUNC(BIN) compiler option are converted as if they were moved to a numeric-edited item that has:
 - As many integer positions as the numeric item has, but with at least one integer position
 - An explicit decimal point, if the numeric item has at least one decimal position
 - The same number of decimal positions as the numeric item has
 - A leading '-' picture symbol if the data item is signed (has an S in its PICTURE clause)
- COMPUTATIONAL-5 (COMP-5) binary data items or binary data items compiled with the TRUNC(BIN) compiler option are converted in the same way as the other fixed-point numeric items, except for the number of integer positions. The number of integer positions is computed depending on the number of '9' symbols in the picture character string as follows:
 - 5 minus the number of decimal places, if the data item has 1 to 4 '9' picture symbols
 - 10 minus the number of decimal places, if the data item has 5 to 9 '9' picture symbols
 - 20 minus the number of decimal places, if the data item has 10 to 18 '9' picture symbols
- Internal floating-point data items are converted as if they were moved to a data item as follows:
 - For COMP-1: an external floating-point data item with PICTURE -9.9(8)E+99
 - For COMP-2: an external floating-point data item with PICTURE -9.9(17)E+99 (illegal because of the number of digit positions)

For native (IEEE) floating-point items, the special values positive infinity, negative infinity, and NaN (not-a-number) are represented as 'INF', '-INF', and 'NaN', respectively.
- Index data items are converted as if they were declared USAGE COMP-5 PICTURE S9(9).

Trimming:

After any conversion to character format, leading and trailing spaces and leading zeroes are eliminated, as described under “Trimming of generated XML data” on page 478.

Conversion to the document encoding:

All values are converted as necessary to the encoding of the document, as follows. If *identifier-1* is:

- Category alphanumeric, and either the NATIVE phrase was specified in the data description or CHAR(NATIVE) was in effect, values are converted to UTF-8 or to the chosen ASCII code page.
- Category alphanumeric, and the NATIVE phrase was not specified in the data description, and CHAR(EBCDIC) was in effect, values are converted to the chosen EBCDIC code page.
- Category national: Any nonnational values are converted to national format.

Conversion of special characters to XML references:

Any remaining instances of the five characters & (ampersand), ' (apostrophe), > (greater-than sign), < (less-than sign), and " (quotation mark) are converted into the equivalent XML references '&';', ''';', '>';', '<';', and '"';', respectively.

Replacement of out-of-range Unicode characters:

Any remaining Unicode character that has a Unicode scalar value greater than x'FFFF' is replaced by an XML character reference. For example, if the document contains a character with Unicode scalar value x'10813', in UTF-16, that value is represented by the surrogate pair (NX'D802', NX'DC13'), which is replaced by the reference '𐠓'. For a document encoding of UTF-8, the byte sequence that is equivalent to character reference '𐠓' is X'F090A093'.

Trimming of generated XML data

Trimming is performed on data values after their conversion to character format. (Conversion is described under "Format conversion of elementary data" on page 477.)

For values converted from signed numeric values, the leading space is removed if the value is positive.

For values converted from numeric items, leading zeroes (after any initial minus sign) up to but not including the digit immediately before the actual or implied decimal point are eliminated. Trailing zeroes after a decimal point are retained. For example:

- -012.340 becomes -12.340.
- 0000.45 becomes 0.45.
- 0013 becomes 13.
- 0000 becomes 0.

Character values from data items of class alphabetic, alphanumeric, DBCS, and national have either trailing or leading spaces removed, depending on whether the corresponding data items have left (default) or right justification, respectively. That is, trailing spaces are removed from values whose corresponding data items do not specify the JUSTIFIED clause. Leading spaces are removed from values whose data items do specify the JUSTIFIED clause. If a character value consists solely of spaces, one space remains as the value after trimming is finished.

XML element name and attribute name formation

In the XML documents that are generated from *identifier-2*, the XML element names and attribute names are derived from the names of the data item specified by *identifier-2* and from any eligible data-names that are subordinate to *identifier-2* as follows:

- The exact mixed-case spelling of data-names from the data description entry is retained. The spellings from any references to data items (for example, in an OCCURS DEPENDING ON clause) are not used.
- Data-names that start with a digit are prefixed by an underscore. For example, the data-name '3D' becomes XML tag or attribute name '_3D'.
- Data-names that start with the characters 'xml', in any combination of uppercase and lowercase, are prefixed by an underscore. For example, the data-name 'Xml' becomes XML tag or attribute name '_Xml'.

Multibyte data-names, when translated to Unicode, must be legal as names in the XML specification, version 1.0. For details about the XML specification, see *XML specification*.

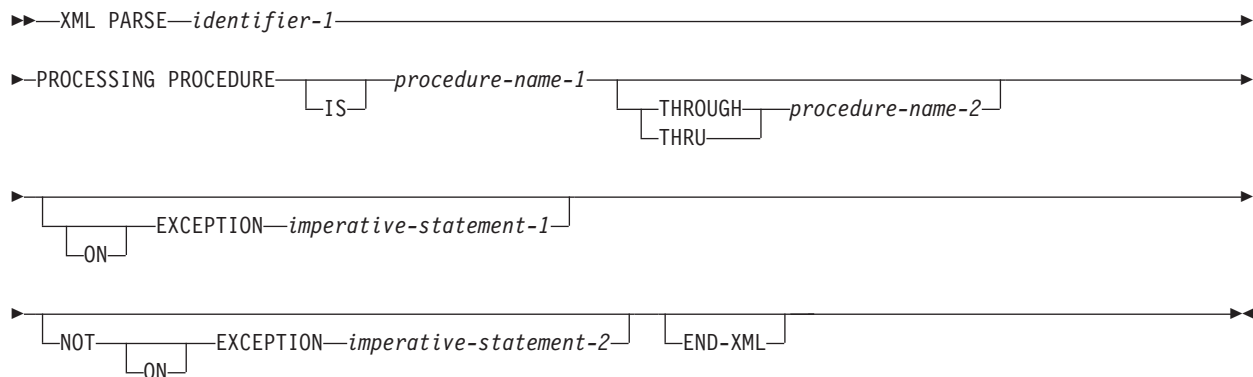
XML PARSE statement

The XML PARSE statement is the COBOL language interface to the high-speed XML parser that is part of the COBOL run time.

The XML PARSE statement parses an XML document into its individual pieces and passes each piece, one at a time, to a user-written processing procedure.

XML PARSE statements must not be specified in declarative procedures.

Format



identifier-1

identifier-1 must be an elementary data item of category national, a national group, an elementary data item of category alphanumeric, or an alphanumeric group item. *identifier-1* cannot be a function-identifier. *identifier-1* contains the XML document character stream.

If *identifier-1* is a national group item, *identifier-1* is processed as an elementary data item of category national.

If *identifier-1* is of category national, its content must be encoded using Unicode UTF-16BE (CCSID 1200). *identifier-1* must not contain any character entities that are represented using multiple encoding units. Use a character reference to represent any such characters, for example:

- "𐠓"
- "𐠓"

The letter x must be lowercase.

CHAR(NATIVE) and alphanumeric identifier-1

If *identifier-1* is alphanumeric and the CHAR(EBCDIC) compiler option is not in effect, the content of *identifier-1* must be encoded using UTF-8 Unicode or a single-byte ASCII code page that is supported by ICU conversion libraries (see *International Components for Unicode: Converter Explorer*).

To ensure that a document in such a data item is parsed in UTF-8 rather than ASCII:

- The code page for the runtime locale must be a UTF-8 locale, or
- The document must include an XML encoding declaration specifying UTF-8, or
- The document must start with a UTF-8 byte order mark.

UTF-8 documents must not contain any characters with a Unicode scalar value greater than x'FFFF'. Use a character reference for such characters.

If the XML document in such a data item does not specify an encoding declaration and does not start with a UTF-8 byte order mark, it is parsed with the code page indicated by the current runtime locale.

CHAR(EBCDIC) and alphanumeric identifier-1

If *identifier-1* is alphanumeric and the CHAR(EBCDIC) compiler option is in effect, the content of *identifier-1* must be encoded using a single-byte EBCDIC code page that is supported by ICU conversion libraries (see *International Components for Unicode: Converter Explorer*). If *identifier-1* is an elementary item, the NATIVE keyword must not be specified in its data description entry.

If the XML document in such a data item does not specify an encoding declaration, the XML document is parsed with the code page specified by the EBCDIC_CODEPAGE environment variable, or if the EBCDIC_CODEPAGE environment variable is not set, the default EBCDIC code page selected for the current runtime locale, as described in *Locales and code pages that are supported in the COBOL for AIX Programming Guide*.

Setting and using runtime locales and code pages

For more information about setting and using runtime locales and code pages, see *Locales and code pages that are supported in the COBOL for AIX Programming Guide*. The single-byte ASCII and EBCDIC code pages are those for which the column labeled Language group (the rightmost column) of the table *Locales and code pages supported* does not specify "Ideographic languages."

PROCESSING PROCEDURE phrase

Specifies the name of a procedure to handle the various events that the XML parser generates.

procedure-name-1, procedure-name-2

Must name a section or paragraph in the procedure division.

Procedure-name-1 and *procedure-name-2* must not name a procedure name in a declarative section.

procedure-name-1

Specifies the first (or only) section or paragraph in the processing procedure.

procedure-name-2

Specifies the last section or paragraph in the processing procedure.

For each XML event, the parser transfers control to the first statement of the procedure named *procedure-name-1*. Control is always returned from the processing procedure to the XML parser. The point from which control is returned is determined as follows:

- If *procedure-name-1* is a paragraph name and *procedure-name-2* is not specified, the return is made after the execution of the last statement of the *procedure-name-1* paragraph.

- If *procedure-name-1* is a section name and *procedure-name-2* is not specified, the return is made after the execution of the last statement of the last paragraph in the *procedure-name-1* section.
- If *procedure-name-2* is specified and it is a paragraph name, the return is made after the execution of the last statement of the *procedure-name-2* paragraph.
- If *procedure-name-2* is specified and it is a section name, the return is made after the execution of the last statement of the last paragraph in the *procedure-name-2* section.

The only necessary relationship between *procedure-name-1* and *procedure-name-2* is that they define a consecutive sequence of operations to execute, beginning at the procedure named by *procedure-name-1* and ending with the execution of the procedure named by *procedure-name-2*.

If there are two or more logical paths to the return point, then *procedure-name-2* can name a paragraph that consists of only an EXIT statement; all the paths to the return point must then lead to this paragraph.

The processing procedure consists of all the statements at which XML events are handled. The range of the processing procedure includes all statements executed by CALL, EXIT, GO TO, GOBACK, INVOKE, MERGE, PERFORM, and SORT statements that are in the range of the processing procedure, as well as all statements in declarative procedures that are executed as a result of the execution of statements in the range of the processing procedure.

The range of the processing procedure must not cause the execution of any GOBACK or EXIT PROGRAM statement, except to return control from a method or program to which control was passed by an INVOKE or CALL statement, respectively, that is executed in the range of the processing procedure.

The range of the processing procedure must not cause the execution of an XML PARSE statement, unless the XML PARSE statement is executed in a method or outermost program to which control was passed by an INVOKE or CALL statement that is executed in the range of the processing procedure.

A program executing on multiple threads can execute the same XML statement or different XML statements simultaneously.

The processing procedure can terminate the run unit with a STOP RUN statement.

For more details about the processing procedure, see “Control flow” on page 484.

ON EXCEPTION

The ON EXCEPTION phrase specifies imperative statements that are executed when the XML PARSE statement raises an exception condition.

An exception condition exists when the XML parser detects an error in processing the XML document. The parser first signals an XML exception by passing control to the processing procedure with special register XML-EVENT containing 'EXCEPTION'. The parser also provides a numeric error code in special register XML-CODE, as detailed in *Handling XML PARSE exceptions* in the *COBOL for AIX Programming Guide*.

An exception condition also exists if the processing procedure sets XML-CODE to -1 before returning to the parser for any normal XML event. In this case, the parser does not signal an EXCEPTION XML event and parsing is terminated.

If the ON EXCEPTION phrase is specified, the parser transfers control to *imperative-statement-1*. If the ON EXCEPTION phrase is not specified, the NOT ON EXCEPTION phrase, if any, is ignored and control is transferred to the end of the XML PARSE statement.

Special register XML-CODE contains the numeric error code for the XML exception or -1 after execution of the XML PARSE statement.

If the processing procedure handles the XML exception event and sets XML-CODE to zero before returning control to the parser, the exception condition no longer exists. If no other unhandled exceptions occur before termination of the parser, control is transferred to *imperative-statement-2* of the NOT ON EXCEPTION phrase, if specified.

NOT ON EXCEPTION

The NOT ON EXCEPTION phrase specifies imperative statements that are executed when no exception condition exists at the termination of XML PARSE processing.

If an exception condition does not exist at termination of XML PARSE processing, control is transferred to *imperative-statement-2* of the NOT ON EXCEPTION phrase, if specified. If the NOT ON EXCEPTION phrase is not specified, control is transferred to the end of the XML PARSE statement. The ON EXCEPTION phrase, if specified, is ignored.

Special register XML-CODE contains zero after execution of the XML PARSE statement.

END-XML phrase

This explicit scope terminator delimits the scope of XML GENERATE or XML PARSE statements. END-XML permits a conditional XML GENERATE or XML PARSE statement (that is, an XML GENERATE or XML PARSE statement that specifies the ON EXCEPTION or NOT ON EXCEPTION phrase) to be nested in another conditional statement.

The scope of a conditional XML GENERATE or XML PARSE statement can be terminated by:

- An END-XML phrase at the same level of nesting
- A separator period

END-XML can also be used with an XML GENERATE or XML PARSE statement that does not specify either the ON EXCEPTION or NOT ON EXCEPTION phrase.

For more information about explicit scope terminators, see “Delimited scope statements” on page 289.

Nested XML GENERATE or XML PARSE statements

When a given XML GENERATE or XML PARSE statement appears as *imperative-statement-1* or *imperative-statement-2*, or as part of *imperative-statement-1* or *imperative-statement-2* of another XML GENERATE or XML PARSE statement, that given XML GENERATE or XML PARSE statement is a *nested* XML GENERATE or XML PARSE statement.

Nested XML GENERATE or XML PARSE statements are considered to be matched XML GENERATE and END-XML, or XML PARSE and END-XML combinations proceeding from left to right. Thus, any END-XML phrase that is encountered is matched with the nearest preceding XML GENERATE or XML PARSE statement that has not been implicitly or explicitly terminated.

Control flow

When the XML parser receives control from an XML PARSE statement, the parser analyzes the XML document and transfers control at the following points in the process:

- The start of the parsing process
- When a document fragment is found
- When the parser detects an error in parsing the XML document
- The end of processing the XML document

Control returns to the XML parser when the end of the processing procedure is reached.

The exchange of control between the parser and the processing procedure continues until either:

- The entire XML document has been parsed, ending with the END-OF-DOCUMENT event.
- The processing procedure terminates parsing deliberately by setting XML-CODE to -1 before returning to the parser.
- The parser detects an exception (other than an encoding conflict) and the processing procedure does not reset special register XML-CODE to zero before returning to the parser.
- The parser detects an encoding conflict exception and the processing procedure does not reset special register XML-CODE to zero or to the CCSID of the document encoding.

In each case, the processing procedure returns control to the parser. Then, the parser terminates and returns control to the XML PARSE statement with the XML-CODE special register containing the most recent value set by the parser or -1 (which might have been set by the parser or by the processing procedure).

For each XML event passed to the processing procedure, the XML-CODE and XML-EVENT special registers contain information about the particular event. Special register XML-EVENT is set to the event name, such as 'START-OF-DOCUMENT'. For most events, the XML-TEXT or XML-NTEXT special register contains document text. See "XML-EVENT" on page 26 for details.

The content of the XML-CODE special register is defined during and after execution of an XML PARSE statement. The contents of all other XML special registers are undefined outside the range of the processing procedure.

For normal XML events, special register XML-CODE contains zero when the processing procedure receives control. For XML exception events, XML-CODE contains an XML exception code as described in *XML exceptions that allow continuation* and *XML exceptions that do not allow continuation* in the *COBOL for AIX Programming Guide*.

For more information about the XML special registers, see:

- “XML-CODE” on page 25
- “XML-EVENT” on page 26
- “XML-NTEXT” on page 27
- “XML-TEXT” on page 28

For an introduction to special registers, see “Special registers” on page 16

For more information about the EXCEPTION event and exception processing, see *Handling XML PARSE exceptions* in the *COBOL for AIX Programming Guide*.

Part 7. Intrinsic functions

Chapter 22. Intrinsic functions	489	WHEN-COMPILED	531
Specifying a function	489	YEAR-TO-YYYY	532
Function definition and evaluation	490	YEARWINDOW	533
Types of functions	490		
Rules for usage	491		
Arguments	492		
Examples	494		
ALL subscripting	494		
Function definitions	496		
ACOS	500		
ANNUITY	500		
ASIN	501		
ATAN	501		
CHAR	501		
COS	502		
CURRENT-DATE	502		
DATE-OF-INTEGER	504		
DATE-TO-YYYYMMDD	504		
DATEVAL	505		
DAY-OF-INTEGER	506		
DAY-TO-YYYYDDD	507		
DISPLAY-OF	507		
FACTORIAL	509		
INTEGER	509		
INTEGER-OF-DATE	510		
INTEGER-OF-DAY	510		
INTEGER-PART	511		
LENGTH	511		
LOG	513		
LOG10	513		
LOWER-CASE	513		
MAX	514		
MEAN	515		
MEDIAN	516		
MIDRANGE	516		
MIN	517		
MOD	518		
NATIONAL-OF	518		
NUMVAL	519		
NUMVAL-C	520		
ORD	522		
ORD-MAX	523		
ORD-MIN	523		
PRESENT-VALUE	524		
RANDOM	524		
RANGE	525		
REM	526		
REVERSE	526		
SIN	527		
SQRT	527		
STANDARD-DEVIATION	528		
SUM	528		
TAN	529		
UNDATE	529		
UPPER-CASE	530		
VARIANCE	530		

Chapter 22. Intrinsic functions

Data processing problems often require the use of values that are not directly accessible in the data storage associated with the object program, but instead must be derived through performing operations on other data. An *intrinsic function* is a function that performs a mathematical, character, or logical operation, and thereby allows you to make reference to a data item whose value is derived automatically during execution.

The functions can be grouped into six categories, based on the type of service performed:

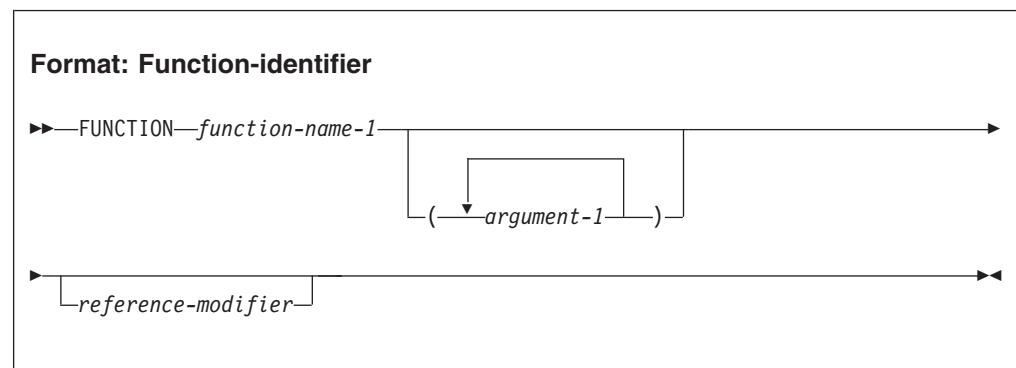
- Mathematical
- Statistical
- Date/time
- Financial
- Character-handling
- General

You can reference a function by specifying its name, along with any required arguments, in a procedure division statement.

Functions are elementary data items, and return alphanumeric character, national character, numeric, or integer values. Functions cannot serve as receiving operands.

Specifying a function

The general format of a function-identifier is:



function-name-1

function-name-1 must be one of the intrinsic function names.

argument-1

argument-1 must be an identifier, a literal (other than a figurative constant), or an arithmetic expression that satisfies the argument requirements for the specified function.

argument-1 cannot be a windowed date field, except in the UNDATE intrinsic function.

reference-modifier

Can be specified only for functions of type alphanumeric or national.

A function-identifier can be specified wherever a data item of the type of the function is allowed. The argument to a function can be any function or an expression containing a function, including another evaluation of the same function, whose result meets the requirements for the argument.

Within a procedure division statement, each function-identifier is evaluated at the same time as any reference modification or subscripting associated with an identifier in that same position would be evaluated.

Function definition and evaluation

The class and characteristics of a function, and the number and types of arguments it requires, are determined by its function definition. These characteristics include:

- For functions of type alphanumeric and national, the size of the returned value
- For functions of type numeric and integer, the sign of the returned value and whether the function is integer
- The actual value returned by the function

For some functions, the class and characteristics are determined by the arguments to the function.

The evaluation of any intrinsic function is not affected by the context in which it appears; in other words, function evaluation is not affected by operations or operands outside the function. However, evaluation of a function can be affected by the attributes of its arguments.

Within a procedure division statement, each function-identifier is evaluated at the same time as any reference modification or subscripting associated with an identifier in that same position would be evaluated.

Types of functions

COBOL has the following types of functions:

- Alphanumeric
- National
- Numeric
- Integer

Alphanumeric functions are of class and category alphanumeric. The value returned has an implicit usage of DISPLAY without the NATIVE phrase. The number of character positions in the value returned is determined by the function definition.

National functions are of class and category national. The value returned has an implicit usage of NATIONAL and is represented in national characters (UTF-16). The number of character positions in the value returned is determined by the function definition.

Numeric functions are of class and category numeric. The returned value is always considered to have an operational sign and is a numeric intermediate result. For more information, see *Using numeric intrinsic functions* in the *COBOL for AIX Programming Guide*.

Integer functions are of class and category numeric. The returned value is always considered to have an operational sign and is an integer intermediate result. The number of digit positions in the value returned is determined by the function definition. For more information, see *Using numeric intrinsic functions* in the *COBOL for AIX Programming Guide*.

Rules for usage

Alphanumeric functions

An alphanumeric function can be specified anywhere in the general formats that a data item of class and category alphanumeric is permitted and where the rules associated with the general formats do not specifically prohibit reference to functions, except as noted below.

An alphanumeric function can be used as an argument for any function that allows an alphanumeric argument.

Reference modification of an alphanumeric function is allowed. If reference modification is specified for a function, the evaluation of the reference modification takes place immediately after the evaluation of the function; that is, the function's returned value is reference-modified.

An alphanumeric function cannot be used:

- As a receiving operand of any statement
- Where the rules associated with the general formats require the data item being referenced to have particular characteristics (such as class and category, usage, size, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have those characteristics

National functions

A national function can be specified anywhere in the general formats that a data item of class and category national is permitted and where the rules associated with the general formats do not specifically prohibit reference to functions, except as noted below.

A national function can be used as an argument for any function that allows a national argument.

Reference modification of a national function is allowed. If reference modification is specified for a function, the evaluation of the reference modification takes place immediately after the evaluation of the function; that is, the function's returned value is reference-modified.

A national function cannot be used:

- As a receiving operand of any statement
- Where the rules associated with the general formats require the data item being referenced to have particular characteristics (such as class and category, usage, size, and permissible values) and the evaluation of the function according to its definition and the particular arguments specified would not have those characteristics

Numeric functions

A numeric function can be used only where an arithmetic expression can be specified.

A numeric function can be referenced as an argument for a function that allows a numeric argument.

A numeric function cannot be used where an integer operand is required, even if the particular reference would yield an integer value. The `INTEGER` or `INTEGER-PART` functions can be used to force the type of a numeric argument to be an integer.

Integer functions

An integer function can be used only where an arithmetic expression can be specified.

An integer function can be referenced as an argument for a function that allows an integer argument.

Usage notes:

- *identifier-2* of the `CALL` statement must not be a function-identifier.
- The `COPY` statement accepts function-identifiers of all types in the `REPLACING` phrase.

Arguments

The value returned by some functions is determined by the arguments specified in the function-identifier when the functions are evaluated. Some functions require no arguments; others require a fixed number of arguments, and still others accept a variable number of arguments.

An argument must be one of the following:

- A data item identifier
- An arithmetic expression
- A function-identifier
- A literal other than a figurative constant
- A special-register

See “Function definitions” on page 496 for function-specific argument specifications.

The types of arguments are:

Alphabetic

An elementary data item of the class alphabetic or an alphanumeric literal containing only alphabetic characters. The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

Alphanumeric

A data item of the class alphabetic or alphanumeric or an alphanumeric literal. The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

DBCS An elementary data item of class DBCS or a DBCS literal. The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function. (A DBCS data item or literal can be used as an argument only for the `NATIONAL-OF` function.)

National

A data item of class national (category national, national-edited, or

numeric-edited). The content of the argument is used to determine the value of the function. The length of the argument can be used to determine the value of the function.

Integer

An arithmetic expression that always results in an integer value. The value of the expression, including its sign, is used to determine the value of the function.

Numeric

An arithmetic expression. The expression can include numeric literals and data items of categories numeric, internal floating-point, and external floating-point. The numeric data items can have any usage permitted for the category of the data item (including NATIONAL). The value of the expression, including its sign, is used to determine the value of the function.

Some functions place constraints on their arguments, such as the acceptable range of values. If the values assigned as arguments for a function do not comply with specified constraints, the returned value is undefined.

If a nested function is used as an argument, the evaluation of its arguments is not affected by the arguments in the outer function.

Only those arguments at the same function level interact with each other. This interaction occurs in two areas:

- The computation of an arithmetic expression that appears as a function argument is affected by other arguments for that function.
- The evaluation of the function takes into consideration the attributes of all of its arguments.

When a function is evaluated, its arguments are evaluated individually in the order specified in the list of arguments, from left to right. The argument being evaluated can be a function-identifier or an expression that includes function-identifiers.

If an arithmetic expression is specified as an argument and if the first operator in the expression is a unary plus or a unary minus, the expression must be immediately preceded by a left parenthesis.

Floating-point literals are allowed wherever a numeric argument is allowed and in arithmetic expressions used in functions that allow a numeric argument.

Internal floating-point items and external floating-point items (both display floating-point and national floating-point) can be used wherever a numeric argument is allowed and in arithmetic expressions as arguments to a function that allows a numeric argument.

Floating-point items and floating-point literals *cannot* be used where an integer argument is required or where an argument of class alphanumeric or national is required (such as in the LOWER-CASE, REVERSE, UPPER-CASE, NUMVAL, and NUMVAL-C functions).

Examples

The following statement illustrates the use of intrinsic function UPPER-CASE to replace each lowercase letter in an alphanumeric argument with the corresponding uppercase letter.

```
MOVE FUNCTION UPPER-CASE('hello') TO DATA-NAME.
```

This statement moves HELLO into DATA-NAME.

The following statement illustrates the use of intrinsic function LOWER-CASE to replace each uppercase letter in a national argument with the corresponding lowercase letter.

```
MOVE FUNCTION LOWER-CASE(N'HELLO') TO N-DATA-NAME.
```

This statement moves national characters hello into N-DATA-NAME.

The following statement illustrates the use of a numeric intrinsic function:

```
COMPUTE NUM-ITEM = FUNCTION SUM(A B C)
```

This statement uses the numeric function SUM to add the values of A, B, and C and places the result in NUM-ITEM.

ALL subscripting

When a function allows an argument to be repeated a variable number of times, you can refer to a table by specifying the data-name and any qualifiers that identify the table. This can be followed immediately by subscripting where one or more of the subscripts is the word ALL.

Tip: The evaluation of an ALL subscript must result in at least one argument or the value returned by the function will be undefined; however, the situation can be diagnosed at run time by specifying the SSRANGE compiler option and the CHECK runtime option.

Specifying ALL as a subscript is equivalent to specifying all table elements possible using every valid subscript in that subscript position.

For a table argument specified as Table-name(ALL), the order of the implicit specification of each table element as an argument is from left to right, where the first (or leftmost) argument is Table-name(1) and ALL has been replaced by 1. The next argument is Table-name(2), where the subscript has been incremented by 1. This process continues, with the subscript being incremented by 1 to produce an implicit argument, until the ALL subscript has been incremented through its range of values.

For example,

```
FUNCTION MAX(Table(ALL))
```

is equivalent to

```
FUNCTION MAX(Table(1) Table(2) Table(3) ... Table(n))
```

where *n* is the number of elements in Table.

If there are multiple ALL subscripts, Table-name(ALL, ALL, ALL), the first implicit argument is Table-name(1, 1, 1), where each ALL has been replaced by 1. The next argument is Table-name(1, 1, 2), where the rightmost subscript has been incremented by 1. The subscript represented by the rightmost ALL is incremented through its range of values to produce an implicit argument for each value.

Once a subscript specified as ALL has been incremented through its range of values, the next subscript to the left that is specified as ALL is incremented by 1. Each subscript specified as ALL to the right of the newly incremented subscript is set to 1 to produce an implicit argument. Once again, the subscript represented by the rightmost ALL is incremented through its range of values to produce an implicit argument for each value. This process is repeated until each subscript specified as ALL has been incremented through its range of values.

For example,

```
FUNCTION MAX(Table(ALL, ALL))
```

is equivalent to

```
FUNCTION MAX(Table(1, 1) Table(1, 2) Table(1, 3) ... Table(1, n)
              Table(2, 1) Table(2, 2) Table(2, 3) ... Table(2, n)
              Table(3, 1) Table(3, 2) Table(3, 3) ... Table(3, n)
              ...
              Table(m, 1) Table(m, 2) Table(m, 3) ... Table(m, n))
```

where n is the number of elements in the column dimension of Table, and m is the number of elements in the row dimension of Table.

ALL subscripts can be combined with literal, data-name, or index-name subscripts to reference multidimensional tables.

For example,

```
FUNCTION MAX(Table(ALL, 2))
```

is equivalent to

```
FUNCTION MAX(Table(1, 2)
              Table(2, 2)
              Table(3, 2)
              ...
              Table(m, 2))
```

where m is the number of elements in the row dimension of Table.

If an ALL subscript is specified for an argument and the argument is reference-modified, then the reference-modifier is applied to each of the implicitly specified elements of the table.

If an ALL subscript is specified for an operand that is reference-modified, the reference-modifier is applied to each of the implicitly specified elements of the table.

If the ALL subscript is associated with an OCCURS DEPENDING ON clause, the range of values is determined by the object of the OCCURS DEPENDING ON clause.

For example, given a payroll record definition such as:

```

01 PAYROLL.
  02 PAYROLL-WEEK    PIC 99.
  02 PAYROLL-HOURS   PIC 999 OCCURS 1 TO 52
    DEPENDING ON PAYROLL-WEEK.

```

The following COMPUTE statements could be used to identify total year-to-date hours, the maximum hours worked in any week, and the specific week corresponding to the maximum hours:

```

COMPUTE YTD-HOURS = FUNCTION SUM (PAYROLL-HOURS(ALL))
COMPUTE MAX-HOURS = FUNCTION MAX (PAYROLL-HOURS(ALL))
COMPUTE MAX-WEEK  = FUNCTION ORD-MAX (PAYROLL-HOURS(ALL))

```

In these function invocations, the subscript ALL is used to reference all elements of the PAYROLL-HOURS array (depending on the execution time value of the PAYROLL-WEEK field).

Function definitions

Table 58 on page 497 provides an overview of the argument type, function type, and value returned for each of the intrinsic functions. Argument types and function types are abbreviated as follows:

Abbreviation	Meaning
A	Alphabetic
D	DBCS
I	Integer
N	Numeric
X	Alphanumeric
U	National
O	Other, as specified in the function definition (pointer, function-pointer, procedure-pointer, or object reference)

The behavior of functions marked "DP" depends on whether the DATEPROC or NODATEPROC compiler option is in effect.

If the DATEPROC compiler option is in effect, the following intrinsic functions return date fields:

Function	Returned value has implicit DATE FORMAT
DATE-OF-INTEGERS	YYYYXXXX
DATE-TO-YYYYMMDD	YYYYXXXX
DAY-OF-INTEGERS	YYYYXXXX
DAY-TO-YYYYDDD	YYYYXXXX
YEAR-TO-YYYY	YYYY
DATEVAL	Depends on the format specified by DATEVAL
YEARWINDOW	YYYY

If the NODATEPROC compiler option is in effect:

- The following intrinsic functions return the same values as when DATEPROC is in effect, but their returned values are nondates:

- DAY-OF-INTEGER
- DATE-TO-YYYYMMDD
- DAY-TO-YYYYDDD
- YEAR-TO-YYYY
- The DATEVAL and UNDATE intrinsic functions have no effect, and simply return their (first) arguments unchanged.
- The YEARWINDOW intrinsic function returns 0 unconditionally.

Each intrinsic function is described in detail in the topics that follow the table below.

Table 58. Table of functions

Function name	Arguments	Function type	Value returned
ACOS	N1	N	Arccosine of N1
ANNUITY	N1, I2	N	Ratio of annuity paid for I2 periods at interest of N1 to initial investment of one
ASIN	N1	N	Arcsine of N1
ATAN	N1	N	Arctangent of N1
CHAR	I1	X	Character in position I1 of program collating sequence
COS	N1	N	Cosine of N1
CURRENT-DATE	None	X	Current date and time and difference from Greenwich mean time
DATE-OF-INTEGER ^{DP}	I1	I	Standard date equivalent (YYYYMMDD) of integer date
DATE-TO-YYYYMMDD ^{DP}	I1, I2	I	Standard date equivalent (YYYYMMDD) of I1 (standard date with a windowed year, YYYYMMDD), according to the 100-year interval whose ending year is specified by the sum of I2 and the year at execution time
DATEVAL ^{DP}	I1	I	Date field equivalent of I1
	X1	X	Date field equivalent of X1
DAY-OF-INTEGER ^{DP}	I1	I	Julian date equivalent (YYYYDDD) of integer date
DAY-TO-YYYYDDD ^{DP}	I1, I2	I	Julian date equivalent (YYYYDDD) of I1 (Julian date with a windowed year, YYDDD), according to the 100-year interval whose ending year is specified by the sum of I2 and the year at execution time
DISPLAY-OF	U1 or U1, I2	X	Each character in U1 converted to a corresponding character representation using a code page identified by I2, if specified, or the code page specified by the runtime locale if I2 is unspecified
FACTORIAL	I1	I	Factorial of I1
INTEGER	N1	I	The greatest integer not greater than N1
INTEGER-OF-DATE	I1	I	Integer date equivalent of standard date (YYYYMMDD)

Table 58. Table of functions (continued)

Function name	Arguments	Function type	Value returned
INTEGER-OF-DAY	I1	I	Integer date equivalent of Julian date (YYYYDDD)
INTEGER-PART	N1	I	Integer part of N1
LENGTH	A1, N1, O1, X1, or U1	I	Length of argument in national character positions or in alphanumeric character positions or bytes, depending on the argument type
LOG	N1	N	Natural logarithm of N1
LOG10	N1	N	Logarithm to base 10 of N1
LOWER-CASE	A1 or X1	X	All letters in the argument set to lowercase
	U1	U	All letters in the argument set to lowercase
MAX	A1...	X	Value of maximum argument; note that the type of function depends on the arguments
	I1...	I	Value of maximum argument; note that the type of function depends on the arguments
	N1...	N	Value of maximum argument; note that the type of function depends on the arguments
	X1...	X	Value of maximum argument; note that the type of function depends on the arguments
	U1...	U	Value of maximum argument; note that the type of function depends on the arguments
MEAN	N1...	N	Arithmetic mean of arguments
MEDIAN	N1...	N	Median of arguments
MIDRANGE	N1...	N	Mean of minimum and maximum arguments
MIN	A1...	X	Value of minimum argument; note that the type of function depends on the arguments
	I1...	I	Value of minimum argument; note that the type of function depends on the arguments
	N1...	N	Value of minimum argument; note that the type of function depends on the arguments
	X1...	X	Value of minimum argument; note that the type of function depends on the arguments
	U1...	U	Value of minimum argument; note that the type of function depends on the arguments
MOD	I1, I2	I	I1 modulo I2
NATIONAL-OF	A1, X1, or D1	U	The characters in the argument converted to national characters, using the code page identified by I2, if specified, or a code page specified by the runtime locale if I2 is unspecified
	A1, X1, or D1; I2	U	The characters in the argument converted to national characters, using the code page identified by I2, if specified, or a code page specified by the runtime locale if I2 is unspecified
NUMVAL	X1	N	Numeric value of simple numeric string

Table 58. Table of functions (continued)

Function name	Arguments	Function type	Value returned
NUMVAL-C	X1 or X1, X2	N	Numeric value of numeric string with optional commas and currency sign
ORD	A1 or X1	I	Ordinal position of the argument in collating sequence
ORD-MAX	A1..., N1..., X1..., or U1...	I	Ordinal position of maximum argument
ORD-MIN	A1..., N1..., X1..., or U1...	I	Ordinal position of minimum argument
PRESENT-VALUE	N1, N2...	N	Present value of a series of future period-end amounts, N2, at a discount rate of N1
RANDOM	I1, none	N	Random number
RANGE	I1...	I	Value of maximum argument minus value of minimum argument; note that the type of function depends on the arguments.
	N1...	N	Value of maximum argument minus value of minimum argument; note that the type of function depends on the arguments.
REM	N1, N2	N	Remainder of N1/N2
REVERSE	A1 or X1	X	Reverse order of the characters of the argument
	U1	U	Reverse order of the characters of the argument
SIN	N1	N	Sine of N1
SQRT	N1	N	Square root of N1
STANDARD-DEVIATION	N1...	N	Standard deviation of arguments
SUM	I1...	I	Sum of arguments; note that the type of function depends on the arguments.
	N1...	N	Sum of arguments; note that the type of function depends on the arguments.
TAN	N1	N	Tangent of N1
UNDATE ^{DP}	I1	I	Nondate equivalent of date field I1 or X1
	X1	X	Nondate equivalent of date field I1 or X1
UPPER-CASE	A1 or X1	X	All letters in the argument set to uppercase
	U1	U	All letters in the argument set to uppercase
VARIANCE	N1...	N	Variance of arguments
WHEN-COMPILED	None	X	Date and time when program was compiled
YEAR-TO-YYYY ^{DP}	I1, I2	I	Expanded year equivalent (YYYY) of I1 (windowed year, YY), according to the 100-year interval whose ending year is specified by the sum of I2 and the year at execution time
YEARWINDOW ^{DP}	None	I	If the DATEPROC compiler option is in effect, returns the starting year (in the format YYYY) of the century window specified by the YEARWINDOW compiler option; if NODATEPROC is in effect, returns 0

ACOS

The ACOS function returns a numeric value in radians that approximates the arccosine of the argument specified.

The function type is numeric.

Format

►►—FUNCTION ACOS—(—*argument-1*—)————►◄

argument-1

Must be class numeric. The value of *argument-1* must be greater than or equal to -1 and less than or equal to +1.

The returned value is the approximation of the arccosine of the argument and is greater than or equal to zero and less than or equal to Pi.

ANNUITY

The ANNUITY function returns a numeric value that approximates the ratio of an annuity paid at the end of each period, for a given number of periods, at a given interest rate, to an initial value of one. The number of periods is specified by *argument-2*; the rate of interest is specified by *argument-1*. For example, if *argument-1* is zero and *argument-2* is four, the value returned is the approximation of the ratio 1 / 4.

The function type is numeric.

Format

►►—FUNCTION ANNUITY—(—*argument-1*—*argument-2*—)————►◄

argument-1

Must be class numeric. The value of *argument-1* must be greater than or equal to zero.

argument-2

Must be a positive integer.

When the value of *argument-1* is zero, the value returned by the function is the approximation of:

$1 / \textit{argument-2}$

When the value of *argument-1* is not zero, the value of the function is the approximation of:

$$\text{argument-1} / (1 - (1 + \text{argument-1}) ** (- \text{argument-2}))$$

ASIN

The ASIN function returns a numeric value in radians that approximates the arcsine of the argument specified.

The function type is numeric.

Format

►►—FUNCTION ASIN—(—*argument-1*—)————►◄

argument-1

Must be class numeric. The value of *argument-1* must be greater than or equal to -1 and less than or equal to +1.

The returned value is the approximation of the arcsine of *argument-1* and is greater than or equal to -Pi/2 and less than or equal to +Pi/2.

ATAN

The ATAN function returns a numeric value in radians that approximates the arctangent of the argument specified.

The function type is numeric.

Format

►►—FUNCTION ATAN—(—*argument-1*—)————►◄

argument-1

Must be class numeric.

The returned value is the approximation of the arctangent of *argument-1* and is greater than -Pi/2 and less than +Pi/2.

CHAR

The CHAR function returns a one-character alphanumeric value that is a character in the program collating sequence having the ordinal position equal to the value of the argument specified.

The function type is alphanumeric.

Format

►►—FUNCTION CHAR—(—*argument-1*—)————►◄

argument-1

Must be an integer. The value must be greater than zero and less than or equal to the number of positions in the collating sequence associated with alphanumeric data items (a maximum of 256).

If more than one character has the same position in the program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.

If the current program collating sequence was not specified by an ALPHABET clause, the COLLSEQ compiler option indicates the collating sequence used. For example, if COLLSEQ(EBCDIC) is specified and the PROGRAM COLLATING SEQUENCE is not specified (or is NATIVE), the EBCDIC collating sequence is applied. (See “Conditional expressions” on page 262.)

COS

The COS function returns a numeric value that approximates the cosine of the angle or arc specified by the argument in radians.

The function type is numeric.

Format

►►—FUNCTION COS—(—*argument-1*—)————►◄

argument-1

Must be class numeric.

The returned value is the approximation of the cosine of the argument and is greater than or equal to -1 and less than or equal to +1.

CURRENT-DATE

The CURRENT-DATE function returns a 21-character alphanumeric value that represents the calendar date, time of day, and time differential from Greenwich mean time provided by the system on which the function is evaluated.

The function type is alphanumeric.

Format

►►—FUNCTION CURRENT-DATE—◄◄

Reading from left to right, the 21 character positions of the returned value are as follows:

Character positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar
5-6	Two numeric digits of the month of the year, in the range 01 through 12
7-8	Two numeric digits of the day of the month, in the range 01 through 31
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99. The value 00 is returned if the system on which the function is evaluated does not have the facility to provide the fractional part of a second.
17	Either the character '-' or the character '+'. The character '-' is returned if the local time indicated in the previous character positions is behind Greenwich mean time. The character '+' is returned if the local time indicated is the same as or ahead of Greenwich mean time. The character '0' is returned if the system on which this function is evaluated does not have the facility to provide the local time differential factor.
18-19	If character position 17 is '-', two numeric digits are returned in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich mean time. If character position 17 is '+', two numeric digits are returned in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich mean time. If character position 17 is '0', the value 00 is returned.
20-21	Two numeric digits are returned in the range 00 through 59 indicating the number of additional minutes that the reported time is ahead of or behind Greenwich mean time, depending on whether character position 17 is '+' or '-', respectively. If character position 17 is '0', the value 00 is returned.

For more information, see *Examples: numeric intrinsic functions* in the *COBOL for AIX Programming Guide*.

DATE-OF-INTEGER

The DATE-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to standard date form (YYYYMMDD).

The function type is integer.

The function result is an eight-digit integer.

If the DATEPROC compiler option is in effect, the returned value is an expanded date field with implicit DATE FORMAT YYYYXXXX.

Format

►►—FUNCTION DATE-OF-INTEGER—(—*argument-1*—)————►►

argument-1

A positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar. The valid range is 1 to 3,067,671, which corresponds to dates ranging from January 1, 1601 thru December 31, 9999.

The returned value represents the International Standards Organization (ISO) standard date equivalent to the integer specified as *argument-1*.

The returned value is an integer of the form YYYYMMDD where YYYY represents a year in the Gregorian calendar; MM represents the month of that year; and DD represents the day of that month.

DATE-TO-YYYYMMDD

The DATE-TO-YYYYMMDD function converts *argument-1* from a date with a two-digit year (YYnnnn) to a date with a four-digit year (YYYYnnnn). *argument-2*, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of *argument-1* falls.

The function type is integer.

If the DATEPROC compiler option is in effect, the returned value is an expanded date field with implicit DATE FORMAT YYYYXXXX.

Format

►►—FUNCTION DATE-TO-YYYYMMDD—(—*argument-1*—
 └─*argument-2*─┐)—————►►

argument-1

Must be zero or a positive integer less than 991232.

Note: The COBOL run time does not verify that the value is a valid date.

argument-2

Must be an integer. If *argument-2* is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of *argument-2* must be less than 10,000 and greater than 1,699.

The following are examples of returned values from the DATE-TO-YYYYMMDD function:

Current year	<i>argument-1</i> value	<i>argument-2</i> value	Returned value
2002	851003	120	20851003
2002	851003	-20	18851003
2002	851003	10	19851003
1994	981002	-10	18981002

DATEVAL

The DATEVAL function converts a nondate to a date field, for unambiguous use with date fields.

If the DATEPROC compiler option is in effect, the returned value is a date field containing the value of *argument-1* unchanged. For information about using the resulting date field:

- In arithmetic, see “Arithmetic with date fields” on page 260
- In conditional expressions, see “Comparison of date fields” on page 276

If the NODATEPROC compiler option is in effect, the DATEVAL function has no effect, and returns the value of *argument-1* unchanged.

The function type depends on the type of *argument-1*:

Argument type	Function type
Alphanumeric	Alphanumeric
Integer	Integer

Format

►►—FUNCTION DATEVAL—(—*argument-1*—*argument-2*—)————►►

argument-1

Must be one of the following:

- A class alphanumeric item with the same number of characters as the date format specified by *argument-2*.
- An integer. This can be used to specify values outside the range specified by *argument-2*, including negative values.

The value of *argument-1* represents a date of the form specified by *argument-2*.

argument-2

Must be an alphanumeric literal specifying a date pattern, as defined in “DATE FORMAT clause” on page 184. The date pattern consists of YY or YYYY (representing a windowed year or expanded year, respectively), optionally preceded or followed by one or more Xs (representing other parts of a date, such as month and day), as shown below. Note that the values are case insensitive; the letters X and Y in *argument-2* can be any mix of uppercase and lowercase.

Date-pattern string	Specifies that <i>argument-1</i> contains
YY	A windowed (two-digit) year
YYYY	An expanded (four-digit) year
X	A single character; for example, a digit representing a semester or quarter (1–4)
XX	Two characters; for example, digits representing a month (01–12)
XXX	Three characters; for example, digits representing a day of the year (001–366)
XXXX	Four characters; for example, two digits representing a month (01–12) and two digits representing a day of the month (01–31)

DAY-OF-INTEGER

The DAY-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to Julian date form (YYYYDDD).

The function type is integer.

The function result is a seven-digit integer.

If the DATEPROC compiler option is in effect, the returned value is an expanded date field with implicit DATE FORMAT YYYYXX.

Format

►►—FUNCTION DAY-OF-INTEGER—(—*argument-1*—)————►►

argument-1

A positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar. The valid range is 1 to 3,067,671, which corresponds to dates ranging from January 1, 1601 thru December 31, 9999.

The returned value represents the Julian equivalent of the integer specified as *argument-1*. The returned value is an integer of the form YYYYDDD where YYYY represents a year in the Gregorian calendar and DDD represents the day of that year.

DAY-TO-YYYYDDD

The DAY-TO-YYYYDDD function converts *argument-1* from a date with a two-digit year (YYnnn) to a date with a four-digit year (YYYYnnn). *argument-2*, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of *argument-1* falls.

The function type is integer.

If the DATEPROC compiler option is in effect, the returned value is an expanded date field with implicit DATE FORMAT YYYYXX.

Format

►►FUNCTION DAY-TO-YYYYDDD(—*argument-1*—

└*argument-2*┐

)—►►

- argument-1*
Must be zero or a positive integer less than 99367.
The COBOL run time does not verify that the value is a valid date.
- argument-2*
Must be an integer. If *argument-2* is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of *argument-2* must be less than 10,000 and greater than 1,699.

Some examples of returned values from the DAY-TO-YYYYDDD function follow:

Current year	<i>argument-1</i> value	<i>argument-2</i> value	Returned value
2002	10004	-20	1910004
2002	10004	-120	1810004
2002	10004	20	2010004
2013	95005	-10	1995005

DISPLAY-OF

The DISPLAY-OF function returns an alphanumeric character string consisting of the content of *argument-1* converted to a specific code page representation.

The type of the function is alphanumeric.

Format

►►FUNCTION DISPLAY-OF(—*argument-1*—*argument-2*)—►►

argument-1

Must be of class national (categories national, national-edited, and numeric-edited described with usage NATIONAL). *argument-1* identifies the source string for the conversion.

argument-2

Must be an integer or of class alphanumeric. *argument-2* identifies the output code page for the conversion.

If *argument-2* is of class alphanumeric, it must identify a primary or alias code-page name that is supported by ICU conversion libraries (see *International Components for Unicode: Converter Explorer*).

If *argument-2* is an integer, the integer must be a valid CCSID number.

If *argument-2* is omitted, the output code page is determined from the runtime locale.

The returned value is an alphanumeric character string consisting of the characters of *argument-1* converted to the output code page representation. When a source character cannot be converted to a character in the output code page, the source character is replaced with a substitution character. The following table shows substitution characters for some widely-used code pages:

Output code page	Substitution character
SBCS ASCII PC Windows SBCS	X'7F'
ISO SBCS	X'1A'
EBCDIC SBCS	X'3F'
ASCII DBCS	X'FCFC'
EBCDIC DBCS (except for Thai)	X'FEFE'
EBCDIC DBCS (Thai)	X'41B8'
PC DBCS (Japanese or Chinese)	X'FCFC'
PC DBCS (Korean)	X'BFFC'
EUC (Korean)	X'AFFE'
EUC (Japanese)	X'747E'
UTF-8	From SBCS: X'1A' From MBCS: X'EFBFD'
UTF-16	From SBCS: X'001A' From MBCS: X'FFFD'

No exception condition is raised.

The length of the returned value depends on the content of *argument-1* and the characteristics of the output code page.

Usage notes

- Use of code page names provides consistency with other AIX software, but the source code is not portable to Enterprise COBOL for z/OS.
- The CCSID for UTF-8 is 1208.
- If the output code page includes DBCS characters, the returned value can be a mixed SBCS and DBCS string.
- The DISPLAY-OF function, with *argument-2* specified, can be used to generate character data represented in a code page that differs from the code page in effect for ASCII or EBCDIC data. Subsequent COBOL operations on that data can involve implicit conversions that assume the data is represented in the ASCII or EBCDIC code page in effect. See *Converting to or from national (Unicode) representation* in the *COBOL for AIX Programming Guide* for examples and programming techniques for processing data represented using more than one code page within a single program.

Exception: If the conversion fails, a severe runtime error occurs.

FACTORIAL

The FACTORIAL function returns an integer that is the factorial of the argument specified.

The function type is integer.

Format

►►—FUNCTION FACTORIAL—(—*argument-1*—)————►►

argument-1

If the ARITH(COMPAT) compiler option is in effect, *argument-1* must be an integer greater than or equal to zero and less than or equal to 28. If the ARITH(EXTEND) compiler option is in effect, *argument-1* must be an integer greater than or equal to zero and less than or equal to 29.

If the value of *argument-1* is zero, the value 1 is returned; otherwise, the factorial of *argument-1* is returned.

INTEGER

The INTEGER function returns the greatest integer value that is less than or equal to the argument specified.

The function type is integer.

Format

►►—FUNCTION INTEGER—(—*argument-1*—)————►◄

argument-1

Must be class numeric.

The returned value is the greatest integer less than or equal to the value of *argument-1*. For example, FUNCTION INTEGER (2.5) returns a value of 2 and FUNCTION INTEGER (-2.5) returns a value of -3.

INTEGER-OF-DATE

The INTEGER-OF-DATE function converts a date in the Gregorian calendar from standard date form (YYYYMMDD) to integer date form.

The function type is integer.

The function result is a seven-digit integer with a range from 1 to 3,067,671.

Format

►►—FUNCTION INTEGER-OF-DATE—(—*argument-1*—)————►◄

argument-1

Must be an integer of the form YYYYMMDD, whose value is obtained from the calculation $(YYYY * 10,000) + (MM * 100) + DD$, where:

- YYYY represents the year in the Gregorian calendar. It must be an integer greater than 1600, but not greater than 9999.
- MM represents a month and must be a positive integer less than 13.
- DD represents a day and must be a positive integer less than 32, provided that it is valid for the specified month and year combination.

The returned value is an integer that is the number of days that the date represented by *argument-1* succeeds December 31, 1600 in the Gregorian calendar.

INTEGER-OF-DAY

The INTEGER-OF-DAY function converts a date in the Gregorian calendar from Julian date form (YYYYDDD) to integer date form.

The function type is integer.

The function result is a seven-digit integer.

Format

►►—FUNCTION INTEGER-OF-DAY—(—*argument-1*—)————►◄

argument-1

Must be an integer of the form YYYYDDD whose value is obtained from the calculation (YYYY * 1000) + DDD, where:

- YYYY represents the year in the Gregorian calendar. It must be an integer greater than 1600, but not greater than 9999.
- DDD represents the day of the year. It must be a positive integer less than 367, provided that it is valid for the year specified.

The returned value is an integer that is the number of days that the date represented by *argument-1* succeeds December 31, 1600 in the Gregorian calendar.

INTEGER-PART

The INTEGER-PART function returns an integer that is the integer portion of the argument specified.

The function type is integer.

Format

►►—FUNCTION INTEGER-PART—(—*argument-1*—)————►◄

argument-1

Must be class numeric.

If the value of *argument-1* is zero, the returned value is zero. If the value of *argument-1* is positive, the returned value is the greatest integer less than or equal to the value of *argument-1*. If the value of *argument-1* is negative, the returned value is the least integer greater than or equal to the value of *argument-1*.

LENGTH

The LENGTH function returns an integer equal to the length of the argument in national character positions for arguments of usage NATIONAL and in alphanumeric character positions or bytes for all other arguments. An alphanumeric character position and a byte are equivalent.

The type of the function is integer.

Format

►—FUNCTION LENGTH—(—*argument-1*—)————►

argument-1

Can be:

- An alphanumeric literal or a national literal
- A data item of any class except DBCS
- A data item described with usage POINTER, PROCEDURE-POINTER, FUNCTION-POINTER, or OBJECT REFERENCE
- The ADDRESS OF special register
- The LENGTH OF special register
- The XML-NTEXT special register
- The XML-TEXT special register

The returned value is a nine-digit integer if the ADDR(32) compiler option is specified or an 18-digit integer if the ADDR(64) compiler option is specified and is determined as follows:

- If *argument-1* is an alphanumeric literal or an elementary data item of class alphabetic or alphanumeric, the value returned is equal to the number of alphanumeric character positions in the argument.

If *argument-1* is a null-terminated alphanumeric literal, the returned value is equal to the number of alphanumeric character positions in the literal excluding the null character at the end of the literal.

The length of an alphanumeric data item or literal containing a mix of single-byte and double-byte characters is counted as though each byte were a single-byte character.

- If *argument-1* is an alphanumeric group item, the value returned is equal to the length of *argument-1* in alphanumeric character positions regardless of the content of the group. If any data item subordinate to *argument-1* is described with the DEPENDING phrase of the OCCURS clause, the length of *argument-1* is determined using the contents of the data item specified in the DEPENDING phrase. This evaluation is accomplished according to the rules of the OCCURS clause for a sending data item. For more information, see the discussions of the “OCCURS clause” on page 192 and the “USAGE clause” on page 229.

The returned value includes implicit FILLER positions, if any.

- If *argument-1* is a national literal or an elementary data item described with usage NATIONAL, the value returned is equal to the length of *argument-1* in national character positions.

For example, if *argument-1* is defined as PIC 9(3) with usage NATIONAL, the returned value is 3, although the storage size of the argument is 6 bytes.

- If *argument-1* is a national group item, the value returned is equal to the length of *argument-1* in national character positions. If any data item subordinate to *argument-1* is described with the DEPENDING phrase of the OCCURS clause, the length of *argument-1* is determined using the contents of the data item specified in the DEPENDING phrase. This evaluation is accomplished according to the rules of the OCCURS clause for a sending data item. For more information, see the discussions of the “OCCURS clause” on page 192 and the “USAGE clause” on page 229.

The returned value includes implicit FILLER positions, if any.

- Otherwise, the returned value is the number of bytes of storage occupied by *argument-1*.

LOG

The LOG function returns a numeric value that approximates the logarithm to the base e (natural log) of the argument specified.

The function type is numeric.

Format

►►—FUNCTION LOG—(*—argument-1—*)—————►◄

argument-1

Must be class numeric. The value of *argument-1* must be greater than zero.

The returned value is the approximation of the logarithm to the base e of *argument-1*.

LOG10

The LOG10 function returns a numeric value that approximates the logarithm to the base 10 of the argument specified.

The function type is numeric.

Format

►►—FUNCTION LOG10—(*—argument-1—*)—————►◄

argument-1

Must be class numeric. The value of *argument-1* must be greater than zero.

The returned value is the approximation of the logarithm to the base 10 of *argument-1*.

LOWER-CASE

The LOWER-CASE function returns a character string that contains the characters in the argument with each uppercase letter replaced by the corresponding lowercase letter.

The function type depends on the type of the argument, as follows:

Argument type	Function type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
National	National

Format

►►—FUNCTION LOWER-CASE—(—*argument-1*—)————►◄

argument-1

Must be class alphabetic, alphanumeric, or national and must be at least one character position in length.

The same character string as *argument-1* is returned, except that each uppercase letter is replaced by the corresponding lowercase letter.

The conversion of characters from uppercase to lowercase is based on the specification of character attributes in the applicable runtime locale.

For some locales, the conversion of characters from uppercase to lowercase can result in a character string with a different length than the length of *argument-1*. This can occur for all possible argument types. For alphabetic and alphanumeric arguments that consist solely of uppercase Latin letters A through Z, lowercase Latin letters a through z, and digits 0 through 9, the length of the returned character string is the same as the length of *argument-1*.

MAX

The MAX function returns the content of the argument that contains the maximum value.

The function type depends on the argument type, as follows:

Argument type	Function type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
National	National
All arguments integer (includes integer arguments of usage NATIONAL)	Integer
Numeric (some arguments can be integer) (includes numeric arguments of usage NATIONAL)	Numeric

Format

»»FUNCTION MAX—(—*argument-1*—)—»

argument-1

Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the content of *argument-1* having the greatest value. The comparisons used to determine the greatest value are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 262.

If more than one *argument-1* has the same greatest value, the leftmost *argument-1* having that value is returned.

If the type of the function is alphanumeric or national, the size of the returned value is the size of the selected *argument-1*.

MEAN

The MEAN function returns a numeric value that approximates the arithmetic average of its arguments.

The function type is numeric.

Format

»»FUNCTION MEAN—(—*argument-1*—)—»

argument-1

Must be class numeric.

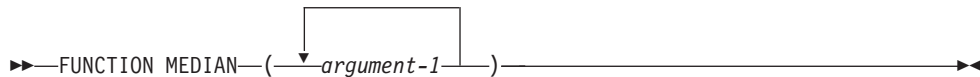
The returned value is the arithmetic mean of the *argument-1* series. The returned value is defined as the sum of the *argument-1* series divided by the number of occurrences referenced by *argument-1*.

MEDIAN

The MEDIAN function returns the content of the argument whose value is the middle value in the list formed by arranging the arguments in sorted order.

The function type is numeric.

Format



argument-1

Must be class numeric.

The returned value is the content of *argument-1* having the middle value in the list formed by arranging all *argument-1* values in sorted order.

If the number of occurrences referenced by *argument-1* is odd, the returned value is such that at least half of the occurrences referenced by *argument-1* are greater than or equal to the returned value and at least half are less than or equal. If the number of occurrences referenced by *argument-1* is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences.

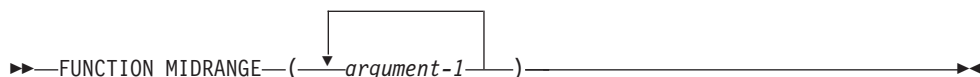
The comparisons used to arrange the argument values in sorted order are made according to the rules for simple conditions. For more information, see "Conditional expressions" on page 262.

MIDRANGE

The MIDRANGE function returns a numeric value that approximates the arithmetic average of the values of the minimum argument and the maximum argument.

The function type is numeric.

Format



argument-1

Must be class numeric.

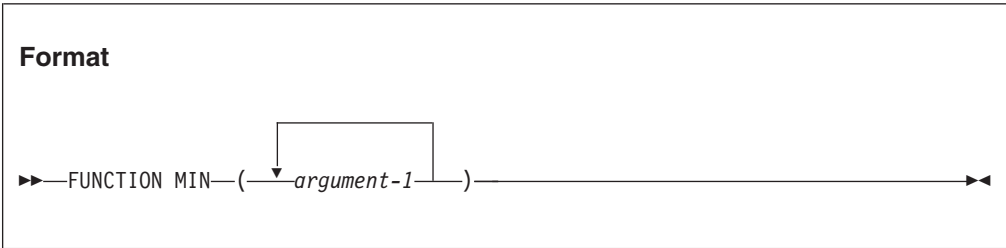
The returned value is the arithmetic mean of the value of the greatest *argument-1* and the value of the least *argument-1*. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 262.

MIN

The MIN function returns the content of the argument that contains the minimum value.

The function type depends on the argument type, as follows:

Argument type	Function type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
National	National
All arguments integer (includes integer arguments of usage NATIONAL)	Integer
Numeric (some arguments can be integer) (includes numeric arguments of usage NATIONAL)	Numeric



argument-1
Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the content of *argument-1* having the least value. The comparisons used to determine the least value are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 262.

If more than one *argument-1* has the same least value, the leftmost *argument-1* having that value is returned.

If the type of the function is alphanumeric or national, the size of the returned value is the size of the selected *argument-1*.

MOD

The MOD function returns an integer value that is *argument-1* modulo *argument-2*.

The function type is integer.

The function result is an integer with as many digits as the shorter of *argument-1* and *argument-2*.

Format

►►—FUNCTION MOD—(*—argument-1—argument-2—*)————►►

argument-1

Must be an integer.

argument-2

Must be an integer. Must not be zero.

The returned value is *argument-1* modulo *argument-2*. The returned value is defined as:

argument-1 - (*argument-2* * FUNCTION INTEGER (*argument-1* / *argument-2*))

The following are expected results for some values of *argument-1* and *argument-2*.

<i>argument-1</i>	<i>argument-2</i>	Returned value
11	5	1
-11	5	4
11	-5	-4
-11	-5	-1

NATIONAL-OF

The **NATIONAL-OF** function returns a national character string consisting of the national character representation of the characters in *argument-1*.

The type of the function is national.

Format

►►FUNCTION NATIONAL-OF-(*argument-1* *argument-2*)◄◄

argument-1

Must be of class alphabetic, alphanumeric, or DBCS. *argument-1* specifies the source string for the conversion.

argument-2

Must be an integer or of class alphanumeric. *argument-2* identifies the source code page for the conversion.

If *argument-2* is of class alphanumeric, it must identify a primary or alias code-page name that is supported by ICU conversion libraries (see *International Components for Unicode: Converter Explorer*).

If *argument-2* is an integer, the integer must be a valid CCSID number.

If *argument-2* is omitted, the source code page is determined as follows:

- If *argument-1* is a native item (USAGE DISPLAY or USAGE DISPLAY-1 containing ASCII or ASCII DBCS, EUC, or UTF-8 data), the source code page is determined from the runtime locale.
- If *argument-1* is a USAGE DISPLAY or USAGE DISPLAY-1 item containing EBCDIC or EBCDIC DBCS data, the source code page is determined from the EBCDIC_CODEPAGE environment variable, if it is set. If the EBCDIC_CODEPAGE environment variable is not set, the source code page is the default code page specified in the *Locales and code pages that are supported* in the *COBOL for AIX Programming Guide*.

The returned value is a national character string consisting of the characters of *argument-1* converted to national character representation. When a source character cannot be converted to a national character, the source character is converted to a substitution character. The substitution character is:

- X'001A' if converting a single-byte character
- X'FFFD' if converting a multi-byte character

No exception condition is raised.

The length of the returned value depends on the content of *argument-1* and the characteristics of the source code page.

Usage notes:

- Use of code page names provides consistency with other AIX software, but the source code is not portable to Enterprise COBOL for z/OS.
- The CCSID for UTF-8 is 1208.
- The CCSID for UTF-16LE is 1200.

Exception: If the conversion fails, a severe runtime error occurs.

NUMVAL

The NUMVAL function returns the numeric value represented by the alphanumeric character string or national character string specified as the argument. The function removes any leading or trailing spaces in the string to produce a numeric value.

The function type is numeric.

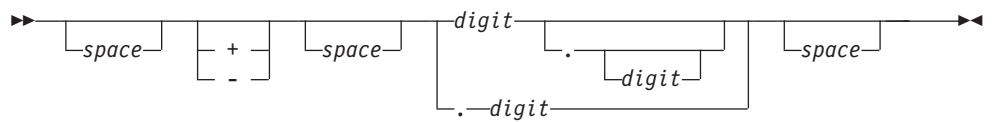
Format

►►FUNCTION NUMVAL—(—*argument-1*—)◄◄

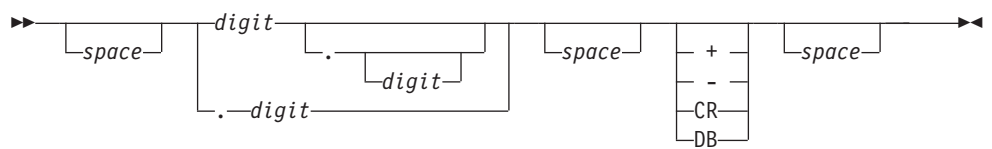
argument-1

Must be an alphanumeric literal, a national literal, or a data item of class national or class alphanumeric that contains a character string in either of the following formats:

Format 1: *argument-1*



Format 2: *argument-1*, monetary format



space A string of one or more spaces.

digit A string of one or more digits. If the ARITH(COMPAT) compiler option is in effect, the total number of digits must not exceed 18. If the ARITH(EXTEND) compiler option is in effect, the total number of digits must not exceed 31.

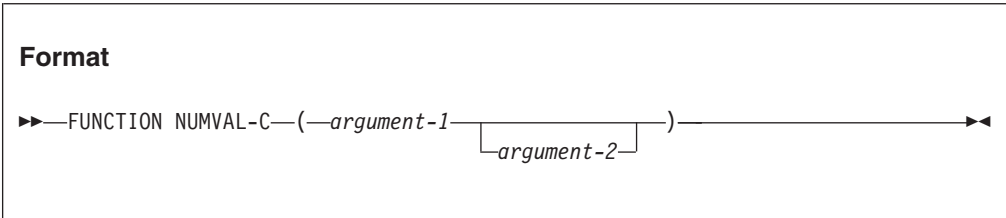
If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma must be used in *argument-1* rather than a decimal point.

The returned value is a floating-point approximation of the numeric value represented by *argument-1*. The precision of the returned value depends on the setting of the ARITH compiler option. For details, see *Converting to number (NUMVAL, NUMVAL-C)* in the *COBOL for AIX Programming Guide*.

NUMVAL-C

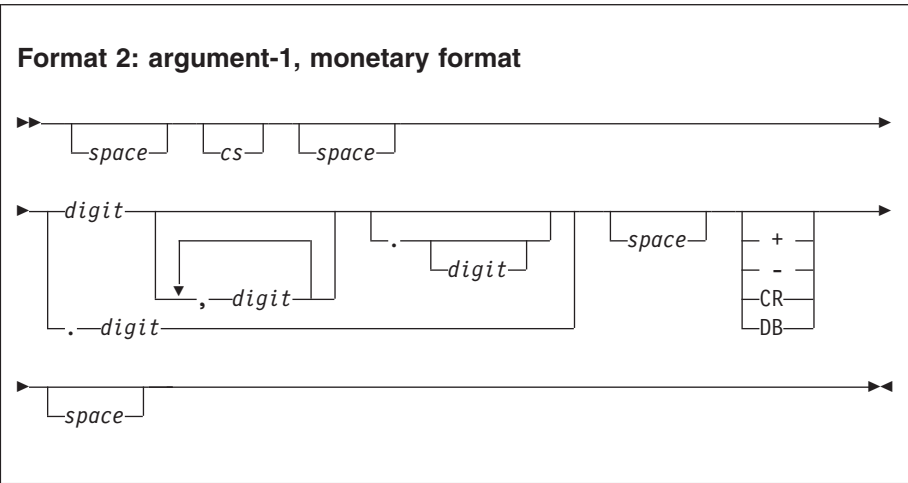
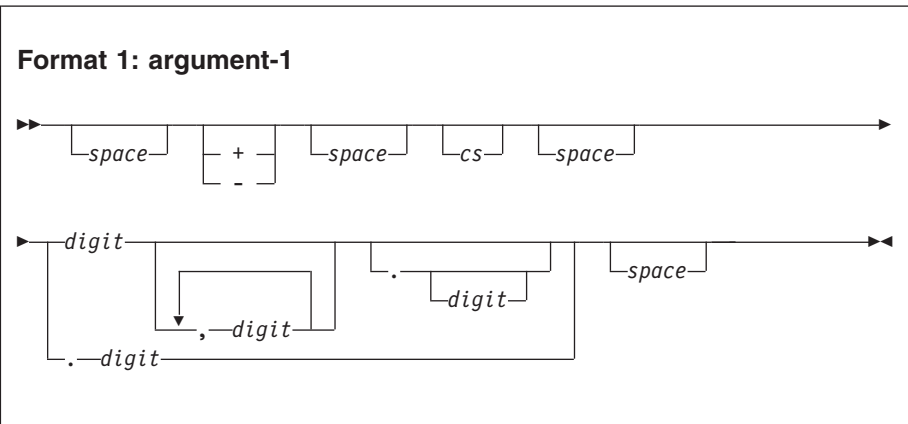
The NUMVAL-C function returns the numeric value represented by the alphanumeric character string or national character string specified as *argument-1*. The function removes the currency string, if any, and any grouping separators (commas or periods) to produce a numeric value.

The function type is numeric.



argument-1

Must be an alphanumeric literal, a national literal, or a data item of class alphanumeric or class national that contains a character string in either of the following formats:



- space*** A string of one or more spaces.
- cs*** The string of one or more characters that form the currency sign. At most one copy of the characters specified by *cs* can occur in *argument-1*.
- digit*** A string of one or more digits. If the ARITH(COMPAT) compiler

option is in effect, the total number of digits must not exceed 18. If the ARITH(EXTEND) compiler option is in effect, the total number of digits must not exceed 31.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in *argument-1* are reversed.

argument-2

Specifies the currency string value.

The following rules apply:

- *argument-2* must be specified if the program contains more than one CURRENCY SIGN clause.
- *argument-2*, if specified, must be of the same class as *argument-1*.
- *argument-2* must not contain any of the digits 0 through 9, any leading or trailing spaces, or any of the special characters '+', '-', '.', or ','.
- *argument-2* can be of any length valid for an elementary or group data item of the class of *argument-2*, including zero.
- Matching of *argument-2* is case sensitive. For example, if you specify *argument-2* as 'CHF', it will not match 'ChF', 'chf' or 'chF'.

If *argument-2* is not specified, the character used for *cs* is the currency symbol specified for the program.

The returned value is a floating-point approximation of the numeric value represented by *argument-1*. The precision of the returned value depends on the setting of the ARITH compiler option. For details, see *Converting to number* (NUMVAL, NUMVAL-C) in the *COBOL for AIX Programming Guide*.

ORD

The ORD function returns an integer value that is the ordinal position of its argument in the collating sequence for the program. The lowest ordinal position is 1.

The function type is integer.

The function result is a three-digit integer.

Format

►►—FUNCTION ORD—(—*argument-1*—)————►◄

argument-1

Must be one character in length and must be class alphabetic or alphanumeric.

The returned value is the ordinal position of *argument-1* in the collating sequence for the program; it ranges from 1 to 256 depending on the collating sequence.

ORD-MAX

The ORD-MAX function returns a value that is the ordinal position in the argument list of the argument that contains the maximum value.

The function type is integer.

Format

» FUNCTION ORD-MAX—(—*argument-1*—)—»

argument-1

Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the ordinal number that corresponds to the position of the *argument-1* having the greatest value in the *argument-1* series.

The comparisons used to determine the greatest-valued *argument-1* are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 262.

If more than one *argument-1* has the same greatest value, the number returned corresponds to the position of the leftmost *argument-1* having that value.

ORD-MIN

The ORD-MIN function returns a value that is the ordinal position in the argument list of the argument that contains the minimum value.

The function type is integer.

Format

» FUNCTION ORD-MIN—(—*argument-1*—)—»

argument-1

Must be class alphabetic, alphanumeric, national, or numeric.

All arguments must be of the same class, except that a combination of alphabetic and alphanumeric arguments is allowed.

The returned value is the ordinal number that corresponds to the position of the *argument-1* having the least value in the *argument-1* series.

The comparisons used to determine the least-valued *argument-1* are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 262.

If more than one *argument-1* has the same least value, the number returned corresponds to the position of the leftmost *argument-1* having that value.

PRESENT-VALUE

The PRESENT-VALUE function returns a value that approximates the present value of a series of future period-end amounts specified by *argument-2* at a discount rate specified by *argument-1*.

The function type is numeric.

Format

►►—FUNCTION PRESENT-VALUE—(—*argument-1*—*argument-2*—)—◄◄

argument-1

Must be class numeric. Must be greater than -1.

argument-2

Must be class numeric.

The returned value is an approximation of the summation of a series of calculations with each term in the following form:

$$\textit{argument-2} / (1 + \textit{argument-1})^{**} n$$

There is one term for each occurrence of *argument-2*. The exponent *n* is incremented from 1 by 1 for each term in the series.

RANDOM

The RANDOM function returns a numeric value that is a pseudorandom number from a rectangular distribution.

The function type is numeric.

Format

►►—FUNCTION RANDOM—┐
└(—*argument-1*—)—┘

argument-1

If *argument-1* is specified, it must be zero or a positive integer. However, only values in the range from zero up to and including 2,147,483,645 yield a distinct sequence of pseudorandom numbers.

If a subsequent reference specifies *argument-1*, a new sequence of pseudorandom numbers is started.

If the first reference to this function in the run unit does not specify *argument-1*, the seed value used will be zero.

In each case, subsequent references without specifying *argument-1* return the next number in the current sequence.

The returned value is exclusively between zero and one.

For a given seed value, the sequence of pseudorandom numbers is always the same.

The RANDOM function can be used in threaded programs. For an initial seed, a single sequence of pseudorandom numbers is returned, regardless of the thread that is running when RANDOM is invoked.

RANGE

The RANGE function returns a value that is equal to the value of the maximum argument minus the value of the minimum argument.

The function type depends on the argument types, as follows:

Argument type	Function type
All arguments integer	Integer
Numeric (some arguments can be integer)	Numeric

Format

►►—FUNCTION RANGE—(—*argument-1*—)—┐
└—*argument-2*—┘

argument-1

Must be class numeric.

The returned value is equal to *argument-1* with the greatest value minus the *argument-1* with the least value. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions. For more information, see “Conditional expressions” on page 262.

REM

The REM function returns a numeric value that is the remainder of *argument-1* divided by *argument-2*.

The function type is numeric.

Format

►►FUNCTION REM—(—*argument-1*—*argument-2*—)————►►

- argument-1*
Must be class numeric.
- argument-2*
Must be class numeric. Must not be zero.

The returned value is the remainder of *argument-1* divided by *argument-2*. It is defined as the expression:

$$argument-1 - (argument-2 * FUNCTION INTEGER-PART (argument-1 / argument-2))$$

REVERSE

The REVERSE function returns a character string of exactly the same length as the argument, whose characters are exactly the same as those specified in the argument except that they are in reverse order. For arguments of type national, character positions are reversed.

The function type depends on the type of the argument, as follows:

Argument type	Function type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
National	National

Format

►►FUNCTION REVERSE—(—*argument-1*—)————►►

argument-1

Must be class alphabetic, alphanumeric, or national and must be at least one character in length.

The returned value is a character string of the same length as *argument-1*, with the characters of *argument-1* in reversed order. For example, if *argument-1* contains ABC, the returned value is CBA.

SIN

The SIN function returns a numeric value that approximates the sine of the angle or arc specified by the argument in radians.

The function type is numeric.

Format

►►—FUNCTION SIN—(*—argument-1—*)—►►

argument-1

Must be class numeric.

The returned value is the approximation of the sine of *argument-1* and is greater than or equal to -1 and less than or equal to +1.

SQRT

The SQRT function returns a numeric value that approximates the square root of the argument specified.

The function type is numeric.

Format

►►—FUNCTION SQRT—(*—argument-1—*)—►►

argument-1

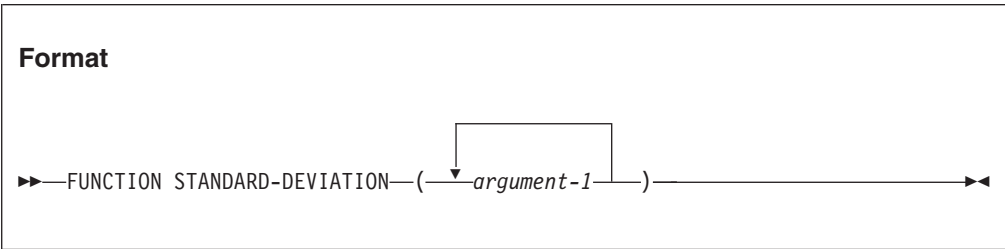
Must be class numeric. The value of *argument-1* must be zero or positive.

The returned value is the absolute value of the approximation of the square root of *argument-1*.

STANDARD-DEVIATION

The STANDARD-DEVIATION function returns a numeric value that approximates the standard deviation of its arguments.

The function type is numeric.



argument-1

Must be class numeric.

The returned value is the approximation of the standard deviation of the *argument-1* series. The returned value is calculated as follows:

1. The difference between each *argument-1* and the arithmetic mean of the *argument-1* series is calculated and squared.
2. The values obtained are then added together. This quantity is divided by the number of values in the *argument-1* series.
3. The square root of the quotient obtained is then calculated. The returned value is the absolute value of this square root.

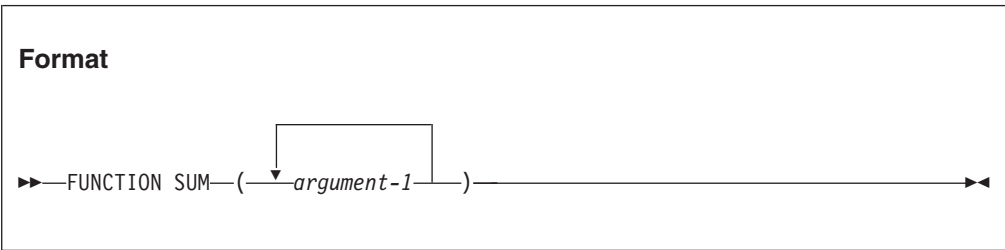
If the *argument-1* series consists of only one value, or if the *argument-1* series consists of all variable-occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.

SUM

The SUM function returns a value that is the sum of the arguments.

The function type depends on the argument types, as follows:

Argument type	Function type
All arguments integer	Integer
Numeric (some arguments can be integer)	Numeric



argument-1

Must be class numeric.

The returned value is the sum of the arguments. If the *argument-1* series are all integers, the value returned is an integer. If the *argument-1* series are not all integers, a numeric value is returned.

TAN

The TAN function returns a numeric value that approximates the tangent of the angle or arc that is specified by the argument in radians.

The function type is numeric.

Format

►►—FUNCTION TAN—(*—argument-1—*)—►►

argument-1

Must be class numeric.

The returned value is the approximation of the tangent of *argument-1*.

UNDATE

The UNDATE function converts a date field to a nondate for unambiguous use with nondates.

If the NODATEPROC compiler option is in effect, the UNDATE function has no effect.

The function type depends on the type of *argument-1*:

Argument type	Function type
Alphanumeric	Alphanumeric
Integer	Integer

Format

►►—FUNCTION UNDATE—(*—argument-1—*)—►►

argument-1

A date field.

The returned value is a nondate that contains the value of *argument-1* unchanged.

UPPER-CASE

The UPPER-CASE function returns a character string that contains the characters in the argument with each lowercase letter replaced by the corresponding uppercase letter.

The function type depends on the type of the argument, as follows:

Argument type	Function type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
National	National

Format

►►—FUNCTION UPPER-CASE—(—*argument-1*—)—————►►

argument-1

Must be class alphabetic, alphanumeric, or national and must be at least one character position in length.

The same character string as *argument-1* is returned, except that each lowercase letter is replaced by the corresponding uppercase letter.

The conversion of characters from lowercase to uppercase is based on the specification of character attributes in the applicable runtime locale.

For some locales, the conversion of characters from lowercase to uppercase can result in a character string with a different length than the length of *argument-1*. This can occur for all possible arguments type. For alphabetic and alphanumeric arguments that consist solely of uppercase Latin letters A through Z, lowercase Latin letters a through z, and digits 0 through 9, the length of the returned character string is the same as the length of *argument-1*.

VARIANCE

The VARIANCE function returns a numeric value that approximates the variance of its arguments.

The function type is numeric.

Format

►►—FUNCTION VARIANCE—(—*argument-1*—)—►◄

argument-1

Must be class numeric.

The returned value is the approximation of the variance of the *argument-1* series.

The returned value is defined as the square of the standard deviation of the *argument-1* series. This value is calculated as follows:

1. The difference between each *argument-1* value and the arithmetic mean of the *argument-1* series is calculated and squared.
2. The values obtained are then added together. This quantity is divided by the number of values in the argument series.

If the *argument-1* series consists of only one value, or if the *argument-1* series consists of all variable-occurrence data items and the total number of occurrences for all of them is one, the returned value is zero.

WHEN-COMPILED

The WHEN-COMPILED function returns the date and time that the program was compiled as provided by the system on which the program was compiled.

The function type is alphanumeric.

Format

►►—FUNCTION WHEN-COMPILED—►◄

Reading from left to right, the 21 character positions of the returned value are as follows:

Character positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar
5-6	Two numeric digits of the month of the year, in the range 01 through 12
7-8	Two numeric digits of the day of the month, in the range 01 through 31
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59

Character positions	Contents
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99. The value 00 is returned if the system on which the function is evaluated does not have the facility to provide the fractional part of a second.
17	Either the character '-' or the character '+'. The character '-' is returned if the local time indicated in the previous character positions is behind Greenwich mean time. The character '+' is returned if the local time indicated is the same as or ahead of Greenwich mean time. The character '0' is returned if the system on which this function is evaluated does not have the facility to provide the local time differential factor.
18-19	If character position 17 is '-', two numeric digits are returned in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich mean time. If character position 17 is '+', two numeric digits are returned in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich mean time. If character position 17 is '0', the value 00 is returned.
20-21	Two numeric digits are returned in the range 00 through 59 indicating the number of additional minutes that the reported time is ahead of or behind Greenwich mean time, depending on whether character position 17 is '+' or '-', respectively. If character position 17 is '0', the value 00 is returned.

The returned value is the date and time of compilation of the source unit that contains this function. If a program is a contained program, the returned value is the compilation date and time associated with the containing program.

YEAR-TO-YYYY

The YEAR-TO-YYYY function converts *argument-1*, a two-digit year, to a four-digit year. *argument-2*, when added to the year at the time of execution, defines the ending year of a 100-year interval, or sliding century window, into which the year of *argument-1* falls.

The function type is integer.

If the DATEPROC compiler option is in effect, then the returned value is an expanded date field with implicit DATE FORMAT YYYY.

Format

►►—FUNCTION YEAR-TO-YYYY—(*argument-1* *argument-2*)—►

argument-1

Must be a non-negative integer that is less than 100.

argument-2

Must be an integer. If *argument-2* is omitted, the function is evaluated assuming the value 50 was specified.

The sum of the year at the time of execution and the value of *argument-2* must be less than 10,000 and greater than 1,699.

Examples of return values from the YEAR-TO-YYYY function are shown in the following table.

Current year	<i>argument-1</i> value	<i>argument-2</i> value	Returned value
1995	4	23	2004
1995	4	-15	1904
2008	98	23	1998
2008	98	-15	1898

YEARWINDOW

If the DATEPROC compiler option is in effect, the YEARWINDOW function returns the starting year of the century window specified by the YEARWINDOW compiler option. The returned value is an expanded date field with implicit DATE FORMAT YYYY.

If the NODATEPROC compiler option is in effect, the YEARWINDOW function returns 0.

The function type is integer.

Format

►►—FUNCTION YEARWINDOW—◄◄

Part 8. Compiler-directing statements

Chapter 23. Compiler-directing statements	537
BASIS statement	537
CBL (PROCESS) statement	538
*CONTROL (*CBL) statement	538
Source code listing	540
Object code listing	540
Storage map listing	540
COPY statement	540
SUPPRESS phrase	543
REPLACING phrase	543
Replacement and comparison rules	544
DELETE statement	547
EJECT statement	548
ENTER statement	548
INSERT statement	549
READY or RESET TRACE statement	550
REPLACE statement	550
Continuation rules for pseudo-text	552
Comparison operation	552
REPLACE statement notes	553
SERVICE LABEL statement	554
SERVICE RELOAD statement	554
SKIP statements	554
TITLE statement	555
USE statement	556
EXCEPTION/ERROR declarative	556
Precedence rules for nested programs	557
LABEL declarative	558
DEBUGGING declarative	558
Chapter 24. Compiler directives	561
CALLINTERFACE	561
Syntax and general rules	561

Chapter 23. Compiler-directing statements

A *compiler-directing statement* is a statement that causes the compiler to take a specific action during compilation.

You can use compiler-directing statements for the following:

- Extended source library control (BASIS, DELETE, and INSERT statements)
- Source text manipulation (COPY and REPLACE statements)
- Exception handling (USE statement)
- Controlling compiler listings (*CONTROL, *CBL, EJECT, TITLE, SKIP1, SKIP2, and SKIP3 statements)
- Specifying compiler options (CBL and PROCESS statements)
- Specifying COBOL exception handling procedures (USE statements)

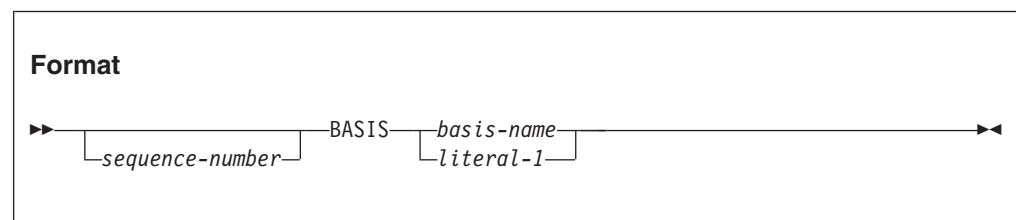
You use the compiler directive CALLINTERFACE to specify interface conventions for the CALL statement.

The following compiler directing statements have no effect: ENTER, READY or RESET TRACE, SERVICE LABEL, and SERVICE RELOAD.

BASIS statement

The BASIS statement is an extended source text library statement. It provides a complete COBOL program as the source for a compilation.

A complete program can be stored as an entry in a user-defined library and can be used as the source for a compilation. Compiler input is a BASIS statement, optionally followed by any number of INSERT and DELETE statements.



sequence-number

Can optionally appear in columns 1 through 6, followed by a space. The content of this field is ignored.

BASIS

Can appear anywhere in columns 1 through 72 in fixed source format, or columns 1 through 252 in extended source format, followed by *basis-name*. There must be no other text in the statement.

basis-name, literal-1

Is the name by which the library entry is known to the system environment.

For rules of formation and processing rules, see the description under *literal-1* and *text-name* of the “COPY statement” on page 540.

The source file remains unchanged after execution of the BASIS statement.

Usage note: If INSERT or DELETE statements are used to modify the COBOL source text provided by a BASIS statement, the sequence field of the COBOL source text must contain numeric sequence numbers in ascending order.

CBL (PROCESS) statement

With the CBL (PROCESS) statement, you can specify compiler options to be used in the compilation of the program. The CBL (PROCESS) statement is placed before the identification division header of an outermost program.

Format



options-list

A series of one or more compiler options, each one separated by a comma or a space.

For more information about compiler options, see *Compiler options* in the *COBOL for AIX Programming Guide*.

The CBL (PROCESS) statement can be preceded by a sequence number in columns 1 through 6. The first character of the sequence number must be numeric, and CBL or PROCESS can begin in column 8 or after; if a sequence number is not specified, CBL or PROCESS can begin in column 1 or after.

The CBL (PROCESS) statement must end before or at column 72 in fixed source format, or column 252 in extended source format, and options cannot be continued across multiple CBL (PROCESS) statements. However, you can use more than one CBL (PROCESS) statement. Multiple CBL (PROCESS) statements must follow one another with no intervening statements of any other type.

The CBL (PROCESS) statement must be placed before any comment lines or other compiler-directing statements.

*CONTROL (*CBL) statement

With the *CONTROL (or *CBL) statement, you can selectively display or suppress the listing of source code and storage maps throughout the source text.

The diagram shows a central horizontal line with arrows at both ends. Above the line, from left to right, are the labels `*CONTROL`, `SOURCE`, and `.`. Below the line, from left to right, are the labels `*CBL`, `NOSOURCE`, `LIST`, `NOLIST`, `MAP`, and `NOMAP`. A vertical line segment connects the central line to a box containing the labels `SOURCE`, `NOSOURCE`, `LIST`, `NOLIST`, `MAP`, and `NOMAP`. A horizontal line segment connects the central line to a box containing the label `.`. A vertical line segment connects the central line to a box containing the label `*CONTROL`. A horizontal line segment connects the central line to a box containing the label `*CBL`.

For information about specifying whether listings are encoded in UTF-8 or in the code page specified by the compile-time locale, see *LSTFILE* in the *COBOL for AIX Programming Guide*.

The characters *CONTROL or *CBL can start in any column beginning with column 7, followed by at least one space or comma and one or more option keywords. The option keywords must be separated by one or more spaces or commas. This statement must be the only statement on the line, and continuation is not allowed. The statement can be terminated with a period.

The source line containing the *CONTROL (*CBL) statement will not appear in the source listing.

- PARM (if available)
- CBL statement
- *CONTROL (*CBL) statement

1. If an option or its negation appears more than once in a *CONTROL statement, the last occurrence of the option word is used.
2. If the corresponding option has been requested as a parameter to the compiler, then a *CONTROL statement with the negation of the option word must precede the portions of the source text for which listing output is to be inhibited. Listing output then resumes when a *CONTROL statement with the affirmative option word is encountered.
3. If the negation of the corresponding option has been requested as a parameter to the compiler, then that listing is *always* inhibited.

4. The `*CONTROL` statement is in effect only within the source program in which it is written, including any contained programs. It does not remain in effect across batch compiles of two or more COBOL source programs.

Source code listing

Listing of the input source text lines is controlled by any of the following statements:

<code>*CONTROL SOURCE</code>	<code>[*CBL SOURCE]</code>
<code>*CONTROL NOSOURCE</code>	<code>[*CBL NOSOURCE]</code>

If a `*CONTROL NOSOURCE` statement is encountered and `SOURCE` has been requested as a compilation option, printing of the source listing is suppressed from this point on. An informational (I-level) message is issued stating that printing of the source has been suppressed.

Object code listing

The compiler always produces an object code listing. The `LIST` and `NOLIST` options of the `*CONTROL` (or `*CBL`) statement are syntax checked but have no effect on the object code listing.

Storage map listing

Listing of storage map entries is controlled by any of the following statements occurring in the data division:

<code>*CONTROL MAP</code>	<code>[*CBL MAP]</code>
<code>*CONTROL NOMAP</code>	<code>[*CBL NOMAP]</code>

If a `*CONTROL NOMAP` statement is encountered, and `MAP` has been requested as a compilation option, listing of storage map entries is suppressed from this point on.

For example, either of the following sets of statements produces a storage map listing in which A and B will not appear:

<code>*CONTROL NOMAP</code>	<code>*CBL NOMAP</code>
01 A	01 A
02 B	02 B
<code>*CONTROL MAP</code>	<code>*CBL MAP</code>

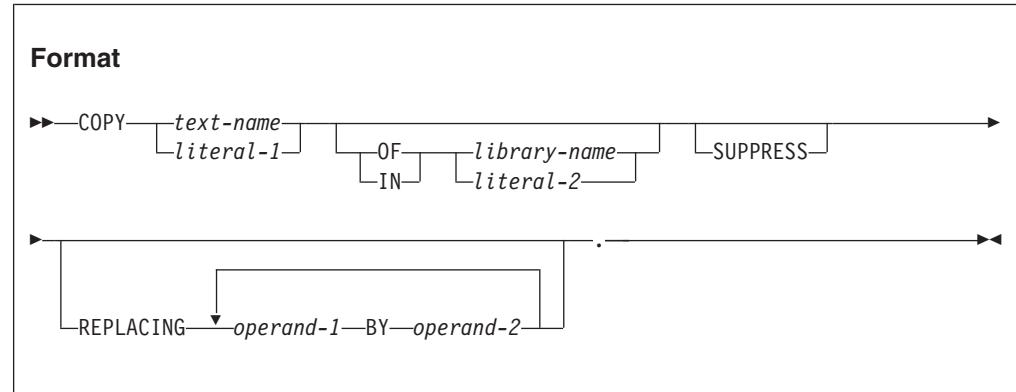
COPY statement

The `COPY` statement is a library statement that places prewritten text in a COBOL compilation unit.

Prewritten source code entries can be included in a compilation unit at compile time. Thus, an installation can use standard file descriptions, record descriptions, or procedures without recoding them. These entries and procedures can then be saved in user-created libraries; they can then be included in programs and class definitions by means of the `COPY` statement.

Compilation of the source code containing `COPY` statements is logically equivalent to processing all `COPY` statements before processing the resulting source text.

The effect of processing a COPY statement is that the library text associated with *text-name* is copied into the compilation unit, logically replacing the entire COPY statement, beginning with the word COPY and ending with the period, inclusive. When the REPLACING phrase is not specified, the library text is copied unchanged.



text-name, library-name

text-name identifies the copy text. *library-name* identifies where the copy text exists.

- Can be from 1-30 characters in length
- Can contain the following characters: Latin uppercase letters A-Z, Latin lowercase letters a-z, digits 0-9, hyphen, and underscore

Neither *text-name* nor *library-name* need to be unique within a program. They can be identical to other user-defined words in the program.

text-name need not be qualified. If *text-name* is not qualified, a library-name of SYSLIB is assumed.

literal-1, literal-2

Must be alphanumeric literals. *literal-1* identifies the copy text. *literal-2* identifies where the copy text exists.

The literal can be from 1-160 characters in length.

The uniqueness of *text-name* and *library-name* is determined after the formation and conversion rules for a system-dependent name have been applied.

For information about the mapping of characters in the *text-name*, *library-name*, and literals, see *Compiler-directing statements* in the *COBOL for AIX Programming Guide*.

operand-1, operand-2

Can be either pseudo-text, an identifier, a function-identifier, a literal, or a COBOL word (except the word COPY). For details, see "REPLACING phrase" on page 543.

Library text and pseudo-text can consist of or include any words (except COPY), identifiers, or literals that can be written in the source text. This includes multibyte user-defined words, multibyte literals, and national literals.

Multibyte user-defined words must be wholly formed; that is, there is no partial-word replacement for multibyte words.

Words or literals containing multibyte characters cannot be continued across lines.

Each COPY statement must be preceded by a space and ended with a separator period.

A COPY statement can appear in the source text anywhere a character string or a separator can appear.

COPY statements can be nested. However, nested COPY statements cannot contain the REPLACING phrase, and a COPY statement with the REPLACING phrase cannot contain nested COPY statements.

A nested COPY statement cannot cause recursion. That is, a COPY member can be named only once in a set of nested COPY statements until the end-of-file for that COPY member is reached. For example, assume that the source text contains the statement: COPY X. and library text X contains the statement: COPY Y..

In this case, library text Y must not have a COPY X or a COPY Y statement.

Debugging lines are permitted within library text and pseudo-text. Text words within a debugging line participate in the matching rules as if the "D" did not appear in the indicator area. A debugging line is specified within pseudo-text if the debugging line begins in the source text after the opening pseudo-text delimiter but before the matching closing pseudo-text delimiter.

If additional lines are introduced into the source text as a result of a COPY statement, each text word introduced appears on a debugging line if the COPY statement begins on a debugging line or if the text word being introduced appears on a debugging line in library text. When a text word specified in the BY phrase is introduced, it appears on a debugging line if the first library text word being replaced is specified on a debugging line.

When a COPY statement is specified on a debugging line, the copied text is treated as though it appeared on a debugging line, except that comment lines in the text appear as comment lines in the resulting source text.

If the word COPY appears in a comment-entry, or in the place where a comment-entry can appear, it is considered part of the comment-entry.

After all COPY and REPLACE statements have been processed, a debugging line will be considered to have all the characteristics of a comment line, if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph.

Comment lines, inline comments, or blank lines can occur in library text. Comment lines, inline comments, or blank lines appearing in library text are copied into the resultant source text unchanged with the following exception: a comment line, an inline comment, or a blank line in library text is not copied if that comment line, inline comment, or blank line appears within the sequence of text words that match *operand-1* (see "Replacement and comparison rules" on page 544).

Lines containing *CONTROL (*CBL), EJECT, SKIP1, SKIP2, SKIP3, or TITLE statements can occur in library text. Such lines are treated as comment lines during COPY statement processing.

The syntactic correctness of the entire COBOL source text cannot be determined until all COPY and REPLACE statements have been completely processed, because the syntactic correctness of the library text cannot be independently determined.

Library text copied from the library is placed into the same area of the resultant program as it is in the library. Library text must conform to the rules for Standard COBOL 85 format.

Note: Characters outside those defined for COBOL words and separators must not appear in library text or pseudo-text except in comment lines, comment-entries, alphanumeric literals, DBCS literals, or national literals.

SUPPRESS phrase

The SUPPRESS phrase specifies that the library text is not to be printed on the source listing.

REPLACING phrase

In the discussion that follows, each *operand* can consist of one of the following:

- Pseudo-text
- An identifier
- A literal
- A COBOL word (except the word COPY)
- Function identifier

When the REPLACING phrase is specified, the library text is copied, and each properly matched occurrence of *operand-1* within the library text is replaced by the associated *operand-2*.

pseudo-text

A sequence of character-strings or separators, or both, bounded by, but not including, pseudo-text delimiters (==). Both characters of each pseudo-text delimiter must appear on one line; however, character-strings within pseudo-text can be continued.

Individual character-strings within pseudo-text can be up to 322 characters long; they can be continued subject to the normal continuation rules for source code format.

Keep in mind that a character-string must be delimited by separators. For more information, see Chapter 1, “Characters,” on page 3.

pseudo-text-1 refers to pseudo-text when used for *operand-1*, and *pseudo-text-2* refers to pseudo-text when used for *operand-2*.

pseudo-text-1 can consist solely of the separator comma or separator semicolon. *pseudo-text-2* can be null; it can consist solely of space characters or comment lines.

Pseudo-text must not contain the word COPY.

Each text word in *pseudo-text-2* that is to be copied into the program is placed in the same area of the resultant program as the area in which it appears in *pseudo-text-2*.

Pseudo-text can consist of or include any words (except COPY), identifiers, or literals that can be written in the source text. This includes multibyte user-defined words, DBCS literals, and national literals.

Multibyte user-defined words must be wholly formed; that is, there is no partial-word replacement for multibyte words.

Words or literals containing multibyte characters cannot be continued across lines.

identifier

Can be defined in any section of the data division.

literal Can be numeric, alphanumeric, DBCS, or national.

word Can be any single COBOL word (except COPY), including multibyte user-defined words. Multibyte user-defined words must be wholly formed. You cannot replace part of a multibyte word.

You can include the nonseparator COBOL characters (for example, +, *, /, \$, <, >, and =) as part of a COBOL word when used as REPLACING operands. In addition, a hyphen or underscore can be at the beginning of the word or a hyphen can be at the end of the word.

For purposes of matching, each *identifier-1*, *literal-1*, or *word-1* is treated as pseudo-text containing only *identifier-1*, *literal-1*, or *word-1*, respectively.

Replacement and comparison rules

1. Arithmetic and logical operators are considered text words and can be replaced only through the pseudo-text option.
2. Beginning and ending blanks are not included in the text comparison process. Embedded blanks are used in the text comparison process to separate multiple text words.
3. When *operand-1* is a figurative constant, *operand-1* matches only the same exact figurative constant. For example, if ALL "AB" is specified in the library text, then "ABAB" is not considered a match; only ALL "AB" is considered a match.
4. When replacing a PICTURE character-string, the pseudo-text option should be used; to avoid ambiguities, *pseudo-text-1* should specify the entire PICTURE clause, including the keyword PICTURE or PIC.
5. Any separator comma, semicolon, or space preceding the leftmost word in the library text is copied into the source text. Beginning with the leftmost library text word and the first *operand-1* specified in the REPLACING option, the entire REPLACING operand that precedes the keyword BY is compared to an equivalent number of contiguous library text words.
6. *operand-1* matches the library text if, and only if, the ordered sequence of text words in *operand-1* is equal, character for character, to the ordered sequence of library words. For national characters, the sequence of national characters must be equal, national character for national character, to the ordered sequence of library words. For matching purposes, each occurrence of a comma or semicolon separator and each sequence of one or more space separators is considered to be a single space. However, when *operand-1* consists solely of a separator comma or semicolon, it participates in the match as a text word (in this case, the space following the comma or semicolon separator can be omitted).

When the library text contains a closing quotation mark that is not immediately followed by a separator space, comma, semicolon, or period, the closing quotation mark is considered a separator quotation mark.

7. If no match occurs, the comparison is repeated with each successive *operand-1*, if specified, until either a match is found or there are no further REPLACING operands.
8. Whenever a match occurs between *operand-1* and the library text, the associated *operand-2* is copied into the source text.
9. The COPY statement with REPLACING phrase can be used to replace parts of words. By inserting a dummy operand delimited by colons into the program text, the compiler will replace the dummy operand with the desired text. Example 3 shows how this is used with the dummy operand :TAG:. The colons serve as separators and make TAG a stand-alone operand.
10. When all operands have been compared and no match is found, the leftmost library text word is copied into the source text.
11. The next successive uncopied library text word is then considered to be the leftmost text word, and the comparison process is repeated, beginning with the first *operand-1*. The process continues until the rightmost library text word has been compared.
12. Comment lines, inline comments, or blank lines occurring in the library text and in *pseudo-text-1* are ignored for purposes of matching; and the sequence of text words in the library text and in *pseudo-text-1* is determined by the rules for reference format. Comment lines, inline comments, or blank lines appearing in *pseudo-text-2* are copied into the resultant program unchanged whenever *pseudo-text-2* is placed into the source text as a result of text replacement. Comment lines, inline comments, or blank lines appearing in library text are copied into the resultant source text unchanged with the following exception: a comment line, an inline comment, or a blank line in library text is not copied if that comment line, inline comment, or blank line appears within the sequence of text words that match *pseudo-text-1*.
13. Text words, after replacement, are placed in the source text according to Standard COBOL 85 format rules.
14. When text words are placed in the source text, additional spaces are introduced only between text words where there already exists a space (including the assumed space between source lines).
15. COPY REPLACING does not affect the EJECT, SKIP1, SKIP2, SKIP3, or TITLE compiler-directing statements.

Sequences of code (such as file and data descriptions, error and exception routines) that are common to a number of programs can be saved in a library, and then used in conjunction with the COPY statement. If naming conventions are established for such common code, then the REPLACING phrase need not be specified. If the names will change from one program to another, then the REPLACING phrase can be used to supply meaningful names for this program.

Example 1

In this example, the library text PAYLIB consists of the following data division entries:

```
01 A.
   02 B    PIC S99.
   02 C    PIC S9(5)V99.
   02 D    PIC S9999 OCCURS 1 TO 52 TIMES
           DEPENDING ON B OF A.
```

The programmer can use the COPY statement in the data division of a program as follows:

```
COPY PAYLIB.
```

In this program, the library text is copied; the resulting text is treated as if it had been written as follows:

```
01  A.
    02  B      PIC S99.
    02  C      PIC S9(5)V99.
    02  D      PIC S9999 OCCURS 1 TO 52 TIMES
            DEPENDING ON B OF A.
```

Example 2

To change some (or all) of the names within the library text, the programmer can use the REPLACING phrase:

```
COPY PAYLIB REPLACING A BY PAYROLL
                      B BY PAY-CODE
                      C BY GROSS-PAY
                      D BY HOURS.
```

In this program, the library text is copied; the resulting text is treated as if it had been written as follows:

```
01  PAYROLL.
    02  PAY-CODE      PIC S99.
    02  GROSS-PAY     PIC S9(5)V99.
    02  HOURS         PIC S9999 OCCURS 1 TO 52 TIMES
            DEPENDING ON PAY-CODE OF PAYROLL.
```

The changes shown are made only for this program. The text, as it appears in the library, remains unchanged.

Example 3

If the following conventions are followed in library text, then parts of names (for example the prefix portion of data names) can be changed with the REPLACING phrase.

In this example, the library text PAYLIB consists of the following data division entries:

```
01  :TAG:.
    02  :TAG:-WEEK      PIC S99.
    02  :TAG:-GROSS-PAY PIC S9(5)V99.
    02  :TAG:-HOURS     PIC S9999 OCCURS 1 TO 52 TIMES
            DEPENDING ON :TAG:-WEEK OF :TAG:.
```

The programmer can use the COPY statement in the data division of a program as follows:

```
COPY PAYLIB REPLACING ==:TAG:== BY ==Payroll==.
```

Usage Note: It is important to notice in this example the required use of colons or parentheses as delimiters in the library text. Colons are recommended for clarity because parentheses can be used for a subscript, for instance in referencing a table element.

In this program, the library text is copied; the resulting text is treated as if it had been written as follows:

```

01 PAYROLL.
02 PAYROLL-WEEK          PIC S99.
02 PAYROLL-GROSS-PAY     PIC S9(5)V99.
02 PAYROLL-HOURS         PIC S999 OCCURS 1 TO 52 TIMES
    DEPENDING ON PAYROLL-WEEK OF PAYROLL.

```

The changes shown are made only for this program. The text, as it appears in the library, remains unchanged.

Example 4

This example shows how to selectively replace level numbers without replacing the numbers in the PICTURE clause:

```

COPY xxx REPLACING ==(01)== BY ==(01)==
                == 01 == BY == 05 ==.

```

DELETE statement

The DELETE statement is an extended source library statement. It removes COBOL statements from a source program that was included by a BASIS statement.

Format

```

▶▶ [sequence-number] DELETE—sequence-number-field ▶◀

```

sequence-number

Can optionally appear in columns 1 through 6, followed by a space. The content of this field is ignored.

DELETE

Can appear anywhere within columns 1 through 72 in fixed source format, or columns 1 through 252 in extended source format. The keyword DELETE must be followed by a space and the *sequence-number-field*. There must be no other text in the statement.

sequence-number-field

Each number must be equal to a *sequence-number* in the BASIS source program. This *sequence-number* is the six-digit number the programmer assigns in columns 1 through 6 of the COBOL coding form. The numbers referenced in the *sequence-number-field* of INSERT or DELETE statements must always be specified in ascending numeric order.

The *sequence-number-field* must be one of the following:

- A single number
- A series of single numbers
- A range of numbers (indicated by separating the two bounding numbers of the range by a hyphen)
- A series of ranges of numbers
- Any combination of one or more single numbers and one or more ranges of numbers

Each entry in the *sequence-number-field* must be separated from the preceding entry by a comma followed by a space. For example:

```
000250 DELETE 000010-000050, 000400, 000450
```

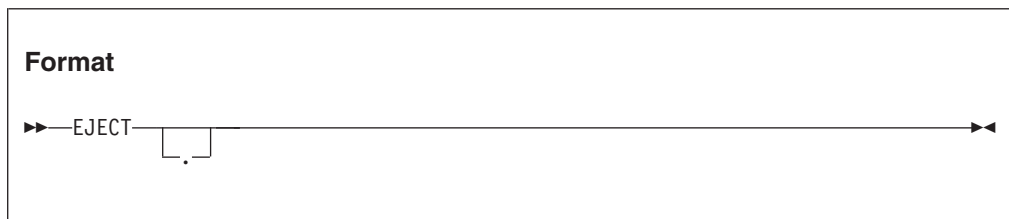
Source program statements can follow a DELETE statement. These source program statements are then inserted into the BASIS source program before the statement following the last statement deleted (that is, in the example above, before the next statement following deleted statement 000450).

If a DELETE statement begins in column 12 or higher and a valid *sequence-number-field* does not follow the keyword DELETE, the compiler assumes that this DELETE statement is a COBOL DELETE statement.

Usage note: If INSERT or DELETE statements are used to modify the COBOL source program provided by a BASIS statement, the sequence field of the COBOL source program must contain numeric sequence numbers in ascending order. The source file remains unchanged. Any INSERT or DELETE statements referring to these sequence numbers must occur in ascending order.

EJECT statement

The EJECT statement specifies that the next source statement is to be printed at the top of the next page.



The EJECT statement must be the only statement on the line. It can be written in either Area A or Area B, and can be terminated with a separator period.

The EJECT statement must be embedded in a program source. For example, in the case of batch applications, the EJECT statement must be placed between the CBL (PROCESS) statement and the end of the program (or the END PROGRAM marker, if specified).

The EJECT statement has no effect on the compilation of the source unit itself.

ENTER statement

The ENTER statement is designed to facilitate the use of more than one source language in the same source program. However, only COBOL is allowed in the source program.

The ENTER statement is syntax checked but has no effect on the execution of the program.

Format

►►—ENTER—*language-name-1*—┐
 └*routine-name-1*┘.—————►►

language-name-1

A system name that has no defined meaning. It must be either a correctly formed user-defined word or the word "COBOL." At least one character must be alphabetic.

routine-name-1

Must follow the rules for formation of a user-defined word. At least one character must be alphabetic.

INSERT statement

The INSERT statement is a library statement that adds COBOL statements to a source program that was included by a BASIS statement.

Format

►►—┐—————INSERT—*sequence-number-field*————►►
 └*sequence-number*┘

sequence-number

Can optionally appear in columns 1 through 6, followed by a space. The content of this field is ignored.

INSERT

Can appear anywhere within columns 1 through 72 in fixed source format, or columns 1 through 252 in extended source format, followed by a space and the *sequence-number-field*. There must be no other text in the statement.

sequence-number-field

A number that must be equal to a *sequence-number* in the BASIS source program. This *sequence-number* is a six-digit number that the programmer assigns in columns 1 through 6 of the COBOL source line.

The numbers referenced in the *sequence-number-field* of INSERT or DELETE statements must always be specified in ascending numeric order.

The *sequence-number-field* must be a single number (for example, 000130). At least one new source program statement must follow the INSERT statement for insertion after the statement number specified by the *sequence-number-field*.

New source program statements following the INSERT statement can include any COBOL syntax.

Usage note: If INSERT or DELETE statements are used to modify the COBOL source program provided by a BASIS statement, the sequence field of the COBOL

source program must contain numeric sequence numbers in ascending order. The source file remains unchanged. Any INSERT or DELETE statements referring to these sequence numbers must occur in ascending order.

READY or RESET TRACE statement

The READY or RESET TRACE statement was designed to trace the execution of procedures. The READY or RESET TRACE statement can appear only in the procedure division, but has no effect on your program.

Format

```
➡ [READY  
   RESET] TRACE—.➡
```

You can trace the execution of procedures by using the USE FOR DEBUGGING declarative as described in *Example: USE FOR DEBUGGING* in the *COBOL for AIX Programming Guide*.

REPLACE statement

The REPLACE statement is used to replace source text.

A REPLACE statement can occur anywhere in the source text that a character-string can occur. It must be preceded by a separator period except when it is the first statement in a separately compiled program. It must be terminated by a separator period.

The REPLACE statement provides a means of applying a change to an entire COBOL compilation group, or part of a compilation group, without manually having to find and modify all places that need to be changed. It is an easy method of doing simple string substitutions. It is similar in action to the REPLACING phrase of the COPY statement, except that it acts on the entire source text, not just on the text in COPY libraries.

If the word REPLACE appears in a comment-entry or in the place where a comment-entry can appear, it is considered part of the comment-entry.

Format 1

```
►►—REPLACE— == —pseudo-text-1— == —BY— == —pseudo-text-2— ==►
```

```
►.——————►
```

Each matched occurrence of *pseudo-text-1* in the source text is replaced by the corresponding *pseudo-text-2*.

Format 2

```
►►—REPLACE OFF.——————►
```

Any text replacement currently in effect is discontinued with the format-2 form of REPLACE. If format 2 is not specified, a given occurrence of the REPLACE statement is in effect from the point at which it is specified until the next occurrence of a REPLACE statement or the end of the separately compiled program.

pseudo-text-1

Must contain one or more text words. Character-strings can be continued in accordance with normal source code rules.

pseudo-text-1 can consist solely of a separator comma or a separator semicolon.

pseudo-text-2

Can contain zero, one, or more text words. Character strings can be continued in accordance with normal source code rules.

pseudo-text-1 and *pseudo-text-2* can contain any text words (except the word COPY) that can be written in source text, including national literals, DBCS literals, and multibyte user-defined words.

Characters outside those allowed for COBOL words and separators must not appear in library text or pseudo-text except in comment lines, comment-entries, alphanumeric literals, DBCS literals, or national literals.

Multibyte user-defined words must be wholly formed; that is, there is no partial-word replacement for multibyte words.

pseudo-text-1 and *pseudo-text-2* can contain single-byte and multibyte characters in comment lines and comment entries.

Individual character-strings within pseudo-text can be up to 322 characters long, except that strings containing multibyte characters cannot be continued.

The compiler processes REPLACE statements in source text after the processing of any COPY statements. COPY must be processed first, to assemble complete source text. Then REPLACE can be used to modify that source text, performing simple string substitution. REPLACE statements cannot themselves contain COPY statements.

The text produced as a result of the processing of a REPLACE statement must not contain a REPLACE statement.

Continuation rules for pseudo-text

The character-strings and separators comprising pseudo-text can start in either Area A or Area B. If, however, there is a hyphen in the indicator area of a line that follows the opening pseudo-text delimiter, Area A of the line must be blank; and the normal rules for continuation of lines apply to the formation of text words. (See "Continuation lines" on page 48.)

Comparison operation

The comparison operation to determine text replacement starts with the leftmost source text word following the REPLACE statement, and with the first *pseudo-text-1*. *pseudo-text-1* is compared to an equivalent number of contiguous source text words. *pseudo-text-1* matches the source text if, and only if, the ordered sequence of text words that forms *pseudo-text-1* is equal, character for character, to the ordered sequence of source text words.

For purposes of matching, each occurrence of a separator comma, semicolon, or space, and each sequence of one or more space separators is considered to be a single space.

However, when *pseudo-text-1* consists solely of a separator comma or semicolon, the comma or semicolon participates in the match as a text word (in this case, the space following the comma or semicolon separator can be omitted).

If no match occurs, the comparison is repeated with each successive occurrence of *pseudo-text-1*, until either a match is found or there is no next successive occurrence of *pseudo-text-1*.

When all occurrences of *pseudo-text-1* have been compared and no match has occurred, the next successive source text word is then considered as the leftmost source text word, and the comparison cycle starts again with the first occurrence of *pseudo-text-1*.

Whenever a match occurs between *pseudo-text-1* and the source text, the corresponding *pseudo-text-2* replaces the matched text in the source text. The source text word immediately following the rightmost text word that participated in the match is then considered as the leftmost source text word. The comparison cycle starts again with the first occurrence of *pseudo-text-1*.

The comparison operation continues until the rightmost text word in the source text that is within the scope of the REPLACE statement has either participated in a match or been considered as a leftmost source text word and participated in a complete comparison cycle.

REPLACE statement notes

Comment lines, inline comments, or blank lines occurring in the source text and in *pseudo-text-1* are ignored for purposes of matching. The sequence of text words in the source text and in *pseudo-text-1* is determined by the rules for reference format (see Chapter 6, "Reference format," on page 45). Comment lines, inline comments, or blank lines in *pseudo-text-2* are placed into the resultant program unchanged whenever *pseudo-text-2* is placed into the source text as a result of text replacement. Comment lines, inline comments, or blank lines appearing in source text are retained unchanged with the following exception: a comment line, an inline comment, or a blank line in source text is not retained if that comment line, inline comment, or blank line appears within the sequence of text words that match *pseudo-text-1*.

Lines containing *CONTROL (*CBL), EJECT, SKIP1/2/3, or TITLE statements can occur in source text. Such lines are treated as comment lines during REPLACE statement processing.

Debugging lines are permitted in pseudo-text. Text words within a debugging line participate in the matching rules as if the letter "D" did not appear in the indicator area.

When a REPLACE statement is specified on a debugging line, the statement is treated as if the letter "D" did not appear in the indicator area.

After all COPY and REPLACE statements have been processed, a debugging line will be considered to have all the characteristics of a comment line if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph.

Except for COPY and REPLACE statements, the syntactic correctness of the source text cannot be determined until after all COPY and REPLACE statements have been completely processed.

Text words inserted into the source text as a result of processing a REPLACE statement are placed in the source text according to the rules for reference format. When inserting text words of *pseudo-text-2* into the source text, additional spaces are introduced only between text words where there already exists a space (including the assumed space between source lines).

If additional lines are introduced into the source text as a result of the processing of REPLACE statements, the indicator area of the introduced lines contains the same character as the line on which the text being replaced begins, unless that line contains a hyphen, in which case the introduced line contains a space.

If any literal within *pseudo-text-2* is of a length too great to be accommodated on a single line without continuation to another line in the resultant program and the literal is not being placed on a debugging line, additional continuation lines are introduced that contain the remainder of the literal. If replacement requires the continued literal to be continued on a debugging line, the program is in error.

Each word in *pseudo-text-2* that is to be placed into the resultant program begins in the same area of the resultant program as it appears in *pseudo-text-2*.

SERVICE LABEL statement

The SERVICE LABEL statement is syntax checked, but has no effect on the execution of the program.

Format

»—SERVICE LABEL—————»

The SERVICE LABEL statement can appear only in the procedure division, but not in the declaratives section.

SERVICE RELOAD statement

The SERVICE RELOAD statement is syntax checked, but has no effect on the execution of the program.

Format

»—SERVICE RELOAD—*identifier-1*—————»

SKIP statements

The SKIP1, SKIP2, and SKIP3 statements specify blank lines that the compiler should add when printing the source listing. SKIP statements have no effect on the compilation of the source text itself.

Format

»—

SKIP1	┌┐	┌┐	┌┐
SKIP2			
SKIP3			

—————»

SKIP1 Specifies a single blank line to be inserted in the source listing.

SKIP2 Specifies two blank lines to be inserted in the source listing.

SKIP3 Specifies three blank lines to be inserted in the source listing.


SKIP1, SKIP2, or SKIP3 can be written anywhere in either Area A or Area B, and can be terminated with a separator period. It must be the only statement on the line.

The SKIP statements must be embedded in a program source. For example, in the case of batch applications, a SKIP1, SKIP2, or SKIP3 statement must be placed between the CBL (PROCESS) statement and the end of the program or class (or the END CLASS marker or END PROGRAM marker, if specified).

TITLE statement

The TITLE statement specifies a title to be printed at the top of each page of the source listing produced during compilation. If no TITLE statement is found, a title containing the identification of the compiler and the current release level is generated. The title is left-justified on the title line.

Format

►►—TITLE—*literal*—►►

literal Must be an alphanumeric literal, DBCS literal, or national literal and can be followed by a separator period.

Must not be a figurative constant.

In addition to the default or chosen title, the right side of the title line contains the following:

- For programs, the name of the program from the PROGRAM-ID paragraph for the outermost program. (This space is blank on pages preceding the PROGRAM-ID paragraph for the outermost program.)
- For classes, the name of the class from the CLASS-ID paragraph.
- Current page number.
- Date and time of compilation.

The TITLE statement:

- Forces a new page immediately, if the SOURCE compiler option is in effect
- Is not itself printed on the source listing
- Has no other effect on compilation
- Has no effect on program execution
- Cannot be continued on another line
- Can appear anywhere in any of the divisions

A title line is produced for each page in the listing produced by the LIST option. This title line uses the last TITLE statement found in the source statements or the default.

The word TITLE can begin in either Area A or Area B.

The TITLE statement must be embedded in a class or program source. For example, in the case of batch applications, the TITLE statement must be placed between the CBL (PROCESS) statement and the end of the class or program (or the END CLASS marker or END PROGRAM marker, if specified).

No other statement can appear on the same line as the TITLE statement.

USE statement

The formats for the USE statement are:

- EXCEPTION/ERROR declarative
- LABEL declarative
- DEBUGGING declarative

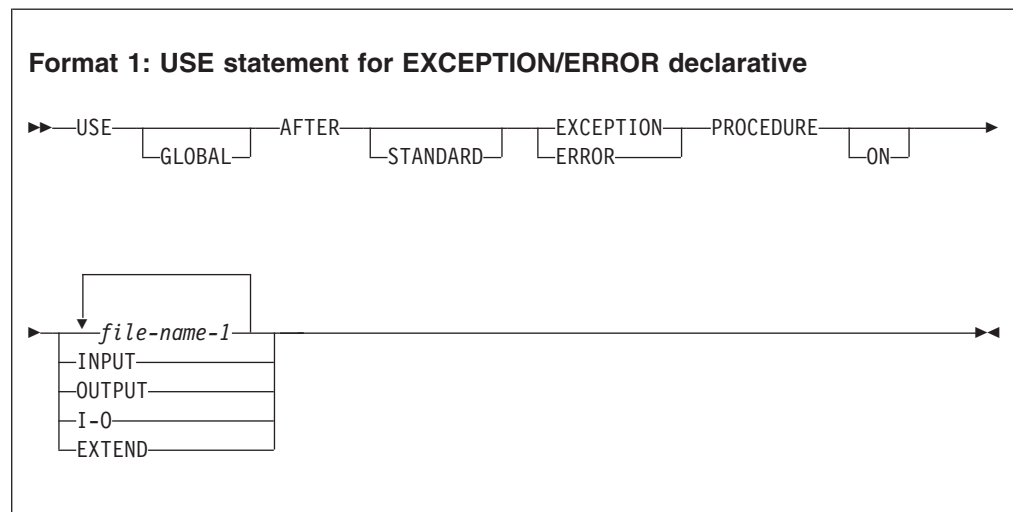
For general information about declaratives, see “Declaratives” on page 255.

LABEL declaratives are supported by IBM Enterprise COBOL for z/OS.

EXCEPTION/ERROR declarative

The EXCEPTION/ERROR declarative specifies procedures for input/output exception or error handling that are to be executed in addition to the standard system procedures.

The words EXCEPTION and ERROR are synonymous and can be used interchangeably.



file-name-1

Valid for all files. When this option is specified, the procedure is executed only for the files named. No file-name can refer to a sort or merge file. For any given file, only one EXCEPTION/ERROR procedure can be specified; thus, file-name specification must not cause simultaneous requests for execution of more than one EXCEPTION/ERROR procedure.

A USE AFTER EXCEPTION/ERROR declarative statement specifying the name of a file takes precedence over a declarative statement specifying the open mode of the file.

INPUT

Valid for all files. When this option is specified, the procedure is executed for all files opened in INPUT mode or in the process of being opened in INPUT mode that get an error.

OUTPUT

Valid for all files. When this option is specified, the procedure is executed for all files opened in OUTPUT mode or in the process of being opened in OUTPUT mode that get an error.

I-O Valid for all direct-access files. When this option is specified, the procedure is executed for all files opened in I-O mode or in the process of being opened in I-O mode that get an error.

EXTEND

Valid for all files. When this option is specified, the procedure is executed for all files opened in EXTEND mode or in the process of being opened in EXTEND mode that get an error.

The EXCEPTION/ERROR procedure is executed:

- Either after completing the system-defined input/output error routine, or
- Upon recognition of an INVALID KEY or AT END condition when an INVALID KEY or AT END phrase has not been specified in the input/output statement, or
- Upon recognition of an IBM-defined condition that causes file status key 1 to be set to 9. (See “File status key” on page 297.)

After execution of the EXCEPTION/ERROR procedure, control is returned to the invoking routine in the input/output control system. If the input/output status value does not indicate a critical input/output error, the input/output control system returns control to the next executable statement following the input/output statement whose execution caused the exception.

An applicable EXCEPTION/ERROR procedure is activated when an input/output error occurs during execution of a READ, WRITE, REWRITE, START, OPEN, CLOSE, or DELETE statement. To determine what conditions are errors, see “Common processing facilities” on page 296.

A declarative procedure must not reference a nondeclarative procedure.

A PERFORM statement in a nondeclarative procedure can reference a declarative procedure; otherwise, a declarative procedure must not be referenced from a nondeclarative procedure.

You can include a statement that executes a previously called USE procedure that is still in control. However, to avoid an infinite loop, you must be sure that there is an eventual exit at the bottom.

EXCEPTION/ERROR procedures can be used to check the file status key values whenever an input/output error occurs.

Precedence rules for nested programs

Special precedence rules are followed when programs are contained within other programs. In applying these rules, only the first qualifying declarative is selected for execution. The order of precedence for selecting a declarative is:

1. A file-specific declarative (that is, a declarative of the form USE AFTER ERROR ON *file-name-1*) within the program that contains the statement that caused the qualifying condition.

2. A mode-specific declarative (that is, a declarative of the form USE AFTER ERROR ON INPUT) within the program that contains the statement that caused the qualifying condition.
3. A file-specific declarative that specifies the GLOBAL phrase and is within the program directly containing the program that was last examined for a qualifying declarative.
4. A mode-specific declarative that specifies the GLOBAL phrase and is within the program directly containing the program that was last examined for a qualifying condition.

Steps 3 and 4 are repeated until the last examined program is the outermost program, or until a qualifying declarative has been found.

LABEL declarative

The LABEL declarative (format 2 of the USE statement) is supported by IBM Enterprise COBOL for z/OS. If a LABEL declarative is encountered by COBOL for AIX, a warning message is issued and the declarative is ignored.

DEBUGGING declarative

Debugging sections are permitted only in the outermost program; they are not valid in nested programs. Debugging sections are never triggered by procedures contained in nested programs.

Debugging sections are not permitted in:

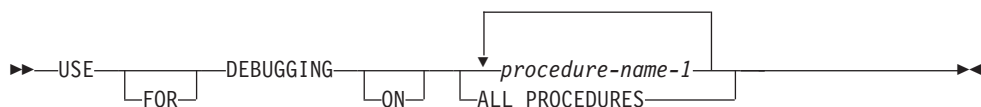
- A method
- A program defined with the recursive attribute
- A program compiled with the THREAD compiler option

The WITH DEBUGGING MODE clause of the SOURCE-COMPUTER paragraph activates all debugging sections and lines that have been compiled into the object code. See Appendix E, “Source language debugging,” on page 593 for additional details.

When the debugging mode is suppressed by not specifying the WITH DEBUGGING MODE clause, all USE FOR DEBUGGING declarative procedures and all debugging lines are inhibited.

Automatic execution of a debugging section is not caused by a statement that appears in a debugging section.

Format 3: USE statement for DEBUGGING declarative



USE FOR DEBUGGING

All debugging statements must be written together in a section immediately after the DECLARATIVES header.

Except for the USE FOR DEBUGGING sentence itself, within the debugging procedure there must be no reference to any nondeclarative procedures.

procedure-name-1

Must not be defined in a debugging session.

Table 59 shows, for each valid option, the points during execution when the USE FOR DEBUGGING procedures are executed.

Any given procedure-name can appear in only one USE FOR DEBUGGING sentence, and only once in that sentence. All procedures must appear in the outermost program.

ALL PROCEDURES

procedure-name-1 must not be specified in any USE FOR DEBUGGING sentences. The ALL PROCEDURES phrase can be specified only once in a program. Only the procedures contained in the outermost program will trigger execution of the debugging section.

Table 59. Execution of debugging declaratives

USE FOR DEBUGGING operand	Upon execution of the following, the USE FOR DEBUGGING procedures are executed immediately
<i>procedure-name-1</i>	Before each execution of the named procedure After the execution of an ALTER statement referring to the named procedure
ALL PROCEDURES	Before each execution of each nondebugging procedure in the outermost program After the execution of each ALTER statement in the outermost program (except ALTER statements in declarative procedures)

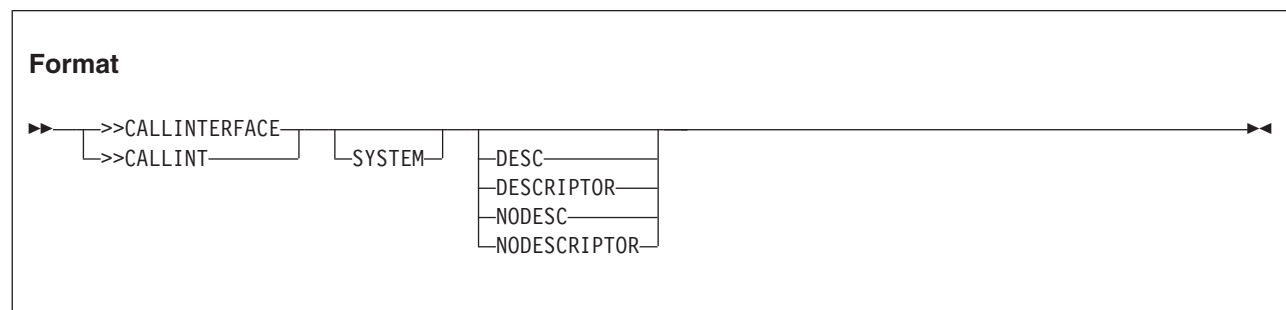
Chapter 24. Compiler directives

A *compiler directive* is a statement that causes the compiler to take a specific action during compilation.

Currently, CALLINTERFACE is the only compiler directive supported. See “CALLINTERFACE.”

CALLINTERFACE

The CALLINTERFACE directive specifies the interface convention for CALL statements. The convention specified stays in effect until another CALLINTERFACE specification is made in the compilation unit.



SYSTEM

Specifies that the call interface convention is the standard system linkage convention of the platform.

DESC, DESCRIPTOR

Indicates that an argument descriptor is passed for each argument on a CALL statement.

NODESC, NODESCRIPTOR

Indicates that no argument descriptors are passed for any arguments on a CALL statement. NODESC/NODESCRIPTOR is the default.

Specify >>CALLINTERFACE only in the procedure division.

The positions of CALL statements relative to the CALLINTERFACE directive are recognized following any processing of COPY and REPLACE statements. For example, CALL statements and CALLINTERFACE directives in COPY text are processed by the rules specified for the CALLINTERFACE directive.

If you specify syntax in the CALLINTERFACE directive that is not supported, the entire CALLINTERFACE directive is ignored.

Syntax and general rules

- You must specify >>CALLINTERFACE on a line by itself, in Area B.
- You cannot specify >>CALLINTERFACE:
 - Within a COPY or REPLACE statement

- Between the lines of a continued character string
 - On a debugging line
 - In the middle of a COBOL statement
- The >>CALLINTERFACE specification is limited to the current compilation unit.
- The REPLACE statement and REPLACING phrase of the COPY statement do not affect the CALLINTERFACE directive.

Part 9. Appendixes

Appendix A. IBM extensions

IBM extensions are features, syntax rules, or behavior defined by IBM rather than by the COBOL standards.

Table 60 lists IBM extensions with a brief description. Standard behavior is shown in brackets, [], when the standard behavior is not obvious. Extensions are described in more detail throughout this document, but they are not further identified as extensions.

Many IBM extensions are distinguished from standard language by their syntax. For others, you use compiler options to choose between standard and extension behavior. Generally, the related compiler options are noted in the detailed rules. For information about compiler options, see *Compiler options* in the *COBOL for AIX Programming Guide*.

If an item is listed as an extension, all related rules are also extensions. For example, USAGE DISPLAY-1 for DBCS characters is listed as an extension; its many uses in statements and clauses are also extensions, but are not listed separately.

Table 60. IBM extension language elements

Language area	Extension elements
COBOL words	User-defined words written in multibyte characters System-names written in multibyte characters Class-names (for object orientation) Method-names User-defined words can include an underscore, but not as the first character.
National character support (Unicode support)	Support for UTF-16 with USAGE NATIONAL Allowance of UTF-8 with USAGE DISPLAY Usage NATIONAL for data categories national, national-edited, numeric, numeric-edited, external decimal, and external floating-point GROUP-USAGE clause with the NATIONAL phrase National literals (basic and hexadecimal) National character value for figurative constants SPACE, ZERO, QUOTE, HIGH-VALUES, LOW-VALUES, ALL literal Intrinsic functions for data conversion: <ul style="list-style-type: none">• DISPLAY-OF• NATIONAL-OF Extended case mapping with UPPER-CASE and LOWER-CASE functions

Table 60. IBM extension language elements (continued)

Language area	Extension elements
Implicit items	<p>Special object references:</p> <ul style="list-style-type: none"> • SELF • SUPER <p>Special registers:</p> <ul style="list-style-type: none"> • ADDRESS OF • JNENVPTR • LENGTH OF • RETURN-CODE • SHIFT-IN • SHIFT-OUT • SORT-CONTROL • SORT-CORE-SIZE • SORT-FILE-SIZE • SORT-MESSAGE • SORT-MODE-SIZE • SORT-RETURN • TALLY • WHEN-COMPILED • XML-CODE • XML-EVENT • XML-NAMESPACE • XML-NAMESPACE-PREFIX • XML-NNAMESPACE • XML-NNAMESPACE-PREFIX • XML-NTEXT • XML-TEXT
Figurative constants	<p>Selection of apostrophe (') as the value of the figurative constant QUOTE</p> <p>NULL and NULLS for pointers and object references</p>

Table 60. IBM extension language elements (continued)

Language area	Extension elements
Literals	<p>Use of apostrophe (') as an alternative to the quotation mark (") in opening and closing delimiters</p> <p>Mixed single-byte and multibyte characters in alphanumeric literals (mixed literals)</p> <p>Hexadecimal notation for alphanumeric literals, defined by opening delimiters X" and X'</p> <p>Null-terminated alphanumeric literals, defined by opening delimiters Z" and Z'</p> <p>DBCS literals, defined by opening delimiters N", N', G", and G'. N" and N' are defined as DBCS when the NSYMBOL(DBCS) compiler option is in effect.</p> <p>Consecutive alphanumeric literals (coding two consecutive alphanumeric literals by ending the first literal in column 72 of a continued line and starting the next literal with a single quotation mark in the continuation line)</p> <p>National literals N", N', NX", NX' for storing literal content as national characters. N" and N' are defined as national when the NSYMBOL(NATIONAL) compiler option is in effect.</p> <p>19- to 31-digit fixed-point numeric literals. [Standard COBOL 85 specifies a maximum of 18 digits.]</p> <p>Floating-point numeric literals</p>
Comments	<p>Comment lines before the identification division header</p> <p>Comment lines and comment entries containing multibyte characters</p> <p>Inline comments</p>
End markers	<p>The following end markers:</p> <ul style="list-style-type: none"> • END CLASS • END FACTORY • END METHOD • END OBJECT
Indexing and subscripting	<p>Referencing a table with an index-name defined for a different table</p> <p>Specifying a positive signed integer literal following the operator + or - in relative subscripting</p>
Millennium language extensions and date fields.	<p>DATE FORMAT clause</p> <p>Windowed date fields, expanded date fields, year-last date fields, compatible date fields, date formats, and century window</p> <p>The following intrinsic functions:</p> <ul style="list-style-type: none"> • DATEVAL • UNDATE • YEARWINDOW • DATE-TO-YYYYMMDD • DAY-TO-YYYYDDD • YEAR-TO-YYYY
Reference format	Extended source format

Table 60. IBM extension language elements (continued)

Language area	Extension elements
Identification division for programs	<p>Abbreviation ID for IDENTIFICATION</p> <p>RECURSIVE clause</p> <p>An optional separator period following PROGRAM-ID, AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY paragraph headers. [Standard COBOL 85 requires a period following each of these paragraph headers.]</p> <p>An optional separator period following program-name in the PROGRAM-ID paragraph. [Standard COBOL 85 requires a period following program-name.]</p> <p>An alphanumeric literal for program-name in the PROGRAM-ID paragraph; the underscore can be the first character; program-name up to 160 characters in length. [Standard COBOL 85 requires that program-name be specified as a user-defined word.]</p>
End markers	<p>Program-name in a literal. [Standard COBOL 85 requires that program-name be specified as a user-defined word.]</p>
Object-oriented structure	<p>In a class definition:</p> <ul style="list-style-type: none"> • CLASS-ID paragraph • INHERITS clause • END CLASS marker <p>In a method definition:</p> <ul style="list-style-type: none"> • METHOD-ID paragraph • EXIT METHOD statement • END METHOD marker
Configuration section	<p>Repository paragraph</p>
SPECIAL-NAMES paragraph	<p>The optional order of clauses. [Standard COBOL 85 requires that the clauses be coded in the order presented in the syntax diagram.]</p> <p>Optionality of a period after the last clause when no clauses are coded. [Standard COBOL 85 requires a period, even when no clauses are coded.]</p> <p>Multiple CURRENCY SIGN clauses. [Standard COBOL 85 allows a single CURRENCY SIGN clause.]</p> <p>WITH PICTURE SYMBOL phrase in the CURRENCY SIGN clause</p> <p>Multiple-character and mixed-case currency signs in the CURRENCY SIGN clause (when the WITH PICTURE SYMBOL phrase is specified). [Standard COBOL 85 allows only one character, and it is both the currency sign and the currency picture symbol. The standard currency sign must not be:</p> <ul style="list-style-type: none"> • The same character as any standard picture symbol • A digit 0-9 • One of the special characters * + - , ; () " = / • A space] <p>Use of lower-case alphabetic characters as a currency sign. [Standard COBOL 85 allows only uppercase characters.]</p> <p>Disallowance of the SYMBOLIC CHARACTERS clause, when a multibyte code page is indicated by the locale setting.</p>

Table 60. IBM extension language elements (continued)

Language area	Extension elements
INPUT-OUTPUT SECTION, FILE-CONTROL paragraph	<p>Optionality of "FILE-CONTROL." when the INPUT-OUTPUT SECTION is specified, no file-control-paragraph is specified, and there are no files defined in the compilation unit. [Standard COBOL 85 requires that "FILE-CONTROL." be coded if "INPUT-OUTPUT SECTION." is coded.]</p> <p>Optionality of the file-control-paragraph when the "FILE CONTROL." syntax is specified and there are no files defined in the compilation unit. [Standard COBOL 85 requires that a file-control-paragraph be coded if "INPUT-OUTPUT SECTION." is coded.]</p> <p>USING phrase of the ASSIGN clause</p> <p>PASSWORD clause</p> <p>The second <i>data-name</i> in the FILE STATUS clause</p> <p>Optionality of RECORD in the ALTERNATE RECORD KEY clause. [Standard COBOL 85 requires the word RECORD.]</p> <p>The lack of a RESERVE clause effect on execution</p> <p>A numeric, numeric-edited, alphanumeric-edited, alphabetic, internal floating-point, external floating-point, national, national-edited, or DBCS primary or alternate record key data item. [Standard COBOL 85 requires that the key be alphanumeric.]</p> <p>A primary or alternate record key defined outside the minimum record size for indexed files containing variable-length records. [Standard COBOL 85 requires that the primary and alternate record keys be within the minimum record size.]</p> <p>Sequential access to records in descending order. [Standard COBOL 85 provides only ascending order of access.]</p> <p>A numeric data item of usage DISPLAY or NATIONAL in the FILE STATUS clause. [Standard COBOL 85 requires an alphanumeric file status data item.]</p> <p>ORGANIZATION IS LINE SEQUENTIAL clause and line-sequential file control format</p> <p>National literal in the PADDING CHARACTER clause</p>
INPUT-OUTPUT SECTION, I-O-CONTROL paragraph	<p>APPLY WRITE-ONLY clause</p> <p>Specifying only one file-name in the SAME clause in the sequential, indexed, and sort-merge formats of the I-O-control entry. [Standard COBOL 85 requires at least two file-names.]</p> <p>Optionality of the keyword ON in the RERUN clause. [Standard COBOL 85 requires that ON be coded.]</p> <p>The line-sequential format I-O-control entry</p> <p>The RERUN clause in the sort-merge I-O-control entry</p> <p>Disallowance of the RERUN clause when using the THREAD compiler option</p> <p>The lack of RERUN clause effect on execution when using the NOTHREAD compiler option</p>

Table 60. IBM extension language elements (continued)

Language area	Extension elements
DATA DIVISION	<p>LOCAL-STORAGE SECTION</p> <p>The GLOBAL clause in the linkage section</p> <p>Specifying level numbers that are lower than other level numbers at the same hierarchical level in a data description entry. [Standard COBOL 85 requires that all elementary or group items at the same level in the hierarchy be assigned identical level numbers.]</p> <p>Data categories internal floating-point, external floating-point, DBCS, national, and national-edited.</p> <p>Data category numeric with usage NATIONAL.</p> <p>Data category numeric-edited with usage NATIONAL.</p>
File section	<p><i>data-name</i> in the LABEL RECORDS clause, for specifying user labels</p> <p>RECORDING MODE clause</p> <p>Line-sequential format file description entry</p>
Sort/merge file description entry	<p>The following clauses:</p> <ul style="list-style-type: none"> • BLOCK CONTAINS • LABEL RECORDS • VALUE OF • LINAGE • CODE-SET • WITH FOOTING • LINES AT
VALUE OF clause	The lack of VALUE clause effect on execution when specified under an SD
DATA RECORDS clause	Optionality of an 01 record description entry for a specified <i>data-name</i> . [Standard COBOL 85 requires that an 01 record with the same <i>data-name</i> be specified.]
LINAGE clause	Specifying LINAGE for files opened in EXTEND mode
Data Description Entry	DATE-FORMAT clause
BLANK WHEN ZERO clause	Alternative spellings ZEROS and ZEROES for ZERO
GLOBAL clause	Specifying GLOBAL in the linkage section
INDEXED BY phrase	Nonunique unreferenced index names

Table 60. IBM extension language elements (continued)

Language area	Extension elements
OCCURS clause	<p>Omission of "<i>integer-1</i> TO" for variable-length tables</p> <p>Complex OCCURS DEPENDING ON. [Standard COBOL 85 requires that an entry containing OCCURS DEPENDING ON be followed only by subordinate entries, and that no entry containing OCCURS DEPENDING ON be subordinate to an entry containing OCCURS DEPENDING ON.]</p> <p>Implicit qualification of a key specified without qualifiers when the key name is not unique</p> <p>Reference to a table through indexing when no INDEXED BY phrase is specified</p> <p>Keys of usages COMPUTATIONAL-1, COMPUTATIONAL-2, COMPUTATIONAL-3, COMPUTATIONAL-4, and COMPUTATIONAL-5 in the ASCENDING/DESCENDING KEY phrase</p> <p>Acceptance of nonunique index-names that are not referenced</p>
PICTURE clause	<p>A picture character-string containing 31 to 50 characters. [Standard COBOL 85 allows a maximum of 30 characters.]</p> <p>Picture symbols G and N</p> <p>Picture symbol E and the external floating-point picture format</p> <p>Coding a trailing comma insertion character or trailing period insertion character immediately followed by a separator comma or separator semicolon in a PICTURE clause that is not the last clause of a data description entry. [Standard COBOL 85 requires that a PICTURE clause containing a picture ending with a comma or period be the last clause in the entry and that it be followed immediately by a separator period.]</p> <p>Selecting a currency sign and currency symbol with the CURRENCY compiler option</p> <p>Case-sensitive currency symbols</p> <p>The maximum of 31 digits for numeric items of usages DISPLAY and PACKED-DECIMAL and for numeric-edited items of USAGE DISPLAY</p> <p>The effect of the TRUNC compiler option on the value of data items described with a usage of BINARY, COMPUTATIONAL, or COMPUTATIONAL-4</p>
REDEFINES clause	<p>Specifying REDEFINES of a redefined data item</p> <p>At a subordinate level, specifying a redefining data item that has a size greater than the size of the redefined data item</p>
SYNCHRONIZED clause	Specifying SYNCHRONIZED for a level 01 entry

Table 60. IBM extension language elements (continued)

Language area	Extension elements
USAGE clause	<p>The following phrases:</p> <ul style="list-style-type: none"> • NATIVE • COMP-1 and COMPUTATIONAL-1 • COMP-2 and COMPUTATIONAL-2 • COMP-3 and COMPUTATIONAL-3 • COMP-4 and COMPUTATIONAL-4 • COMP-5 and COMPUTATIONAL-5 • DISPLAY-1 • OBJECT REFERENCE • NATIONAL • POINTER • PROCEDURE-POINTER • FUNCTION-POINTER <p>Use of the SYNCHRONIZED clause for items of usage INDEX</p>
VALUE clause for condition-name entries	<p>A VALUE clause in file and linkage sections in other than condition-name entries</p> <p>A VALUE clause for a condition-name entry on a group that has usages other than DISPLAY</p> <p>Specifying the range of values in a THROUGH phrase using a locale-defined collating sequence</p> <p>VALUE IS NULL and VALUE IS NULLS</p>
Procedure division	<p>Omission of a section-name</p> <p>Omission of a paragraph-name when a section-name is omitted</p> <p>A method, factory, or object procedure division</p> <p>Referencing data items in the linkage section without a USING phrase in the procedure division header (when those data-names are the operand of an ADDRESS OF phrase or ADDRESS OF special register)</p> <p>The following statements:</p> <ul style="list-style-type: none"> • ENTRY • EXIT METHOD • GOBACK • INVOKE • XML PARSE • XML GENERATE
Procedure division header	<p>The BY VALUE phrase</p> <p>The RETURNING phrase</p> <p>Specifying a data item in the USING phrase when the data item has a REDEFINES clause in its description</p> <p>Specifying multiple instances of a given data item in the USING phrase</p> <p>The formats for method, factory, and object definitions</p>

Table 60. IBM extension language elements (continued)

Language area	Extension elements
Declarative Procedures	<p>The LABEL declarative</p> <p>Executing an active declarative</p>
Procedures	<p>Specifying <i>priority-number</i> as a positive signed numeric literal. [Standard COBOL 85 requires an unsigned integer.]</p> <p>Omitting the section-header after the declaratives or when there are no declaratives. [Standard COBOL 85 requires a section-header following the "DECLARATIVES." syntax and following the "END DECLARATIVES." syntax.]</p> <p>Omitting an initial <i>paragraph-name</i> if there are no declaratives. [Standard COBOL 85 requires a paragraph-name in the following circumstances:</p> <ul style="list-style-type: none"> • After the USE statement if there are statements in the declarative procedure • Following a section header outside declarative procedures • Before any procedural statement if there are no declaratives <p>and Standard COBOL 85 requires that procedural statements be within a paragraph.]</p> <p>Specifying paragraphs that are not contained within a section, even if some paragraphs are so contained. [Standard COBOL 85 requires that paragraphs be within a section except when there are no declaratives. Standard COBOL 85 requires that either all paragraphs be in sections or that none be.]</p>
Conditional expressions	<p>DBCS and KANJI class conditions</p> <p>Specifying data items of usage COMPUTATIONAL-3 or usage PACKED-DECIMAL in a NUMERIC class test</p>
Relation condition	<p>Enclosing an alphanumeric, DBCS, or national literal in parentheses</p> <p>The data-pointer format, the procedure-pointer and function-pointer format, and the object-reference format</p> <p>Comparison of an index-name with an arithmetic expression</p> <p>Use of parentheses within abbreviated combined relation conditions</p>
CORRESPONDING phrase	Specifying an identifier that is subordinate to a filler item
INVALID KEY phrase	Omission of both the INVALID KEY phrase and an applicable EXCEPTION/ERROR procedure. [Standard COBOL 85 requires at least one of them.]
ACCEPT statement	<p>The <i>environment-name</i> operand of the FROM phrase</p> <p>The DATE YYYYMMDD phrase</p> <p>The DAY YYYYDDD phrase</p>
ADD statement	A composite of operands greater than 18 digits

Table 60. IBM extension language elements (continued)

Language area	Extension elements
CALL statement	<p>The procedure-pointer and function-pointer operands for identifying the program to be called</p> <p>The following phrases:</p> <ul style="list-style-type: none"> • ADDRESS OF • LENGTH OF • OMITTED • BY VALUE • RETURNING <p>Specifying the called program-name in an alphabetic or zoned-decimal data item</p> <p>Specifying an argument defined as a subordinate group item. [Standard COBOL 85 requires that arguments be an elementary data item or a group item defined with level 01.]</p>
CANCEL statement	<p>Specifying the name of the program to be canceled in an alphabetic or zoned-decimal data item</p> <p>The effect of the PGMNAME compiler option on the name of the program to be canceled</p>
CLOSE statement	<p>WITH NO REWIND phrase</p> <p>The line-sequential format</p>
COMPUTE statement	The use of the word EQUAL in place of the equal sign (=)
DISPLAY statement	<p>The <i>environment-name</i> operand of the UPON phrase</p> <p>Displaying signed numeric literals and noninteger numeric literals</p>
DIVIDE statement	A composite of operands greater than 18 digits
EXIT statement	Specifying the EXIT statement in a sentence that has statements before or after the EXIT statement or in a paragraph that has other sentences. [Standard COBOL 85 requires that the EXIT statement be specified in a sentence by itself and that the sentence be the only sentence in the paragraph.]
EXIT PROGRAM statement	Specifying EXIT PROGRAM before the last statement in a sequence of imperative statements. [Standard COBOL 85 requires that the EXIT PROGRAM statement be specified as the last statement in a sequence of imperative statements.]
GO TO statement	<p>Coding the unconditional format before the last statement in a sequence of imperative statements. [Standard COBOL 85 requires that an unconditional GO TO be coded:</p> <ul style="list-style-type: none"> • Only in a single-statement paragraph if no procedure-name is specified • Otherwise, as the last statement of a sentence.] <p>The MORE-LABELS format</p>
IF statement	The use of END-IF with the NEXT SENTENCE phrase. [Standard COBOL 85 disallows use of END-IF with NEXT SENTENCE.]
INITIALIZE statement	<p>DBCS, EGCS, NATIONAL, and NATIONAL-EDITED in the REPLACING phrase</p> <p>Initializing a data item that contains the DEPENDING phrase of the OCCURS clause</p>
MERGE statement	Specifying file-names in a SAME clause
MULTIPLY statement	A composite of operands greater than 18 digits
OPEN statement	<p>The line-sequential format</p> <p>Specifying the EXTEND phrase for files that have a LINAGE clause</p>

Table 60. IBM extension language elements (continued)

Language area	Extension elements
PERFORM statement	An empty in-line PERFORM statement A common exit for two or more active PERFORMS
READ statement	PREVIOUS phrase Omission of both the AT END phrase and an applicable declarative procedure Omission of both the INVALID KEY phrase and an applicable declarative procedure Read into an item that is neither an alphanumeric group item nor an elementary alphanumeric item
RETURN statement	Return into an item that is neither an alphanumeric group item nor an elementary alphanumeric item
REWRITE statement	Omission of both the INVALID KEY phrase and an applicable declarative procedure Rewriting a record with a different number of character positions than the number of character positions in the record being rewritten
SEARCH statement	Specifying END SEARCH with NEXT SENTENCE Omission of both the NEXT SENTENCE phrase and imperative statements in the binary search format
SET statement	The data-pointer format The procedure-pointer and function-pointer format The object reference format
SORT statement	Specifying GIVING file-names in the SAME clause
START statement	Omission of both the INVALID KEY phrase and an applicable exception procedure Use of a key of a category other than alphanumeric The following relational operators: <ul style="list-style-type: none"> • LESS THAN • < • NOT GREATER THAN • NOT > • LESS THAN OR EQUAL TO • <=
STOP statement	Specifying a noninteger fixed-point literal or a signed numeric integer or noninteger fixed-point literal Coding STOP as other than the last statement in a sentence
STRING statement	Reference modification of the data item specified in the INTO phrase
SUBTRACT statement	A composite of operands greater than 18 digits
UNSTRING statement	Reference modification of the sending field

Table 60. IBM extension language elements (continued)

Language area	Extension elements
WRITE statement	<p>INVALID KEY and NOT ON INVALID KEY phrases</p> <p>The line-sequential format</p> <p>For a relative file, writing a different number of character positions than the number of character positions in the record being replaced</p> <p>Specifying both the ADVANCING PAGE and END-OF-PAGE phrases in a single WRITE statement</p> <p>For a relative or indexed file, omission of both the INVALID KEY phrase and an applicable exception procedure</p>
Intrinsic functions	<p>The effect of the DATEPROC and INTDATE compiler options on the DATE-OF-INTEGER and DAY-OF-INTEGER functions</p> <p>The following functions:</p> <ul style="list-style-type: none"> • DATE-TO-YYYYMMDD • DATEVAL • DAY-TO-YYYYDDD • DISPLAY-OF • NATIONAL-OF • UNDATE • YEAR-TO-YYYY • YEARWINDOW
LENGTH function	Specifying a pointer, the ADDRESS OF special register, or the LENGTH OF special register as an argument to the function
Compiler-directing statements	<p>The following statements:</p> <ul style="list-style-type: none"> • BASIS • CBL(PROCESS) • *CONTROL and *CBL • DELETE • EJECT • INSERT • READY or RESET TRACE • SERVICE LABEL • SERVICE RELOAD • SKIP1, SKIP2, and SKIP3 • TITLE
COPY statement	<p>The optionality of the syntax "OF <i>library-name</i>" for specifying a text-name qualifier</p> <p>Literals for specifying <i>text-name</i> and <i>library-name</i></p> <p>SUPPRESS phrase</p> <p>Nested COPY statements</p> <p>Hyphen as the first or last character in the <i>word</i> form of REPLACING operands</p> <p>The use of any character (other than a COBOL separator) in the <i>word</i> form of REPLACING operands. [Standard COBOL 85 accepts only the characters used in formation of user-defined words.]</p>
Compiler directives	>>CALLINTERFACE directive

Appendix B. Compiler limits

The following table lists the compiler limits for programs and class definitions.

Table 61. Compiler limits

Language element	Compiler limit
Maximum length of user-defined words (for example, <i>data-name</i> , <i>file-name</i> , <i>class-name</i>)	30 bytes
Size of program	999,999 lines
Number of literals	4,194,303
Total length of literals	4,194,303 bytes
Reserved word table entries	1536
COPY REPLACING . . . BY . . . (items per COPY statement)	No limit
Number of COPY libraries	No limit
Identification division	
Environment division	
Configuration section	
Special-names paragraph	
<i>mnemonic-name</i> IS	18
UPSI- <i>n</i> . . . (switches)	0-7
<i>alphabet-name</i> IS . . .	No limit
Literal THRU . . . or ALSO . . .	256
Input-Output section	
File-control paragraph	
SELECT <i>file-name</i> . . .	A maximum of 65,535 file names can be assigned external names
ASSIGN <i>system-name</i> . . .	No limit
RECORD KEY length	No limit
RESERVE <i>integer</i> (buffers)	255
I-O-control paragraph	
RERUN ON <i>system-name</i> . . .	32,767
RERUN <i>integer</i> RECORDS	16,777,215
SAME RECORD AREA	255
SAME RECORD AREA FOR <i>file-name</i> . . .	255
SAME SORT/MERGE AREA	No limit
MULTIPLE FILE <i>file-name</i> . . .	No limit
Data division	
77 data item size	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes

Table 61. Compiler limits (continued)

Language element	Compiler limit
01-49 data item size	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes
Total 01 + 77 (data items)	No limit
88 condition-names . . .	No limit
VALUE literal . . .	No limit
66 RENAMES . . .	No limit
PICTURE clause, number of characters in <i>character-string</i>	50
PICTURE clause, numeric item digit positions	With ARITH(COMPAT): 18 With ARITH(EXTEND): 31
PICTURE clause, numeric-edited character positions	249
Picture symbol replication ()	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes
Picture symbol replication (editing)	32,767
Picture symbol replication (), class DBCS items	With ADDR(32): 499,999,999 bytes With ADDR(64): 1,073,741,823 bytes
Picture symbol replication (), class national items	With ADDR(32): 499,999,999 bytes With ADDR(64): 1,073,741,823 bytes
Elementary item size	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes
OCCURS integer	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes
Total number of ODOs	4,194,303
Table size	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes
Table element size	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes
ASCENDING or DESCENDING KEY . . . (per OCCURS clause)	12 KEYS
Total length of keys (per OCCURS clause)	256 bytes
INDEXED BY . . . (index names per OCCURS clause)	12
Total number of indexes (index names) per class or program	65,535
Size of relative index	32,765
File section	
FD record description entry	1,048,576 bytes For more information on size limits, see “File input-output limitations” on page 581.
FD <i>file-name</i> . . .	65,535

Table 61. Compiler limits (continued)

Language element	Compiler limit
LABEL <i>data-name</i> . . . (if no optional clauses)	255
Label record length	80 bytes
DATA RECORD <i>data-name</i> . . .	No limit
BLOCK CONTAINS <i>integer</i>	2,147,483,647 bytes
RECORD CONTAINS <i>integer</i>	1,048,576 bytes For more information on size limits, see “File input-output limitations” on page 581.
LINAGE clause values	99,999,999
SD <i>file-name</i> . . .	65,535
DATA RECORD <i>data-name</i> . . .	No limit
STL file system, maximum records	2,147,483,647 bytes For more information on size limits, see “File input-output limitations” on page 581.
SdU file system, maximum records	2,147,483,647 bytes For more information on size limits, see “File input-output limitations” on page 581.
Linkage section	
Total size	With ADDR(32): 2,147,483,646 bytes With ADDR(64): Unlimited, up to the available 64-bit addressing capacity of the machine.
Local-storage section	
Total size	With ADDR(32): 2,147,483,646 bytes With ADDR(64): Unlimited, up to the available 64-bit addressing capacity of the machine.
Working-storage section	
Total size of items without the external attribute	With ADDR(32): 2,147,483,646 bytes With ADDR(64): Unlimited, up to the available 64-bit addressing capacity of the machine.
Total size of items with the external attribute	With ADDR(32): 2,147,483,646 bytes With ADDR(64): Unlimited, up to the available 64-bit addressing capacity of the machine.
Procedure division	
Procedure division USING <i>identifier</i> . . .	32,767
Procedure-names	1,048,575
Subscripted data-names per statement	32,767
Verbs per line (TEST)	7
ADD <i>identifier</i> . . .	No limit
ALTER <i>procedure-name-1</i> TO <i>procedure-name-2</i> . . .	4,194,303

Table 61. Compiler limits (continued)

Language element	Compiler limit
CALL . . . BY CONTENT <i>identifier</i>	With ADDR(32): 2,147,483,647 bytes With ADDR(64): Unlimited, up to the available 64-bit addressing capacity of the machine.
CALL <i>identifier</i> or <i>literal</i> USING <i>identifier</i> or <i>literal</i> . . .	500
CALL <i>literal</i> . . .	4,194,303
Active programs in a run unit	32,767
Number of names called (DYN option)	No limit
CANCEL <i>identifier</i> or <i>literal</i> . . .	No limit
CLOSE <i>file-name</i> . . .	No limit
COMPUTE <i>identifier</i> . . .	No limit
DISPLAY <i>identifier</i> or <i>literal</i> . . .	No limit
DIVIDE <i>identifier</i> . . .	No limit
ENTRY USING <i>identifier</i> or <i>literal</i> . . .	No limit
EVALUATE . . . subjects	64
EVALUATE . . . WHEN clauses	256
GO <i>procedure-name</i> . . . DEPENDING	255
INSPECT TALLYING and REPLACING clauses	No limit
MERGE <i>file-name</i> ASC or DES KEY . . .	No limit
Total merge key length	4,092 bytes
MERGE USING <i>file-name</i> . . .	16
MOVE <i>identifier</i> or <i>literal</i> TO <i>identifier</i> . . .	No limit
MULTIPLY <i>identifier</i> . . .	No limit
OPEN <i>file-name</i> . . .	No limit
PERFORM	4,194,303
PERFORM . . . TIMES <i>identifier</i> or <i>literal</i>	999,999,999
SEARCH . . . WHEN . . .	No limit
SET <i>index</i> or <i>identifier</i> . . . TO	No limit
SET <i>index</i> . . . UP/DOWN	No limit
SORT <i>file-name</i> ASC or DES KEY	No limit
Total sort key length	4,092 bytes
SORT USING <i>file-name</i> . . .	16
STRING <i>identifier</i> . . .	No limit
STRING DELIMITED <i>identifier</i> or <i>literal</i> . . .	No limit
UNSTRING DELIMITED <i>identifier</i> or <i>literal</i> . . .	255
UNSTRING INTO <i>identifier</i> or <i>literal</i> . . .	No limit
USE . . . ON <i>file-name</i> . . .	No limit
XML PARSE statement, maximum size of <i>identifier</i>	With ADDR(32): 999,999,999 bytes With ADDR(64): 2,147,483,646 bytes

File input-output limitations

Line-sequential files

- Maximum record size: 1,048,576 bytes
- Maximum number of bytes allocated for a file: no maximum

SdU files

- Minimum record size: 1 byte
- Maximum record size: 64,000 bytes
- Maximum record key length: 255 bytes
- Maximum relative key value: $2^{31} - 1$
- Maximum number of bytes allocated for a file:
 - If FILEMODE(SMALL) is in effect: 2 GB
 - If FILEMODE(LARGE) is in effect: maximum record size times maximum records

STL files

- Minimum record size: 1 byte
- Maximum record size: 1,048,576 bytes
- Maximum record key length: 255 bytes
- Maximum number of alternate keys: 253
- Maximum relative key value: $2^{31} - 1$
- Maximum number of bytes allocated for a file:
 - If FILEMODE(SMALL) is in effect: 2 GB
 - If FILEMODE(LARGE) is in effect: maximum record size times maximum records

RSD files

- Fixed-length records only
- Minimum record size: 1 byte
- Maximum record size: 1,048,576 bytes
- Maximum number of bytes allocated for a file: no maximum

DB2 files

The limitations listed below assume simple tables and the default 4 KB database page size.

- Minimum record size: 0 bytes
- Maximum row length: 4005 bytes
- Maximum record size for fixed-length records:
 - Indexed records: 4005 bytes
 - Small-format relative or sequential records: 4001 bytes
 - Large-format relative or sequential records: 3997 bytes

Because keys must be at a fixed location in an indexed file record, there are separate constraints for indexed files that end with a variable-length data segment.

- Maximum record size for variable-length records:

- For VARCHAR columns, the limits are 4 bytes fewer than for fixed-length records, because the data for the VARCHAR type is maintained directly in the row, and is preceded by a 4-byte descriptor.
- For other variable length types, such as BLOB, only the descriptor is stored in the row. The data is stored separately, and has a maximum size of 2 GB.
- Maximum number of records for relative and sequential files:
 - For small filemode: 2**32
 - For large filemode: 2**64
- Maximum number of records or alternate keys: 500
- Maximum record key value:
 - For small filemode: 2**32 -1
 - For large filemode: 2**64 -1
- Maximum number of bytes allocated for a file: 64 GB

For information about the limits of DB2 files, see *SQL and XML limits* in the *DB2 for Linux, UNIX, and Windows, Database Administration Concepts and Configuration Reference*.

For information about the limits of Encina SFS files, see the *TXSeries for Multiplatforms: CICS Application Programming Guide*.

For information about FILEMODE, see *FILEMODE* in the *COBOL for AIX Programming Guide*.

For information about ARITH, see *ARITH* in the *COBOL for AIX Programming Guide*.

For information about ADDR, see *ADDR* in the *COBOL for AIX Programming Guide*.

Appendix C. Source conversion utility (scu)

The source conversion utility, `scu`, is a stand-alone AIX program that assists in the conversion of COBOL source programs from non-IBM or free-format source formats to a format that can be compiled by COBOL for AIX.

`scu` does the following transformations:

- Converts white-space characters to true spaces, including tab expansion, with optional controls
- Normalizes line-end characters
- Repositions items to start in the proper areas, such as the indicator area, Area A, or Area B, as required
- Ensures special alignment for CBL(PROCESS) statements
- Optionally blanks sequence numbers in columns 1 through 6, and removes serial numbers in columns 73 through 80
- Converts anomalous fixed-source-format input by default, and optionally converts free-format input

For detailed information about the available functions of `scu`, see the `scu man` page. The following command provides a summary of functions and syntax of the `scu` command:

```
scu -h
```

For further details about which items belong in Area A and Area B, see Chapter 6, “Reference format,” on page 45.

Appendix D. EBCDIC and ASCII collating sequences

The ascending collating sequences for both the single-byte EBCDIC (Extended Binary Coded Decimal Interchange Code) and single-byte ASCII (American National Standard Code for Information Interchange) character sets are shown in this appendix. The collating sequence is defined by the ordinal number of characters in the character set, relative to 1.

The symbols and associated meanings shown for the EBCDIC collating sequence are those defined in the EBCDIC code page defined with CCSID 1140. Symbols and meanings can vary for other EBCDIC code pages, but the collating sequence is unchanged.

EBCDIC collating sequence

The following table presents the collating sequence for single-byte EBCDIC code page 1140. Ellipsis (. . .) indicates omission of a range of ordinal numbers between predecessor and successor ordinal numbers.

Table 62. EBCDIC collating sequence

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
. . .				
65		Space	64	40
. . .				
75	¢	Cent sign	74	4A
76	.	Period, decimal point	75	4B
77	<	Less than sign	76	4C
78	(Left parenthesis	77	4D
79	+	Plus sign	78	4E
80		Vertical bar, logical OR	79	4F
81	&	Ampersand	80	50
. . .				
91	!	Exclamation point	90	5A
92	\$	Dollar sign	91	5B
93	*	Asterisk	92	5C
94)	Right parenthesis	93	5D
95	;	Semicolon	94	5E
96	¬	Logical NOT	95	5F
97	-	Minus, hyphen	96	60
98	/	Slash	97	61
. . .				
108	,	Comma	107	6B
109	%	Percent sign	108	6C
110	_	Underscore	109	6D

Table 62. EBCDIC collating sequence (continued)

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
111	>	Greater than sign	110	6E
112	?	Question mark	111	6F
...				
122	`	Grave accent	121	79
123	:	Colon	122	7A
124	#	Number sign, pound sign	123	7B
125	@	At sign	124	7C
126	'	Apostrophe, prime sign	125	7D
127	=	Equal sign	126	7E
128	"	Quotation marks	127	7F
...				
130	a		129	81
131	b		130	82
132	c		131	83
133	d		132	84
134	e		133	85
135	f		134	86
136	g		135	87
137	h		136	88
138	i		137	89
...				
146	j		145	91
147	k		146	92
148	l		147	93
149	m		148	94
150	n		149	95
151	o		150	96
152	p		151	97
153	q		152	98
154	r		153	99
...				
160	€	Euro currency sign	159	9F
...				
162	~	Tilde	161	A1
163	s		162	A2
164	t		163	A3
165	u		164	A4
166	v		165	A5
167	w		166	A6

Table 62. EBCDIC collating sequence (continued)

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
168	x		167	A7
169	y		168	A8
170	z		169	A9
...				
177	^	Caret	176	B0
...				
188	[Opening square bracket	187	BA
189]	Closing square bracket	188	BB
...				
193	{	Opening brace	192	C0
194	A		193	C1
195	B		194	C2
196	C		195	C3
197	D		196	C4
198	E		197	C5
199	F		198	C6
200	G		199	C7
201	H		200	C8
202	I		201	C9
...				
209	}	Closing brace	208	D0
210	J		209	D1
211	K		210	D2
212	L		211	D3
213	M		212	D4
214	N		213	D5
215	O		214	D6
216	P		215	D7
217	Q		216	D8
218	R		217	D9
...				
225	\	Backslash	224	E0
...				
227	S		226	E2
228	T		227	E3
229	U		228	E4
230	V		229	E5
231	W		230	E6
232	X		231	E7

Table 62. EBCDIC collating sequence (continued)

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
233	Y		232	E8
234	Z		233	E9
...				
241	0		240	F0
242	1		241	F1
243	2		242	F2
244	3		243	F3
245	4		244	F4
246	5		245	F5
247	6		246	F6
248	7		247	F7
249	8		248	F8
250	9		249	F9
...				

US English ASCII code page

The following table presents the collating sequence for the US English ASCII code page. The collating sequence is the order in which characters are defined in ANSI INCITS 4, the 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII), and in the International Reference Version of *ISO/IEC 646, 7-Bit Coded Character Set for Information Interchange*.

Ellipsis (. . .) indicates omission of a range of ordinal numbers between predecessor and successor ordinal numbers.

Table 63. ASCII collating sequence

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
1		Null	0	0
...				
33		Space	32	20
34	!	Exclamation point	33	21
35	"	Quotation mark	34	22
36	#	Number sign	35	23
37	\$	Dollar sign	36	24
38	%	Percent sign	37	25
39	&	Ampersand	38	26
40	'	Apostrophe, prime sign	39	27
41	(Opening parenthesis	40	28
42)	Closing parenthesis	41	29
43	*	Asterisk	42	2A

Table 63. ASCII collating sequence (continued)

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
44	+	Plus sign	43	2B
45	,	Comma	44	2C
46	-	Hyphen, minus	45	2D
47	.	Period, decimal point	46	2E
48	/	Slash, solidus	47	2F
49	0		48	30
50	1		49	31
51	2		50	32
52	3		51	33
53	4		52	34
54	5		53	35
55	6		54	36
56	7		55	37
57	8		56	38
58	9		57	39
59	:	Colon	58	3A
60	;	Semicolon	59	3B
61	<	Less than sign	60	3C
62	=	Equal sign	61	3D
63	>	Greater than sign	62	3E
64	?	Question mark	63	3F
65	@	Commercial At sign	64	40
66	A		65	41
67	B		66	42
68	C		67	43
69	D		68	44
70	E		69	45
71	F		70	46
72	G		71	47
73	H		72	48
74	I		73	49
75	J		74	4A
76	K		75	4B
77	L		76	4C
78	M		77	4D
79	N		78	4E
80	O		79	4F
81	P		80	50
82	Q		81	51

Table 63. ASCII collating sequence (continued)

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
83	R		82	52
84	S		83	53
85	T		84	54
86	U		85	55
87	V		86	56
88	W		87	57
89	X		88	58
90	Y		89	59
91	Z		90	5A
92	[Opening bracket	91	5B
93	\	Backslash, reverse solidus	92	5C
94]	Closing bracket	93	5D
95	^	Caret	94	5E
96	_	Underscore	95	5F
97	`	Grave accent	96	60
98	a		97	61
99	b		98	62
100	c		99	63
101	d		100	64
102	e		101	65
103	f		102	66
104	g		103	67
105	h		104	68
106	i		105	69
107	j		106	6A
108	k		107	6B
109	l		108	6C
110	m		109	6D
111	n		110	6E
112	o		111	6F
113	p		112	70
114	q		113	71
115	r		114	72
116	s		115	73
117	t		116	74
118	u		117	75
119	v		118	76
120	w		119	77
121	x		120	78

Table 63. ASCII collating sequence *(continued)*

Ordinal number	Symbol	Meaning	Decimal representation	Hex representation
122	y		121	79
123	z		122	7A
124	{	Opening brace	123	7B
125		Vertical bar	124	7C
126	}	Closing brace	125	7D
127	~	Tilde	126	7E

Appendix E. Source language debugging

COBOL language elements that implement the debugging feature are:

- Debugging lines
- Debugging sections
- DEBUG-ITEM special register
- Compile-time switch (WITH DEBUGGING MODE clause)
- Object-time switch

Debugging lines

A *debugging line* is a statement that is compiled only when the compile-time switch is activated. Debugging lines allow you, for example, to check the value of a data item at certain points in a procedure.

To specify a debugging line in your program, code a D in column 7 (the indicator area). You can include successive debugging lines, but each must have a D in column 7. You cannot break character-strings across two lines.

All your debugging lines must be written so that the program is syntactically correct, whether the debugging lines are compiled or treated as comments.

You can code debugging lines anywhere in your program after the OBJECT-COMPUTER paragraph.

A debugging line that contains only spaces in Area A and in Area B is treated as a blank line.

Debugging sections

Debugging sections are permitted only in the outermost program; they are not valid in nested programs. Debugging sections are never triggered by procedures contained in nested programs.

Debugging sections are declarative procedures. Declarative procedures are described under “USE statement” on page 556. A debugging section can be called, for example, by a PERFORM statement that causes repeated execution of a procedure. Any associated *procedure-name* debugging declarative section is executed once for each repetition.

A debugging section executes *only* if both the compile-time switch and the object-time switch are activated.

The debug feature recognizes each separate occurrence of an imperative statement *within* an imperative statement as the beginning of a separate statement.

You cannot refer to a procedure defined within a debugging section from a statement outside of the debugging section.

References to the DEBUG-ITEM special register can be made only from within a debugging declarative procedure.

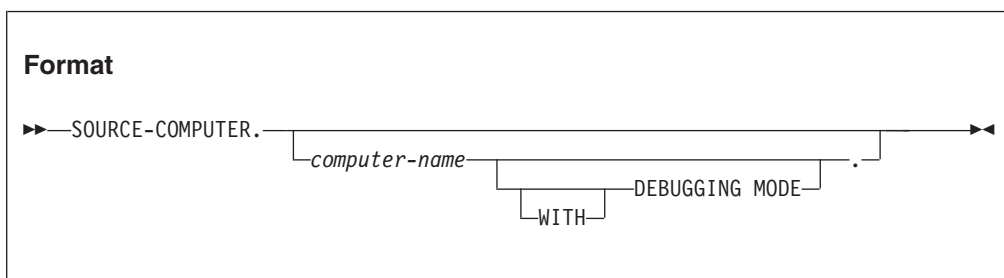
DEBUG-ITEM special register

The DEBUG-ITEM special register provides information for a debugging declarative procedure about the conditions that cause debugging section execution.

For details of the DEBUG-ITEM special register, see “DEBUG-ITEM” on page 17.

Activate compile-time switch

The compile-time switch activates the debugging lines and sections. To place the compile-time switch in effect, specify WITH DEBUGGING MODE in the SOURCE COMPUTER paragraph of the configuration section.



WITH DEBUGGING MODE

When WITH DEBUGGING MODE is specified, all debugging sections and debugging lines are compiled.

When WITH DEBUGGING MODE is omitted, all debugging sections and debugging lines are treated as comments.

Usage note: If you include a COPY statement as a debugging line, the letter "D" must appear on the first line of the COPY statement. The compiler treats the copied text as the debugging line or lines. The COPY statement is executed, regardless of whether WITH DEBUGGING MODE is specified or not.

Activate object-time switch

The object-time switch is set when the runtime option DEBUG or NODEBUG is specified. (NODEBUG is the default supplied by IBM.)

For details on the format, see *DEBUG* in the *COBOL for AIX Programming Guide*.

The USE FOR DEBUGGING declarative procedures are activated when DEBUG is in effect and inhibited when NODEBUG is in effect.

The debugging lines (lines with "D" or "d" in column 7) are not affected by the DEBUG or NODEBUG option; they are always active if they have been compiled.

When WITH DEBUGGING MODE is *not* specified in the SOURCE-COMPUTER paragraph, the object-time switch has no effect on execution of the object program.

You do not have to recompile the source unit to activate or deactivate the object-time switch.

Appendix F. Reserved words

The following table identifies words that are reserved in COBOL for AIX and words that you should avoid because they might be reserved in a future release of COBOL for AIX.

- Words marked *X* under *Reserved* are reserved for function implemented in COBOL for AIX. If used as user-defined names, these words are flagged with an S-level message.
- Words marked *X* under *Standard only* are Standard COBOL 85 reserved words for function not implemented in COBOL for AIX. Use of these words as user-defined names is flagged with an S-level message.
- Words marked *X* under *Potential reserved words* are words that might be reserved in a future release of COBOL for AIX. IBM recommends that you not use these words as user-defined names. Use of these words as user-defined names is flagged with an I-level message.

This column includes words reserved in Standard COBOL 2002.

Table 64. **Reserved words**

Word	Reserved	Standard only	Potential reserved words
ACCEPT	X		
ACCESS	X		
ACTIVE-CLASS			X
ADD	X		
ADDRESS	X		
ADVANCING	X		
AFTER	X		
ALIGNED			X
ALL	X		
ALLOCATE			X
ALPHABET	X		
ALPHABETIC	X		
ALPHABETIC-LOWER	X		
ALPHABETIC-UPPER	X		
ALPHANUMERIC	X		
ALPHANUMERIC-EDITED	X		
ALSO	X		
ALTER	X		
ALTERNATE	X		
AND	X		
ANY	X		
ANYCASE			X
APPLY	X		
ARE	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
AREA	X		
AREAS	X		
ASCENDING	X		
ASSIGN	X		
AT	X		
AUTHOR	X		
B-AND			X
B-NOT			X
B-OR			X
B-XOR			X
BASED			X
BASIS	X		
BEFORE	X		
BEGINNING	X		
BINARY	X		
BINARY-CHAR			X
BINARY-DOUBLE			X
BINARY-LONG			X
BINARY-SHORT			X
BIT			X
BLANK	X		
BLOCK	X		
BOOLEAN			X
BOTTOM	X		
BY	X		
CALL	X		
CANCEL	X		
CBL	X		
CD		X	
CF		X	
CH		X	
CHARACTER	X		
CHARACTERS	X		
CLASS	X		
CLASS-ID	X		
CLOCK-UNITS		X	
CLOSE	X		
COBOL	X		
CODE	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
CODE-SET	X		
COL			X
COLLATING	X		
COLS			X
COLUMN		X	
COLUMNS			X
COM-REG	X		
COMMA	X		
COMMON	X		
COMMUNICATION		X	
COMP	X		
COMP-1	X		
COMP-2	X		
COMP-3	X		
COMP-4	X		
COMP-5	X		
COMPUTATIONAL	X		
COMPUTATIONAL-1	X		
COMPUTATIONAL-2	X		
COMPUTATIONAL-3	X		
COMPUTATIONAL-4	X		
COMPUTATIONAL-5	X		
COMPUTE	X		
CONDITION			X
CONFIGURATION	X		
CONSTANT			X
CONTAINS	X		
CONTENT	X		
CONTINUE	X		
CONTROL		X	
CONTROLS		X	
CONVERTING	X		
COPY	X		
CORR	X		
CORRESPONDING	X		
COUNT	X		
CRT			X
CURRENCY	X		
CURSOR			X

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
DATA	X		
DATA-POINTER			X
DATE	X		
DATE-COMPILED	X		
DATE-WRITTEN	X		
DAY	X		
DAY-OF-WEEK	X		
DBCS	X		
DE		X	
DEBUG-CONTENTS	X		
DEBUG-ITEM	X		
DEBUG-LINE	X		
DEBUG-NAME	X		
DEBUG-SUB-1	X		
DEBUG-SUB-2	X		
DEBUG-SUB-3	X		
DEBUGGING	X		
DECIMAL-POINT	X		
DECLARATIVES	X		
DEFAULT			X
DELETE	X		
DELIMITED	X		
DELIMITER	X		
DEPENDING	X		
DESCENDING	X		
DESTINATION		X	
DETAIL		X	
DISABLE		X	
DISPLAY	X		
DISPLAY-1	X		
DIVIDE	X		
DIVISION	X		
DOWN	X		
DUPLICATES	X		
DYNAMIC	X		
EC			X
EGCS	X		
EGI		X	
EJECT	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
ELSE	X		
EMI		X	
ENABLE		X	
END	X		
END-ACCEPT			X
END-ADD	X		
END-CALL	X		
END-COMPUTE	X		
END-DELETE	X		
END-DISPLAY			X
END-DIVIDE	X		
END-EVALUATE	X		
END-EXEC	X		
END-IF	X		
END-INVOKE	X		
END-MULTIPLY	X		
END-OF-PAGE	X		
END-PERFORM	X		
END-READ	X		
END-RECEIVE		X	
END-RETURN	X		
END-REWRITE	X		
END-SEARCH	X		
END-START	X		
END-STRING	X		
END-SUBTRACT	X		
END-UNSTRING	X		
END-WRITE	X		
END-XML	X		
ENDING	X		
ENTER	X		
ENTRY	X		
ENVIRONMENT	X		
EO			X
EOP	X		
EQUAL	X		
ERROR	X		
ESI		X	
EVALUATE	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
EVERY	X		
EXCEPTION	X		
EXCEPTION-OBJECT			X
EXEC	X		
EXECUTE	X		
EXIT	X		
EXTEND	X		
EXTERNAL	X		
FACTORY	X		
FALSE	X		
FD	X		
FILE	X		
FILE-CONTROL	X		
FILLER	X		
FINAL		X	
FIRST	X		
FLOAT-EXTENDED			X
FLOAT-LONG			X
FLOAT-SHORT			X
FOOTING	X		
FOR	X		
FORMAT			X
FREE			X
FROM	X		
FUNCTION	X		
FUNCTION-ID			X
FUNCTION-POINTER	X		
GENERATE	X		
GET			X
GIVING	X		
GLOBAL	X		
GO	X		
GOBACK	X		
GREATER	X		
GROUP		X	
GROUP-USAGE	X		
HEADING		X	
HIGH-VALUE	X		
HIGH-VALUES	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
I-O	X		
I-O-CONTROL	X		
ID	X		
IDENTIFICATION	X		
IF	X		
IN	X		
INDEX	X		
INDEXED	X		
INDICATE		X	
INHERITS	X		
INITIAL	X		
INITIALIZE	X		
INITIATE		X	
INPUT	X		
INPUT-OUTPUT	X		
INSERT	X		
INSPECT	X		
INSTALLATION	X		
INTERFACE			X
INTERFACE-ID			X
INTO	X		
INVALID	X		
INVOKE	X		
IS	X		
JNIENVPTR	X		
JUST	X		
JUSTIFIED	X		
KANJI	X		
KEY	X		
LABEL	X		
LAST		X	
LEADING	X		
LEFT	X		
LENGTH	X		
LESS	X		
LIMIT		X	
LIMITS		X	
LINAGE	X		
LINAGE-COUNTER	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
LINE	X		
LINE-COUNTER		X	
LINES	X		
LINKAGE	X		
LOCAL-STORAGE	X		
LOCALE			X
LOCK	X		
LOW-VALUE	X		
LOW-VALUES	X		
MEMORY	X		
MERGE	X		
MESSAGE		X	
METHOD	X		
METHOD-ID	X		
MINUS			X
MODE	X		
MODULES	X		
MORE-LABELS	X		
MOVE	X		
MULTIPLE	X		
MULTIPLY	X		
NATIONAL	X		
NATIONAL-EDITED	X		
NATIVE	X		
NEGATIVE	X		
NESTED			X
NEXT	X		
NO	X		
NOT	X		
NULL	X		
NULLS	X		
NUMBER		X	
NUMERIC	X		
NUMERIC-EDITED	X		
OBJECT	X		
OBJECT-COMPUTER	X		
OBJECT-REFERENCE			X
OCCURS	X		
OF	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
OFF	X		
OMITTED	X		
ON	X		
OPEN	X		
OPTIONAL	X		
OPTIONS			X
OR	X		
ORDER	X		
ORGANIZATION	X		
OTHER	X		
OUTPUT	X		
OVERFLOW	X		
OVERRIDE	X		
PACKED-DECIMAL	X		
PADDING	X		
PAGE	X		
PAGE-COUNTER		X	
PASSWORD	X		
PERFORM	X		
PF		X	
PH		X	
PIC	X		
PICTURE	X		
PLUS		X	
POINTER	X		
POSITION	X		
POSITIVE	X		
PRESENT			X
PRINTING		X	
PROCEDURE	X		
PROCEDURE-POINTER	X		
PROCEDURES	X		
PROCEED	X		
PROCESSING	X		
PROGRAM	X		
PROGRAM-ID	X		
PROGRAM-POINTER			X
PROPERTY			X
PROTOTYPE			X

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
PURGE		X	
QUEUE		X	
QUOTE	X		
QUOTES	X		
RAISE			X
RAISING			X
RANDOM	X		
RD		X	
READ	X		
READY	X		
RECEIVE		X	
RECORD	X		
RECORDING	X		
RECORDS	X		
RECURSIVE	X		
REDEFINES	X		
REEL	X		
REFERENCE	X		
REFERENCES	X		
RELATIVE	X		
RELEASE	X		
RELOAD	X		
REMAINDER	X		
REMOVAL	X		
RENAMES	X		
REPLACE	X		
REPLACING	X		
REPORT		X	
REPORTING		X	
REPORTS		X	
REPOSITORY	X		
RERUN	X		
RESERVE	X		
RESET	X		
RESUME			X
RETRY			X
RETURN	X		
RETURN-CODE	X		
RETURNING	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
REVERSED	X		
REWIND	X		
REWRITE	X		
RF		X	
RH		X	
RIGHT	X		
ROUNDED	X		
RUN	X		
SAME	X		
SCREEN			X
SD	X		
SEARCH	X		
SECTION	X		
SECURITY	X		
SEGMENT		X	
SEGMENT-LIMIT	X		
SELECT	X		
SELF	X		
SEND		X	
SENTENCE	X		
SEPARATE	X		
SEQUENCE	X		
SEQUENTIAL	X		
SERVICE	X		
SET	X		
SHARING			X
SHIFT-IN	X		
SHIFT-OUT	X		
SIGN	X		
SIZE	X		
SKIP1	X		
SKIP2	X		
SKIP3	X		
SORT	X		
SORT-CONTROL	X		
SORT-CORE-SIZE	X		
SORT-FILE-SIZE	X		
SORT-MERGE	X		
SORT-MESSAGE	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
SORT-MODE-SIZE	X		
SORT-RETURN	X		
SOURCE		X	
SOURCE-COMPUTER	X		
SOURCES			X
SPACE	X		
SPACES	X		
SPECIAL-NAMES	X		
SQL	X		
STANDARD	X		
STANDARD-1	X		
STANDARD-2	X		
START	X		
STATUS	X		
STOP	X		
STRING	X		
SUB-QUEUE-1		X	
SUB-QUEUE-2		X	
SUB-QUEUE-3		X	
SUBTRACT	X		
SUM		X	
SUPER	X		
SUPPRESS	X		
SYMBOLIC	X		
SYNC	X		
SYNCHRONIZED	X		
SYSTEM-DEFAULT			X
TABLE		X	
TALLY	X		
TALLYING	X		
TAPE	X		
TERMINAL		X	
TERMINATE		X	
TEST	X		
TEXT		X	
THAN	X		
THEN	X		
THROUGH	X		
THRU	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
TIME	X		
TIMES	X		
TITLE	X		
TO	X		
TOP	X		
TRACE	X		
TRAILING	X		
TRUE	X		
TYPE	X		
TYPEDDEF			X
UNIT	X		
UNIVERSAL			X
UNLOCK			X
UNSTRING	X		
UNTIL	X		
UP	X		
UPON	X		
USAGE	X		
USE	X		
USER-DEFAULT			X
USING	X		
VAL-STATUS			X
VALID			X
VALIDATE			X
VALIDATE-STATUS			X
VALUE	X		
VALUES	X		
VARYING	X		
WHEN	X		
WHEN-COMPILED	X		
WITH	X		
WORDS	X		
WORKING-STORAGE	X		
WRITE	X		
WRITE-ONLY	X		
XML	X		
XML-CODE	X		
XML-EVENT	X		
XML-NAMESPACE	X		

Table 64. **Reserved words** (continued)

Word	Reserved	Standard only	Potential reserved words
XML-NAMESPACE-PREFIX	X		
XML-NNAMESPACE	X		
XML-NNAMESPACE-PREFIX	X		
XML-NTEXT	X		
XML-TEXT	X		
ZERO	X		
ZEROES	X		
ZEROS	X		

Appendix G. Locale considerations

A *locale* is a collection of language-specific and culture-specific conventions for information processing. A locale makes information about language and culture available at run time so that the same program or method can display or process data differently for different countries or cultures.

For additional information about locales and the details of setting locales for specific languages and cultures, see *Setting the locale* in the *COBOL for AIX Programming Guide*.

Compile-time versus runtime locale

In general, the COBOL run time determines the locale in effect when the COBOL application is activated. However, the following processing is based on the locale in effect at compile time:

- Evaluation of user-defined names and literal values

The compiler uses the code page indicated by the locale in effect at compile time to evaluate the source program. This includes the evaluation of user-defined names and literal values.

When a literal value is associated with a data item such that code page conversion is required, the code page in effect at run time is used for the data item. For an item of class alphanumeric with native ASCII encoding, the code page indicated by the locale in effect at run time is used. For an item of class alphanumeric with EBCDIC encoding, the EBCDIC code page in effect at run time is used.

- Evaluation of collating sequences

When the COLLSEQ(LOCALE) compiler option or the NCOLLSEQ(LOCALE) compiler option is in effect, the compile-time locale determines the following values:

- The range of characters specified by literals in a THRU phrase for the following language elements:
 - A condition-name VALUE clause
 - An EVALUATE statement
 - An ALPHABET clause in the SPECIAL-NAMES paragraph
 - A CLASS clause in the SPECIAL-NAMES paragraph
- The ordinal positions of characters specified in a SYMBOLIC CHARACTERS clause

The following sections describe the effects of locales on COBOL runtime processing.

Code pages

The runtime locale is used to determine:

- The code page for the encoding of alphanumeric and DBCS data items that have native (non-EBCDIC) encoding. The EBCDIC code page in effect is used when the CHAR(EBCDIC) compiler option is specified, the EBCDIC_CODEPAGE environment variable is not set, and the data item is described without the NATIVE phrase.
- Case mappings for the UPPER-CASE and LOWER-CASE intrinsic functions
- The output code page for the DISPLAY-OF intrinsic function when a codepage argument is omitted. The EBCDIC code page in effect is used when the CHAR(EBCDIC) compiler option is specified, the EBCDIC_CODEPAGE environment variable is not set, and the data item is described without the NATIVE phrase.
- The source code page for the NATIONAL-OF intrinsic function when a codepage argument is omitted. The EBCDIC code page in effect is used when the CHAR(EBCDIC) compiler option is specified, the EBCDIC_CODEPAGE environment variable is not set, and the data item is described without the NATIVE phrase.
- The code page for conversion of data for accepting into data items of usage NATIONAL with the ACCEPT statement
- The code page for conversion of data for displaying from data items of usage NATIONAL with the DISPLAY statement

Two runtime code pages can be in effect: one for native data (non-EBCDIC data) and one for host data (EBCDIC data). For information about how these runtime code pages are determined, see *Specifying the code page for character data* in the *COBOL for AIX Programming Guide*.

Collating sequences

In general, COBOL for AIX uses the collating sequence defined by the runtime locale when the COLLSEQ(LOCALE) compiler option or the NCOLLSEQ(LOCALE) compiler option is in effect. COLLSEQ(LOCALE) affects alphanumeric and DBCS data items; NCOLLSEQ(LOCALE) affects items described with USAGE NATIONAL.

When COLLSEQ(BINARY) or NCOLLSEQ(BINARY) is specified, a binary collating sequence is used.

In some cases, the language rules specify the use of a non-locale collating sequence regardless of the setting of the COLLSEQ or NCOLLSEQ compiler option. For example:

- A binary collating sequence is always used for indexed file keys.
- A collating sequence specified in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph is used for alphanumeric comparisons.
- A collating sequence specified in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph is used for SORT or MERGE statements with alphanumeric keys, unless the COLLATING SEQUENCE phrase is specified in the SORT or MERGE statement.
- A collating sequence specified in the COLLATING SEQUENCE phrase of a SORT or MERGE statement is used for alphabetic keys and alphanumeric keys.

Supported locales

COBOL for AIX supports locales for certain combinations of language, country, and code page. System environment variables identify the runtime locale in effect for specific locale categories. For details, see *Using environment variables to specify a locale* in the *COBOL for AIX Programming Guide*.

Notices

This information was developed for products and services offered in the U.S.A.

IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information in softcopy, the photographs and color illustrations might not appear.

Programming interface information

This Language Reference documents intended Programming Interfaces that allow the customer to write programs to obtain the services of COBOL for AIX.

Trademarks

The following terms are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide:

IBM
The IBM logo
ibm.com
AIX
CICS
DB2
OS/390
System p
z/OS

A current list of IBM trademarks is available at www.ibm.com/legal/copytrade.shtml.

Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Glossary

The terms in this glossary are defined in accordance with their meaning in COBOL. These terms might or might not have the same meaning in other languages.

This glossary includes terms and definitions from the following publications:

- *ANSI INCITS 23-1985, Programming Languages - COBOL* as amended by:
 - *ANSI INCITS 23a-1989, Programming Languages - Intrinsic Function Module for COBOL*,
 - *ANSI INCITS 23b-1993, Programming Language - Correction Amendment for COBOL*
- *ANSI INCITS 172-2002 American National Standard Dictionary of Information Technology*.

American National Standard definitions are preceded by an asterisk (*).

A

*** abbreviated combined relation condition**

The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

abend Abnormal termination of program.

*** access mode**

The manner in which records are to be operated upon within a file.

*** actual decimal point**

The physical representation, using the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

*** alphabet-name**

A user-defined word, defined in the SPECIAL-NAMES paragraph of the environment division, that names a specific character set or collating sequence, or both.

*** alphabetic character**

A letter or a space character.

alphabetic data item

A data item described with a PICTURE

character-string that contains only the symbol A. An alphabetic data item has usage DISPLAY.

alphanumeric character

Any character in the computer's single-byte character set.

alphanumeric character position

See *character position*.

alphanumeric data item

A general reference to a data item described implicitly or explicitly with usage DISPLAY and category alphanumeric, alphanumeric-edited, or numeric-edited, possibly limited to specific data categories or specific data descriptions by detailed specifications.

alphanumeric-edited data item

A data item described by a PICTURE character-string that contains at least one symbol A or X and at least one of the simple insertion symbols B, 0, or /. An alphanumeric-edited data item has usage DISPLAY.

*** alphanumeric function**

A function that returns a value that is composed of a string of one or more characters from the computer's alphanumeric character set.

alphanumeric group item

A group item that is defined *without* a GROUP-USAGE NATIONAL clause. For operations such as INSPECT, STRING, and UNSTRING, an alphanumeric group item is processed as though all its content were described with usage DISPLAY, regardless of the actual content of the group. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE *identifier*, an alphanumeric group item is processed using group semantics.

alphanumeric literal

A literal that has an opening delimiter from the following set:

'
"
X'
X"
Z'
Z"

The literal content can include any character in the character set of the computer.

*** alternate record key**

A key, other than the prime record key, whose contents identify a record within an indexed file.

ANSI (American National Standards Institute)

An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

argument

(1) An identifier, a literal, an arithmetic expression, or a function-identifier that specifies a value to be used in the evaluation of a function. (2) An operand of the USING phrase of a CALL or INVOKE statement, used for passing values to a called program or an invoked method.

*** arithmetic expression**

An identifier of a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or an arithmetic expression enclosed in parentheses.

*** arithmetic operation**

The process caused by the execution of an arithmetic statement, or the evaluation of an arithmetic expression, that results in a mathematically correct solution to the arguments presented.

*** arithmetic operator**

A single character, or a fixed two-character combination that belongs to the following set:

Character	Meaning
+	Addition
-	Subtraction

Character	Meaning
*	Multiplication
/	Division
**	Exponentiation

*** arithmetic statement**

A statement that causes an arithmetic operation to be executed. The arithmetic statements are the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements.

*** ascending key**

A key, upon the values of which data is ordered starting with the lowest value of the key up to the highest value of the key, in accordance with the rules for comparing data items.

ASCII

American National Standard Code for Information Interchange. A standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange between data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

IBM has defined an extension to ASCII (characters 128-255).

ASCII DBCS

See *double-byte ASCII*.

assignment-name

A name that identifies the organization of a COBOL file and the name by which it is known to the system.

*** assumed decimal point**

A decimal point position that does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning with no physical representation.

*** AT END condition**

A condition that exists in the following circumstances:

- During the execution of a READ statement for a sequentially accessed file, when no next logical record exists in the file, or when the number of significant digits in the relative record number is larger than the size of the

relative key data item, or when an optional input file is not available.

- During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.
- During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

B

basic character set

The basic set of characters used in writing words, character-strings, and separators of the language. The basic character set is implemented in single-byte characters. The extended character set includes DBCS, UTF-8, or EUC characters, which can be used in comments, literals, and user-defined words.

Synonymous with *COBOL character set* in Standard COBOL 85.

big-endian

The default format used by the mainframe and the AIX workstation to store binary data. In this format, the least significant digit is on the highest address. See also *little-endian*.

binary item

A numeric data item represented in binary notation (on the base 2 numbering system). Binary items have a decimal equivalent consisting of the decimal digits 0 through 9, plus an operational sign. The leftmost bit of the item is the operational sign.

binary search

A dichotomizing search in which, at each step of the search, the set of data elements is divided by two; some appropriate action is taken in the case of an odd number.

* block

A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block can contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical

records that are either contained within the block or that overlap the block. The term is synonymous with *physical record*.

breakpoint

A place in an object program, usually specified by an instruction, where external intervention or a monitor program can interrupt the object program as it runs.

buffer

A portion of storage used to hold input or output data temporarily.

byte A string consisting of a certain number of bits, usually eight, treated as a unit.

byte order mark (BOM)

A Unicode character that can be used at the start of UTF-16 or UTF-32 text to indicate the byte order of subsequent text; the byte order can be either big endian or little endian.

C

CCSID

See *coded character set identifier*.

century window

A 100-year interval within which any two-digit year is unique. There are several types of century window available to COBOL programmers:

1. For windowed date fields, it is specified by the YEARWINDOW compiler option.
2. For windowing intrinsic functions DATE-TO-YYYYMMDD, DAY-TO-YYYYDDD, and YEAR-TO-YYYY, it is specified by *argument-2*.

* character

The basic indivisible unit of the language.

character encoding unit

A unit of data that corresponds to one code point in a coded character set. One or more character encoding units are used to represent a character in a coded character set. Also known as *encoding unit*.

For usage NATIONAL, a character encoding unit corresponds to one 2-byte code point of UTF-16.

For usage DISPLAY, a character encoding unit corresponds to a byte.

For usage DISPLAY-1, a character encoding unit corresponds to a 2-byte code point in the DBCS character set.

character position

The amount of physical storage or presentation space required for holding or presenting one character. The term applies to any class of character. For specific classes of characters, the following terms apply:

- *Alphanumeric character position*, for characters represented in usage DISPLAY
- *DBCS character position*, for DBCS characters represented in usage DISPLAY-1
- *National character position*, for characters represented in usage NATIONAL; synonymous with character encoding unit for UTF-16

character set

See *basic character set* and *coded character set*.

*** character-string**

A sequence of contiguous characters that forms a COBOL word, a literal, a PICTURE character-string, or a comment-entry. Must be delimited by separators.

checkpoint

A point at which information about the status of a job and the system can be recorded so that the job step can be restarted later.

class (object-oriented)

The entity that defines common behavior and implementation for zero, one, or more objects. The objects that share the same implementation are considered to be objects of the same class.

*** class condition**

The proposition (for which a truth value can be determined) that the content of an item is wholly alphabetic, is wholly numeric, is wholly DBCS, is wholly Kanji, or consists exclusively of the characters that are listed in the definition of a class-name.

class definition

The COBOL source unit that defines a class.

class-name (object-oriented)

The name of an object-oriented class definition. *Class-name* can refer to a COBOL class-name or a Java class-name.

*** class-name (of data)**

A user-defined word, defined in the SPECIAL-NAMES paragraph, that refers to the proposition for which a truth value can be defined, that the content of a data item consists exclusively of those characters listed in the definition of the class-name.

*** clause**

An ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

COBOL character set

See *basic character set*.

*** COBOL word**

See *word*.

code page

An assignment of graphic characters and control character meanings to the code points in a coded character set; for example, assignment of characters and meanings to the 256 code points in single-byte EBCDIC or ASCII. The terms *coded character set* and *code page* can be used interchangeably.

code point

A unique bit pattern defined in a code page. Graphic symbols and control characters are assigned to code points.

coded character set

A set of graphic characters and control characters along with their unambiguous assignment to specific code points (their encodings). EBCDIC is an example of a coded character set. A specific instance of encodings is called a code page. A code page specified by IBM is identified by a CCSID.

coded character set identifier (CCSID)

An IBM-defined number in the range 1 to 65,535 that identifies a specific code page.

*** collating sequence**

The sequence in which the characters that are acceptable to a computer are ordered for purposes of sorting, merging, comparing, and for processing indexed files sequentially.

column

A byte position within a print line or within a reference format line. The columns are numbered from 1, by 1, starting at the leftmost position of the line and extending to the rightmost position of the line. A column holds one single-byte character.

*** combined condition**

A condition that is the result of connecting two or more conditions with the AND or the OR logical operator.

*** comment-entry**

An entry in the identification division that is used for documentation and has no effect on execution.

comment line

A source program line represented by an asterisk (*) in the indicator area of the line or by an asterisk followed by greater-than sign (>) as the first character string in the program text area (Area A plus Area B), and any characters from the character set of the computer that follow in Area A and Area B of that line. A comment line serves only for documentation. A special form of comment line represented by a slant (/) in the indicator area of the line and any characters from the character set of the computer in Area A and Area B of that line causes page ejection before the comment is printed.

*** common program**

A program that, despite being directly contained within another program, is permitted to be called from any program directly or indirectly contained in that other program.

compatible date field

The meaning of the term *compatible*, when applied to date fields, depends on the COBOL division in which the usage occurs:

- data division**

Two date fields are compatible if they have identical USAGE and meet at least one of the following conditions:

- They have the same date format.
- Both are windowed date fields, where one consists of only a windowed year, DATE FORMAT YY.

- Both are expanded date fields, where one consists of only an expanded year, DATE FORMAT YYYY.
- One has DATE FORMAT YYXXXX, the other, YYYY.
- One has DATE FORMAT YYYYXXXX, the other, YYYYXX.

A windowed date field can be subordinate to an expanded date group data item. The two date fields are compatible if the subordinate date field has USAGE DISPLAY, starts two bytes after the start of the group expanded date field, and the two fields meet at least one of the following conditions:

- The subordinate date field has a DATE FORMAT pattern with the same number of Xs as the DATE FORMAT pattern of the group date field.
- The subordinate date field has DATE FORMAT YY.
- The group date field has DATE FORMAT YYYYXXXX and the subordinate date field has DATE FORMAT YYYYX.

- procedure division**

Two date fields are compatible if they have the same date format except for the year part, which can be windowed or expanded. For example, a windowed date field with DATE FORMAT YYXXX is compatible with:

- Another windowed date field with DATE FORMAT YYXXX
- An expanded date field with DATE FORMAT YYYYXXXX

compilation unit

See *source unit*

*** compile time**

The time at which COBOL source code is translated by a COBOL compiler to a COBOL object program.

compiler-directing statement

A statement that causes the compiler to take a specific action during compilation. The standard compiler-directing statements are COPY, REPLACE, and USE.

compiler-directive

A directive that causes the compiler to

take a specific action during compilation. COBOL for AIX has one compiler directive, CALLINTERFACE. You can code multiple CALLINTERFACE directives within a program to use specific interface conventions for specific CALL statements.

*** complex condition**

A condition in which one or more logical operators act upon one or more conditions. See also *negated simple condition*, *combined condition*, and *negated combined condition*.

complex ODO

Certain forms of the OCCURS DEPENDING ON clause:

- A variably located item or group: A data item described with an OCCURS clause with the DEPENDING ON phrase, followed by a nonsubordinate data item or group. The group can be an alphanumeric group or a national group.
- A variably located table: A data item described with an OCCURS clause with the DEPENDING ON phrase, followed by a nonsubordinate data item described with an OCCURS clause.
- A table with variable-length elements: A data item described with an OCCURS clause, where a subordinate data item is described with an OCCURS clause with the DEPENDING ON phrase.
- An index name for a table with variable-length elements.
- An element of a table with variable-length elements.

*** condition (expression)**

A status of data at run time for which a truth value can be determined. Where the term 'condition' (*condition-1*, *condition-2*,...) appears in these language specifications in or in reference to 'condition' (*condition-1*, *condition-2*,...) of a general format, it is a conditional expression consisting of either a simple condition optionally parenthesized, or a combined condition consisting of the syntactically correct combination of simple conditions, logical operators, and parentheses, for which a truth value can be determined.

*** conditional expression**

A simple condition or a complex condition specified in an EVALUATE, IF, PERFORM, or SEARCH statement. See also *simple condition* and *complex condition*.

*** conditional phrase**

A conditional phrase specifies the action to be taken upon determination of the truth value of a condition resulting from the execution of a conditional statement.

*** conditional statement**

A statement specifying that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value.

*** conditional variable**

A data item one or more values of which has a condition-name assigned to it.

*** condition-name**

A user-defined word that assigns a name to a subset of values that a conditional variable is permitted to assume; or a user-defined word assigned to a status of an implementor defined switch or device.

*** condition-name condition**

The proposition, for which a truth value can be determined, that the value of a conditional variable is a member of the set of values attributed to a condition-name associated with the conditional variable.

*** configuration section**

A section of the environment division that describes overall specifications of source and object programs, method definitions, and class definitions.

CONSOLE

A COBOL environment-name associated with the operator console.

*** contiguous items**

Items that are described by consecutive entries in the data division, and that bear a definite hierarchic relationship to each other.

contained program

A COBOL program that is nested within another COBOL program.

*** counter**

A data item used for storing numbers or number representations in a manner that

permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitrary positive or negative value.

cs See *currency symbol*.

currency sign value

A character-string that identifies the monetary units stored in a numeric-edited item. Some examples are '\$', 'USD', 'JPY', and 'EUR'. A currency sign value can be defined by either the CURRENCY compiler option or the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the environment division. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency sign value. See also *currency symbol*.

currency symbol

A character used in a PICTURE clause to indicate the position of a *currency sign value* in a numeric-edited item. A currency symbol can be defined by either the CURRENCY compiler option or by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph of the environment division. If the CURRENCY SIGN clause is not specified and the NOCURRENCY compiler option is in effect, the dollar sign (\$) is used as the default currency sign value and currency symbol. Multiple currency symbols and currency sign values can be defined. See also *currency sign value*.

*** current record**

In file processing, the record that is available in the record area associated with a file.

*** current volume pointer**

A conceptual entity that points to the current volume of a sequential file.

D

*** data description entry**

An entry in the data division composed of a level-number followed by a data-name, if required, and then followed by a set of clauses that describe the attributes of a data item or record.

*** data division**

A COBOL division that describes data and files to be processed at run time.

*** data item**

A unit of data (excluding literals) defined by a COBOL program or by the rules for function evaluation.

*** data-name**

A user-defined word that names a data item described in a data description entry. The maximum length of a data-name is 30 bytes. When used in the general formats, 'data-name' represents a word that must not be reference-modified, subscripted or qualified unless specifically permitted by the rules for the format.

date field

Any of the following:

- A data item whose data description entry includes a DATE FORMAT clause.
- A value returned by one of the following intrinsic functions:
 - DATE-OF-INTEGERS
 - DATE-TO-YYYYMMDD
 - DATEVAL
 - DAY-OF-INTEGERS
 - DAY-TO-YYYYDDD
 - YEAR-TO-YYYY
 - YEARWINDOW
- The conceptual data items DATE, DATE YYYYMMDD, DAY, and DAY YYYYDDD of the ACCEPT statement.
- The result of certain arithmetic operations (for details, see "Arithmetic with date fields" on page 260).

The term date field refers to both *expanded date field* and *windowed date field*. See also *nondate*.

date format

The date pattern of a date field, specified either:

- Explicitly, by the DATE FORMAT clause or DATEVAL intrinsic function argument-2
- Implicitly, by statements and intrinsic functions that return date fields (for details, see "Date field" on page 76).

DBCS

See *Double-Byte Character Set (DBCS)*.

DBCS character

Any character defined in an IBM double-byte character set.

DBCS character position

See *character position*.

DBCS data item

A data item described by a PICTURE character-string that contains at least one symbol G or, when the NSYMBOL(DBCS) compiler option is in effect, at least one symbol N. A DBCS data item has usage DISPLAY-1.

*** debugging line**

A debugging line is any line with a 'D' in the indicator area of the line.

*** debugging section**

A section that contains a USE FOR DEBUGGING statement.

*** declaratives**

A set of one or more special purpose sections, written at the beginning of the procedure division, the first of which is preceded by the keyword DECLARATIVES and the last of which is followed by the keyword END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler-directing sentence, followed by a set of zero, one, or more associated paragraphs.

*** de-edit**

The logical removal of all editing characters from a numeric-edited data item in order to determine that item's unedited numeric value.

*** delimited scope statement**

Any statement that includes its explicit scope terminator.

*** delimiter**

A character or a sequence of contiguous characters that identify the end of a string of characters and separate that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

*** descending key**

A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

digit Any of the numerals from 0 through 9. In COBOL, the term is not used in reference to any other symbol.

*** digit position**

The amount of physical storage required to store a single digit. This amount can vary depending on the usage specified in the data description entry that defines the data item.

*** direct access**

The facility to obtain data from storage devices or to enter data into a storage device in such a way that the process depends only on the location of that data and not on a reference to data previously accessed.

display floating-point data item

A data item described with usage DISPLAY and a picture character-string that describes an external floating-point data item. See *floating-point*.

*** division**

There are four divisions in a COBOL program: identification, environment, data, and procedure.

*** division header**

A combination of words followed by a separator period that indicates the beginning of a division. The division headers are:

- IDENTIFICATION DIVISION.
- ENVIRONMENT DIVISION.
- DATA DIVISION.
- PROCEDURE DIVISION.

do-until

In structured programming, a do-until loop will be executed at least once, and until a given condition is true. In COBOL, a TEST AFTER phrase used with the PERFORM statement functions in the same way.

do-while

In structured programming, a do-while loop will be executed if, and while, a given condition is true. In COBOL, a TEST BEFORE phrase used with the PERFORM statement functions in the same way.

double-byte ASCII

An IBM character set that includes DBCS

and single-byte ASCII characters. (Also known as ASCII DBCS.)

double-byte EBCDIC

An IBM character set that includes DBCS and single-byte EBCDIC characters. (Also known as EBCDIC DBCS.)

Double-Byte Character Set (DBCS)

An IBM coded character set in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires two bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

*** dynamic access**

An access mode in which specific logical records can be obtained from or placed into a mass storage file in a nonsequential manner and obtained from a file in a sequential manner during the scope of the same OPEN statement.

E

EBCDIC (Extended Binary-Coded Decimal Interchange Code)

A coded character set consisting of 8-bit coded characters.

EBCDIC character

Any one of the graphic characters or control characters encoded in EBCDIC.

EBCDIC DBCS

See *double-byte EBCDIC*.

edited data item

A data item that has been modified by suppressing zeroes or inserting editing characters.

*** editing character**

A single character or a fixed two-character combination belonging to the following set:

Character	Meaning
	Space
0	Zero
+	Plus
-	Minus

Character	Meaning
CR	Credit
DB	Debit
Z	Zero suppress
*	Check protect
\$	Currency sign
,	Comma (decimal point)
.	Period (decimal point)
/	Forward slash

*** elementary item**

A data item that is described as not being further logically subdivided.

encoding unit

See *character encoding unit*.

end class marker

A combination of words, followed by a separator period, that indicates the end of a COBOL class definition. The end class marker is:

END CLASS class-name.

end method marker

A combination of words, followed by a separator period, that indicates the end of a COBOL method definition. The end method marker is:

END METHOD method-name.

*** end of procedure division**

The physical position of a COBOL procedure division after which no further procedures appear.

end program marker

A combination of words, followed by a separator period, that indicates the end of a COBOL source program. The end program marker is:

END PROGRAM program-name.

*** entry**

Any descriptive set of consecutive clauses written in the identification division, environment division, or data division of a COBOL program.

*** environment division**

A division of a COBOL source unit that describes the computers upon which the source code is compiled and those on which the object code is run. It provides a linkage between the logical concept of

files and their records and the physical aspects of the devices on which files are stored.

environment-name

A name, specified by IBM, that identifies system logical units, printer and card punch control characters, report codes, or program switches. When an environment-name is associated with a mnemonic-name in the environment division, the mnemonic-name can then be substituted in any format in which such substitution is valid.

environment variable

Any of a number of variables that define some aspect of the computing environment, and are accessible to programs that operate in that environment. Environment variables can affect the behavior of programs that are sensitive to the environment in which they operate.

execution time

See *run time*.

execution-time environment

See *runtime environment*.

expanded date field

A date field containing an expanded (four-digit) year. See also *date field* and *expanded year*.

expanded year

A date field that consists only of a four-digit year. Its value includes the century: for example, 1998. Compare with *windowed year*.

*** explicit scope terminator**

A reserved word that terminates the scope of a particular procedure division statement. For example, END-READ.

exponent

A number, indicating the power to which another number (the base) is to be raised. Positive exponents denote multiplication, negative exponents denote division, fractional exponents denote a root of a quantity. In COBOL, an exponential expression is indicated with the symbol '**' followed by the exponent.

*** expression**

An arithmetic or conditional expression.

*** extend mode**

The state of a file after execution of an OPEN statement, with the EXTEND phrase specified for that file, and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

Extensible Markup Language

See *XML*.

*** external data**

The data described in a program as external data items and external file connectors.

*** external data item**

A data item that is described as part of an external record in one or more programs of a run unit and that itself is permitted to be referenced from any program in which it is described.

*** external data record**

A logical record which is described in one or more programs of a run unit and whose constituent data items are permitted to be referenced from any program in which they are described.

external decimal data item

A zoned decimal data item or a national decimal data item. A zoned decimal data item has usage DISPLAY. A national decimal data item has usage NATIONAL. See *zoned decimal data item* and *national decimal data item*.

*** external file connector**

A file connector which is accessible to one or more object programs in the run unit.

external floating-point data item

A display floating-point data item or a national floating-point data item. A display floating-point data item has usage DISPLAY. A national floating-point data item has usage NATIONAL. See *display floating-point data item* and *national floating-point data item*.

*** external switch**

A hardware or software device, defined and named by the implementor, which is used to indicate that one of two alternate states exists.

F

factory data

Data of a factory object. Factory data is allocated once for a class and shared by all instances of the class. Factory data is declared in the working-storage section in the factory paragraph of a class definition. Factory data is equivalent to private static data in Java.

factory method

A method that is supported by a class independently of any object instance. Factory methods are defined in the factory paragraph of the class definition, and are equivalent to public static methods in Java. They are typically used to customize the creation of objects.

*** figurative constant**

A compiler-generated value referenced through the use of certain reserved words.

*** file** A collection of logical records.

*** file attribute conflict condition**

An unsuccessful attempt has been made to execute an input-output operation on a file and the file attributes, as specified for that file in the program, do not match the fixed attributes for that file.

*** file connector**

A storage area which contains information about a file and is used as the linkage between a file-name and a physical file and between a file-name and its associated record area.

*** file control entry**

A SELECT clause and all its subordinate clauses which declare the relevant physical attributes of a file.

*** file-control paragraph**

A paragraph in the environment division in which the data files for a given source unit are declared.

*** file description entry**

An entry in the file section of the data division that is composed of the level indicator FD, followed by a file-name, and then followed by a set of clauses that include the attributes of the file.

*** file-name**

A user-defined word that names a file connector described in a file description

entry or a sort-merge file description entry within the file section of the data division.

*** file organization**

The permanent logical file structure established at the time that a file is created.

***file position indicator**

A conceptual entity that contains the value of the current key within the key of reference for an indexed file, or the record number of the current record for a sequential file, or the relative record number of the current record for a relative file, or indicates that no next logical record exists, or that an optional input file is not available, or that the at end condition already exists, or that no valid next record has been established.

*** file section**

The section of the data division that contains file description entries and sort-merge file description entries together with their associated record descriptions.

file system

The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

*** fixed file attributes**

Information about a file which is established when a file is created and cannot subsequently be changed during the existence of the file. These attributes include the organization of the file (sequential, relative, or indexed), the prime record key, the alternate record keys, the code set, the minimum and maximum record size, the record type (fixed or variable), the collating sequence of the keys for indexed files, the blocking factor, the padding character, and the record delimiter.

*** fixed-length record**

A record associated with a file whose file description or sort-merge description entry requires that all records contain the same number of bytes.

fixed-point item

A numeric data item defined with a PICTURE clause that specifies the location of an optional sign, the number of digits

it contains, and the location of an optional decimal point. The format can be either binary, packed decimal, or external decimal.

floating-point

A format for representing numbers in which a real number is represented by a pair of distinct numerals. In floating-point representation, the real number is the product of the fixed-point part (the first numeral), and a value obtained by raising the implicit floating-point base to a power denoted by the exponent (the second numeral).

For example, a floating-point representation of the number 0.0001234 is: 0.1234 -3, where 0.1234 is the mantissa and -3 is the exponent.

floating-point item

A numeric data item containing a fraction and an exponent. Its value is obtained by multiplying the fraction by the base of the numeric data item raised to the power specified by the exponent.

*** format**

A specific arrangement of a set of data.

*** function**

A temporary data item whose value is determined at the time the function is referenced during the execution of a statement.

*** function-identifier**

A syntactically correct combination of character-strings and separators that references a function. The data item represented by a function is uniquely identified by a function-name with its arguments, if any. A function-identifier can include a reference-modifier. A function-identifier that references an alphanumeric function can be specified anywhere in the general formats that an identifier can be specified, subject to certain restrictions. A function-identifier that references an integer or numeric function can be referenced anywhere in the general formats that an arithmetic expression can be specified.

function-name

A word that names the mechanism whose

invocation, along with required arguments, determines the value of a function.

function-pointer

A data item that can contain the address of a procedure or function, described with a usage of FUNCTION-POINTER.

G

garbage collection

The automatic freeing by the Java runtime system of the memory for objects that are no longer referenced.

| **GDG** See *generation data group (GDG)*.

| **GDS** See *generation data set (GDS)*.

| **generation data group (GDG)**

| A collection of chronologically related *data sets*; each such data set is called a *generation data set (GDS)*.

| **generation data set (GDS)**

| One of the *data sets* in a *generation data group (GDG)*; a GDS is chronologically related to the other data sets in the group.

*** global name**

A name that is declared in only one program but which can be referenced from that program and from any program contained within that program. Condition-names, data-names, file-names, record-names, report-names, and some special registers can be global names.

group item

(1) A data item that is composed of subordinate data items. A group item that is described with an explicit or implicit GROUP-USAGE NATIONAL clause is a national group item. A group that is described without a GROUP-USAGE NATIONAL clause is an alphanumeric group item. See *alphanumeric group item* and *national group item*. (2) When not qualified explicitly or by context as a national group or an alphanumeric group, the term refers to groups in general.

grouping separator

A character used to separate units of digits in numbers for ease of reading. The default is the character comma.

H

header label

(1) A file label that precedes the data records on a unit of recording media. (2) Synonym for beginning-of-file label.

hide (a method)

To redefine (in a subclass) a factory or static method defined with the same method-name in a parent class. Thus, the method in the subclass *hides* the method in the parent class.

*** high order end**

The leftmost character of a string of characters.

I**IBM extensions**

COBOL syntax and semantics specified by IBM, rather than by Standard COBOL 85.

ICU See *International Components for Unicode (ICU)*.

identification division

One of the four main component parts of a COBOL program, class definition, or method definition. The identification division identifies the program, class, or method. The identification division can include the following documentation: author name, installation, or date.

*** identifier**

Syntax that references a resource, such as a data item. An identifier that refers to data item includes the data-name and optionally includes qualifiers, subscripting, and reference modification.

*** imperative statement**

A statement that specifies an unconditional action to be taken or a conditional statement that is delimited by its explicit scope terminator (a delimited scope statement). An imperative statement can consist of a sequence of imperative statements.

*** implicit scope terminator**

A separator period that terminates the scope of any preceding unterminated statement, or a phrase of a statement that by its occurrence indicates the end of the scope of any statement contained within the preceding phrase.

*** index**

A computer storage area or register, the

content of which represents the identification of a particular element in a table.

*** index data item**

A data item in which the values associated with an index-name can be stored in a form specified by the implementor.

indexed data-name

An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

*** indexed file**

A file with indexed organization.

*** indexed organization**

The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

indexing

Subscripting using index-names.

*** index-name**

A user-defined word that names an index associated with a specific table.

inheritance

A mechanism for using the implementation of a class (the *superclass*) as the basis for a new class (a *subclass*). Each *subclass* inherits from exactly one class. The inherited class can itself be a subclass that inherits from another class.

COBOL for AIX does not support multiple inheritance. It supports the Java object model, which provides single inheritance.

*** initial program**

A program that is placed into an initial state every time the program is called in a run unit.

*** initial state**

The state of a program when it is first called in a run unit.

inline

In a program, instructions that are executed sequentially, without branching to routines, subroutines, or other programs.

*** input file**

A file that is opened in the INPUT mode.

*** input mode**

The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement, without the REEL or UNIT phrase for that file.

*** input-output file**

A file that is opened in the I-O mode.

*** input-output section**

The section of the environment division that names the files and the external media required by a program or method and that provides information required for transmission and handling of data at run time.

*** input-output statement**

A statement that causes files to be processed by performing operations upon individual records or upon the file as a unit. The input-output statements are: ACCEPT (with the identifier phrase), CLOSE, DELETE, DISPLAY, OPEN, READ, REWRITE, SET (with the TO ON or TO OFF phrase), START, and WRITE.

*** input procedure**

A set of statements, to which control is given during the execution of a SORT statement, for the purpose of controlling the release of specified records to be sorted.

instance data

Data defining the state of an object instance. Instance data is declared in the working-storage section of the object paragraph of a class definition. Also called *object instance data*. Each object instance has its own copy of instance data. Instance data is equivalent to private nonstatic member data in a Java class.

instance method

A method defined in the object paragraph of a class definition. Instance methods are equivalent to public nonstatic methods in Java.

*** integer**

(1) A numeric literal that does not include any digit positions to the right of the decimal point. (2) A numeric data item defined in the data division that does not include any digit positions to the right of the decimal point. (3) A numeric function

whose definition provides that all digits to the right of the decimal point are zero in the returned value for any possible evaluation of the function.

*** integer function**

A function whose category is numeric and whose definition does not include any digit positions to the right of the decimal point.

interlanguage communication (ILC)

The ability of routines written in different programming languages to communicate. ILC support allows the application writer to readily build applications from component routines written in a variety of languages.

intermediate result

An intermediate field containing the results of a succession of arithmetic operations.

*** internal data**

The data described in a program excluding all external data items and external file connectors. Items described in the linkage section of a program are treated as internal data.

*** internal data item**

A data item which is described in one program in a run unit. An internal data item can have a global name.

internal decimal data item

A data item that is described with usage PACKED-DECIMAL or COMP-3 and a PICTURE character-string that defines the item as numeric (a valid combination of symbols 9, S, P, or V). Synonymous with *packed decimal item*.

*** internal file connector**

A file connector that is accessible to only one object program in the run unit.

International Components for Unicode (ICU)

An open source development project sponsored, supported, and used by IBM. ICU libraries provide robust and full-featured Unicode services on a wide variety of platforms, including AIX.

internal floating-point data item

A data item that is described with usage COMP-1 or COMP-2. COMP-1 defines a single-precision floating-point data item. COMP-2 defines a double-precision

floating-point data item. There is no PICTURE clause associated with an internal floating-point data item.

intrinsic function

A function defined as part of the COBOL language. In some programming languages, this is called a built-in function.

*** invalid key condition**

A condition, at run time, caused when a specific value of the key associated with an indexed or relative file is determined to be invalid.

*** I-O mode**

The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** I-O status**

A conceptual entity which contains the two-character value indicating the resulting status of an input-output operation. This value is made available to the program through the use of the FILE STATUS clause in the file control entry for the file.

J

Java Native Interface (JNI)

A programming interface that allows Java code running inside a Java virtual machine (JVM) to interoperate with applications and libraries written in other programming languages.

K

K When referring to storage capacity, two to the tenth power; 1024 in decimal notation.

*** key** A data item that identifies the location of a record, or a set of data items which serve to identify the ordering of data.

*** key of reference**

The key, either prime or alternate, currently being used to access records within an indexed file.

*** keyword**

A reserved word or function-name whose presence is required when the format in which the word appears is used in a source unit.

kilobyte (KB)

One kilobyte equals 1024 bytes.

L

*** language-name**

A system-name that specifies a particular programming language.

last-used state

The state of storage in which internal values remain the same as when the program was exited (are not reset to their initial values on reentry).

*** letter**

A character belonging to one of the following two sets:

- Uppercase letters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
- Lowercase letters: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z

*** level indicator**

Two alphabetic characters that identify a specific type of file or a position in a hierarchy. The level indicators in the data division are: CD, FD, and SD.

*** level-number**

A user-defined word, expressed as a two-digit number, which indicates the hierarchical position of a data item or the special properties of a data description entry. Level-numbers in the range from 1 through 49 indicate the position of a data item in the hierarchical structure of a logical record. Level-numbers in the range 1 through 9 can be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

*** library-name**

A user-defined word that names a COBOL library that is to be used by the compiler for a given compilation.

*** library text**

A sequence of text words, comment lines, the separator space, or the separator pseudo-text delimiter in a COBOL library.

Lilian date

The number of days since the beginning of the Gregorian calendar. Day one is Friday, October 15, 1582. The Lilian date

format is named in honor of Luigi Lilio, the creator of the Gregorian calendar.

*** LINAGE-COUNTER**

A special register whose value points to the current position within the page body.

linkage section

The section in the data division of an activated unit (a called program or an invoked method) that describes data items available from the activating unit (a program or a method). These data items can be referred to by both the activated unit and the activating unit.

literal

A character-string whose value is specified either by the ordered set of characters comprising the string, or by the use of a figurative constant.

little-endian

The default format that Intel processors use to store binary data. In this format, the most significant digit is at the highest address. See also *big-endian*.

locale A set of attributes for a program execution environment that indicates culturally sensitive considerations, such as: character code page, collating sequence, date/time format, monetary value representation, numeric value representation, or language.

local-storage section

The section of the data division that defines storage that is allocated and freed on a per-invocation basis, depending on the value assigned in their VALUE clauses.

*** logical operator**

One of the reserved words AND, OR, or NOT. In the formation of a condition, either AND or OR, or both, can be used as logical connectives. NOT can be used for logical negation.

*** logical record**

The most inclusive data item. The level-number for a record is 01. A record can be either an elementary item or a group of items. The term is synonymous with record.

*** low order end**

The rightmost character of a string of characters.

M

main program

In a hierarchy of programs and subroutines, the first program to receive control when the programs are run.

*** mass storage**

A storage medium in which data can be organized and maintained in both a sequential and nonsequential manner.

*** mass storage device**

A device having a large storage capacity; for example, magnetic disk, magnetic drum.

*** mass storage file**

A collection of records that is assigned to a mass storage medium.

MBCS See *multibyte character set (MBCS)*.

*** megabyte (M)**

One megabyte equals 1,048,576 bytes.

*** merge file**

A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

method

Procedural code that defines one of the operations supported by an object. Method procedural code is executed by a COBOL INVOKE statement on a specific object instance. A method can be invoked by a Java invocation expression. A method can be a *factory method* or an *instance method*.

method identification entry

An entry in the METHOD-ID paragraph of the identification division that contains clauses that specify the method-name and assign selected attributes to the method definition.

method invocation

(1) The act of invoking a method. (2) The programming language syntax used to invoke a method (the INVOKE statement in COBOL, a method invocation expression in Java).

method-name

A name that identifies a method, specified as the content of an alphanumeric or national literal in the METHOD-ID paragraph, and as the content of an

alphanumeric literal, national literal, alphanumeric data item, or data item of category national in the INVOKE statement.

method hiding

See *hide*.

method overloading

See *overload*.

method overriding

See *override*.

*** mnemonic-name**

A user-defined word that is associated in the environment division with a specified implementor-name.

multibyte character

Any character that is represented in 2 or more bytes in a multibyte character set. For example, a DBCS character or any UTF-8 character that is represented in two or more bytes. UTF-16 characters are not multibyte characters because UTF-16 is not a multibyte character set.

multibyte character set (MBCS)

A coded character set that is composed of characters represented in a varying number of bytes. Examples are: Extended UNIX Code, UTF-8, and character sets composed of a mixture of single-byte and double-byte EBCDIC or ASCII characters.

N

namespace

See *XML namespace*.

national character

Any character represented in UTF-16.

national character data

A general reference to data represented in UTF-16.

national character position

See *character position*.

national data

See *national character data*.

national data item

A data item of class national. Class national includes categories national, national-edited, and numeric-edited with USAGE NATIONAL.

national decimal data item

A data item described by a PICTURE

character-string that contains valid combinations of picture symbols 9, S, P, and V. A national decimal data item is an external decimal data item that has usage NATIONAL.

national-edited data item

A data item described by a PICTURE character-string that contains the symbol N and at least one of the simple insertion symbols B, 0, and /. A national-edited data item has usage NATIONAL.

national floating-point data item

A data item described with usage NATIONAL and a picture character-string that describes a floating-point data item. See *floating-point*.

national group item

A group item that is explicitly or implicitly described with a GROUP-USAGE clause with the NATIONAL phrase. A national group is processed as though it were defined as an elementary data item of category national for operations such as INSPECT, STRING, and UNSTRING. This ensures correct padding and truncation of national characters, as opposed to defining data items described with USAGE NATIONAL within an alphanumeric group item. For operations that require processing of the elementary items within a group, such as MOVE CORRESPONDING, ADD CORRESPONDING, and INITIALIZE *identifier*, a national group is processed using group semantics.

*** native character set**

The implementor-defined character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** native collating sequence**

The implementor-defined collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

*** negated combined condition**

The 'NOT' logical operator immediately followed by a parenthesized combined condition.

*** negated simple condition**

The 'NOT' logical operator immediately followed by a simple condition.

nested program

A program that is directly contained within another program.

*** next executable sentence**

The next sentence to which control will be transferred after execution of the current statement is complete.

*** next executable statement**

The next statement to which control will be transferred after execution of the current statement is complete.

*** next record**

The record that logically follows the current record of a file.

*** noncontiguous items**

Elementary data items in the working-storage and linkage sections that bear no hierarchic relationship to other data items.

nondate

Any of the following:

- A data item whose date description entry does not include the DATE FORMAT clause
- A literal
- A date field that has been converted using the UNDATE function
- A reference-modified date field
- The result of certain arithmetic operations that can include date field operands; for example, the difference between two compatible date fields

null

A figurative constant that represents a value used to indicate that a pointer data item does not contain a valid address or that an object reference does not reference an object. NULLS can be used wherever NULL can be used.

*** numeric character**

A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

numeric data item

(1) A data item whose description restricts its content to a value represented by characters chosen from the digits from '0' through '9'; if signed, the item can also contain a '+', '-', or other representation of an operational sign. (2) A data item of class numeric and category numeric, internal floating-point, or external floating-point, possibly limited to specific

data categories or specific data descriptions by detailed specifications. A numeric data item can have usage DISPLAY, NATIONAL, PACKED-DECIMAL, BINARY, COMP, COMP-1, COMP-2, COMP-3, COMP-4, or COMP-5.

numeric-edited data item

A data item that contains numeric data in a form suitable for use in printed output. It can consist of external decimal digits from 0 through 9, the decimal separator, commas, the currency sign, sign control characters, and other editing characters. A numeric-edited data item can be represented in usage DISPLAY or usage NATIONAL.

*** numeric function**

A function whose class and category are numeric but which for some possible evaluation does not satisfy the requirements of integer functions.

*** numeric literal**

A literal composed of one or more numeric characters. It can contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

O**object**

An entity that has state (its data values) and operations (its methods). An object is a way to encapsulate state and behavior.

object code

Output from a compiler or assembler that is itself executable machine code or is suitable for processing to produce executable machine code.

*** object-computer**

The name of an environment division paragraph in which the computer environment, within which the program is executed, is described.

object deck

A portion of an object program suitable as input to a linkage editor. The term is synonymous with *object module* and *text deck*.

object instance

A single object, of possibly many,

instantiated from the specifications in the object paragraph of a COBOL class definition. An object instance has a copy of all the data described in its class definition and all inherited data. The methods associated with an object instance includes the methods defined in its class definition and all inherited methods.

An object instance can be an instance of a Java class.

object module

Synonym for *object deck* or *text deck*.

*** object of entry**

A set of operands and reserved words, within a data division entry of a COBOL program, that immediately follows the subject of the entry.

*** object program**

A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program or on the methods of an object-oriented class definition. Where there is no danger of ambiguity, the word 'program' alone can be used in place of the phrase 'object program'.

object reference

A data item that can contain the information needed to invoke or refer to an object. An object reference is defined in COBOL with the OBJECT REFERENCE phrase in the USAGE clause of a data description entry. See also *typed object reference* and *universal object reference*.

*** object time**

The time at which an object program is executed. The term is synonymous with the terms *execution time* and *run time*.

*** obsolete element**

A COBOL language element in Standard COBOL 85 that was deleted from Standard COBOL 2002.

ODO object

In the example below,

WORKING-STORAGE SECTION

```
01 TABLE-1.
   05 X                                PICS9.
   05 Y OCCURS 3 TIMES
      DEPENDING ON X                PIC X.
```

X is the object of the OCCURS DEPENDING ON clause (ODO object). The value of the ODO object determines how many of the ODO subject appear in the table.

ODO subject

In the example above, Y is the subject of the OCCURS DEPENDING ON clause (ODO subject). The number of Y ODO subjects that appear in the table depends on the value of X.

*** open mode**

The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O, or EXTEND.

operand

Data that is operated upon. In this document, any lowercase word (or words) that appears in a statement or entry format is an operand in that it is a reference to the data identified by that word (or words).

*** operational sign**

An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

optional file

A file that is declared as being not necessarily available each time the object program is executed.

*** optional word**

A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source unit.

*** output file**

A file that is opened in either the OUTPUT mode or EXTEND mode.

*** output mode**

The state of a file after execution of an

OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement without the REEL or UNIT phrase for that file.

*** output procedure**

A set of statements to which control is given during execution of a SORT statement after the sort function is completed, or during execution of a MERGE statement after the merge function reaches a point at which it can select the next record in merged order when requested.

overflow condition

A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overload

To define a method with the same name as another method available in the same class, but with a different signature. See also *signature*.

override

To redefine (in a subclass) an instance method inherited from a parent class.

P

package

In Java, a group of related classes that can be imported individually or as a whole.

packed decimal item

See *internal decimal item*.

padding character

An alphanumeric or national character or literal used to fill the unused character positions in a physical record.

page

A vertical division of output data representing a physical separation of such data, the separation being based on internal logical requirements or external characteristics of the output medium.

*** page body**

That part of the logical page in which lines can be written or spaced.

*** paragraph**

In the procedure division, a paragraph-name followed by a separator period and by zero, one, or more sentences. In the identification and

environment divisions, a paragraph header followed by zero, one, or more entries.

*** paragraph header**

A reserved word, followed by the separator period, that indicates the beginning of a paragraph.

*** paragraph-name**

A user-defined word that identifies and begins a paragraph in the procedure division.

password

A unique string of characters that a program, computer operator, or user must supply to meet security requirements before gaining access to data.

*** phrase**

An ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

*** physical record**

See *block*.

pointer data item

A data item in which address values can be stored. Data items are explicitly defined as pointers with the USAGE IS POINTER clause. ADDRESS OF special registers are implicitly defined as pointer data items. Pointer data items can be compared for equality or moved to other pointer data items.

portability

The ability to transfer an application from one application platform to another with relatively few changes to the source code.

*** prime record key**

A key whose contents uniquely identify a record within an indexed file.

*** priority-number**

A user-defined word that classifies sections in the procedure division for purposes of segmentation. Priority-numbers can contain only the characters '0','1', . . . , '9'.

private

In object orientation, data that is accessible only by methods of the class that defines the data. Instance data is accessible only by instance methods; factory data is accessible only by factory

methods. Thus, instance data is private to instance methods defined in the same class definition; factory data is private to factory methods defined in the same class definition.

*** procedure**

A paragraph or group of logically successive paragraphs, or a section or group of logically successive sections, within the procedure division.

*** procedure branching statement**

A statement that causes the explicit transfer of control to a statement other than the next executable statement in the sequence in which the statements are written in the source unit. The procedure branching statements are: ALTER, CALL, EXIT, EXIT PROGRAM, GO TO, MERGE (with the OUTPUT PROCEDURE phrase), PERFORM, SORT (with the INPUT PROCEDURE or OUTPUT PROCEDURE phrase), and XML PARSE.

procedure division

The division of a program or method that contains procedural statements for performing operations at run time.

*** procedure-name**

A user-defined word that is used to name a paragraph or section in the procedure division. It consists of a paragraph-name (which can be qualified) or a section-name.

procedure pointer

A data item in which a pointer to an entry point can be stored. A data item defined with the USAGE IS PROCEDURE-POINTER clause contains the address of a procedure entry point.

program-name

In the identification division and the end program marker, a user-defined word or an alphanumeric literal that identifies a COBOL source program.

*** pseudo-text**

A sequence of text words, comment lines, or the separator space in a source unit or COBOL library bounded by, but not including, pseudo-text delimiters.

*** pseudo-text delimiter**

Two contiguous equal sign characters (==) used to delimit pseudo-text.

*** punctuation character**

A character that belongs to the following set:

Character	Meaning
,	Comma
;	Semicolon
:	Colon
.	Period (full stop)
"	Quotation mark
(Left parenthesis
)	Right parenthesis
	Space
=	Equal sign

Q

QSAM (Queued Sequential Access Method)

An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

*** qualified data-name**

An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF and IN followed by a data-name qualifier.

*** qualifier**

(1) A data-name or a name associated with a level indicator which is used in a reference either together with another data-name which is the name of an item that is subordinate to the qualifier or together with a condition-name. (2) A section-name that is used in a reference together with a paragraph-name specified in that section. (3) A library-name that is used in a reference together with a text-name associated with that library.

R

*** random access**

An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from, or placed into a relative or indexed file.

*** record**

See *logical record*.

*** record area**

A storage area allocated for the purpose of processing the record described in a record description entry in the file section of the data division. In the file section, the current number of character positions in the record area is determined by the explicit or implicit RECORD clause.

*** record description**

See *record description entry*.

*** record description entry**

The total set of data description entries associated with a particular record. The term is synonymous with record description.

record key

A key whose contents identify a record within an indexed file.

*** record-name**

A user-defined word that names a record described in a record description entry in the data division of a COBOL program.

*** record number**

The ordinal number of a record in the file whose organization is sequential.

recording mode

The format of the logical records in a file. Recording mode can be F (fixed-length), V (variable-length), S (spanned), or U (undefined).

recursion

A program calling itself or being directly or indirectly called by a one of its called programs.

recursively capable

A program is recursively capable (can be called recursively) if the RECURSIVE clause is on the PROGRAM-ID statement.

reel

A discrete portion of a storage medium that contains part of a file, all of a file, or any number of files. The term is synonymous with unit and volume.

reentrant

The attribute of a program or routine that allows more than one user to share a single copy of a load module.

*** reference format**

A format that provides a standard method for writing COBOL source code.

reference modification

A method of defining a new data item by specifying the leftmost character position and length relative to the leftmost character position of another data item.

*** reference-modifier**

A syntactically correct combination of character-strings and separators that defines a unique data item. It includes a delimiting left parenthesis separator, the leftmost character position, a colon separator, optionally a length, and a delimiting right parenthesis separator.

*** relation**

See *relational operator* or *relation condition*.

*** relation character**

A character that belongs to the following set:

Character	Meaning
>	Greater than
<	Less than
=	Equal to

*** relation condition**

The proposition, for which a truth value can be determined, that the value of an arithmetic expression, data item, alphanumeric literal, or index-name has a specific relationship to the value of another arithmetic expression, data item, alphanumeric literal, or index name. See also *relational operator*.

*** relational operator**

A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators and their meanings are:

Character	Meaning
IS GREATER THAN	Greater than
IS >	Greater than
IS NOT GREATER THAN	Not greater than
IS NOT >	Not greater than

Character	Meaning
IS LESS THAN	Less than
IS <	Less than
IS NOT LESS THAN	Not less than
IS NOT <	Not less than
IS EQUAL TO	Equal to
IS =	Equal to
IS NOT EQUAL TO	Not equal to
IS NOT =	Not equal to
IS GREATER THAN OR EQUAL TO	Greater than or equal to
IS >=	Greater than or equal to
IS LESS THAN OR EQUAL TO	Less than or equal to
IS <=	Less than or equal to

*** relative file**

A file with relative organization.

*** relative key**

A key whose contents identify a logical record in a relative file.

*** relative organization**

The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

*** relative record number**

The ordinal number of a record in a file whose organization is relative. This number is treated as a numeric literal that is an integer.

*** reserved word**

A COBOL word specified in the list of words that can be used in a COBOL source unit, but that must not appear in the program as user-defined words or system-names.

*** resource**

A facility or service, controlled by the operating system, that can be used by an executing program.

*** resultant identifier**

A user-defined data item that is to contain the result of an arithmetic operation.

routine

A set of statements in a COBOL program that causes the computer to perform an operation or series of related operations.

*** routine-name**

A user-defined word that identifies a procedure written in a language other than COBOL.

RSD file system

The record sequential delimited file system is a workstation file system that supports sequential files. An RSD file supports all COBOL data types in fixed-length records, can be edited by most file editors, and can be read by programs written in other languages.

*** run time**

The time at which an object program is executed. The term is synonymous with *object time*.

runtime environment

The environment in which a COBOL program executes.

*** run unit**

A stand-alone object program, or several object programs, that interact via COBOL CALL or INVOKE statements and function at run time as an entity.

S

SBCS (Single Byte Character Set)

See *Single Byte Character Set (SBCS)*.

scope terminator

A COBOL reserved word that marks the end of certain procedure division statements. It can be either explicit (END-ADD, for example) or implicit (a separator period, for example).

| SdU file system

| The SdU (SMARTdata Utilities) file
| system supports sequential, indexed, and
| relative files. It does not support LINE
| SEQUENTIAL files.

*** section**

A set of zero, one or more paragraphs or entities, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

*** section header**

A combination of words followed by a separator period that indicates the beginning of a section. For example, WORKING-STORAGE SECTION.

*** section-name**

A user-defined word that names a section in the procedure division.

segmentation

A feature of COBOL for AIX that is based on the Standard COBOL 85 segmentation module. The segmentation feature uses priority-numbers in section headers to assign sections to fixed segments or independent segments. Segment classification affects whether procedures contained in a segment receive control in initial state or last-used state.

*** sentence**

A sequence of one or more statements, the last of which is terminated by a separator period.

*** separately compiled program**

A program that, together with its contained programs, is compiled separately from all other programs.

*** separator**

A character or two contiguous characters used to delimit character-strings.

*** separator comma**

A comma (,) followed by a space used to delimit character-strings.

*** separator period**

A period (.) followed by a space used to delimit character-strings.

*** separator semicolon**

A semicolon (;) followed by a space used to delimit character-strings.

*** sequential access**

An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

*** sequential file**

A file with sequential organization.

*** sequential organization**

The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

serial search

A search in which the members of a set

are consecutively examined, beginning with the first member and ending with the last.

*** 77-level-description-entry**

A data description entry that describes a noncontiguous data item with the level-number 77.

SFS file system

The Encina Structured File Server file system is a record-oriented file system that supports sequential, relative, and key-indexed file access. SFS files can be shared with PL/I applications.

*** sign condition**

The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

signature

The name of a method and the number and types of its formal parameters.

*** simple condition**

Any single condition chosen from the set:

- Relation condition
- Class condition
- Condition-name condition
- Switch-status condition
- Sign condition

Single Byte Character Set (SBCS)

A set of characters in which each character is represented by a single byte. See also *EBCDIC (Extended Binary-Coded Decimal Interchange Code)*.

slack bytes (within records)

Bytes inserted by the compiler between data items to ensure correct alignment of some elementary data items. Slack bytes contain no meaningful data. The SYNCHRONIZED clause instructs the compiler to insert slack bytes when they are needed for proper alignment.

slack bytes (between records)

Bytes inserted by the programmer between blocked logical records of a file, to ensure correct alignment of some elementary data items. In some cases, slack bytes between records improve performance for records processed in a buffer.

*** sort file**

A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

*** sort-merge file description entry**

An entry in the file section of the data division that is composed of the level indicator SD, followed by a file-name, and then followed clauses that describe the attributes of the sort-merge file.

source unit

A unit of COBOL source code that can be separately compiled: a program or a class definition. Also known as *compilation unit*.

special character

A character that belongs to the following set:

Character	Meaning
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (forward slash)
=	Equal sign
\$	Currency sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
'	Apostrophe
(Left parenthesis
)	Right parenthesis
>	Greater than
<	Less than
:	Colon
_	Underscore

SPECIAL-NAMES

The name of an environment division paragraph in which environment-names are related to user-specified mnemonic-names.

*** special registers**

Certain compiler-generated storage areas whose primary use is to store information produced in conjunction with the use of a specific COBOL feature.

Standard COBOL 85

The COBOL language defined by the ANSI and ISO standards .

Standard COBOL 2002

The COBOL language defined by the following standards:

- INCITS/ISO/IEC 1989-2002, Information Technology - Programming Languages - COBOL
- ISO/IEC 1989:2002, Information technology -- Programming languages -- COBOL

*** statement**

A COBOL language construct that specifies one or more actions to be performed. Statements can be procedural statements or compiler-directing statements. An example of a procedural statement is the ADD statement; an example of a compiler-directing statement is the USE statement.

STL file system

The standard language file system is the native workstation file system for COBOL and PL/I. This system supports sequential, relative, and indexed files.

structured programming

A technique for organizing and coding a computer program in which the program comprises a hierarchy of segments, each segment having a single entry point and a single exit point. Control is passed downward through the structure without unconditional branches to higher levels of the hierarchy.

subclass

A class that inherits from another class. When two classes in an inheritance relationship are considered together, the subclass is the inheriting class; the *superclass* is the inherited class.

A subclass is also referred to as a child class or derived class.

*** subject of entry**

An operand or reserved word that appears immediately following the level indicator or the level-number in a data division entry.

*** subprogram**

Any called program.

*** subscript**

An occurrence number represented by either an integer, a data-name optionally followed by an integer with the operator + or -, or an index-name optionally followed by an integer with the operator + or -, that identifies a particular element in a table. A subscript can be the word ALL when the subscripted identifier is used as a function argument for a function allowing a variable number of arguments.

*** subscripted data-name**

An identifier that is composed of a data-name followed by one or more subscripts enclosed in parentheses.

superclass

A class that is inherited by another class. When two classes in an inheritance relationship are considered together, the subclass is the inheriting class; the *superclass* is the inherited class.

The superclass is also referred to as the parent class.

surrogate pair

In the UTF-16 format of Unicode, a pair of encoding units that together represents a single Unicode graphic character. The first unit of the pair is called a *high surrogate* and the second a *low surrogate*. The code value of a high surrogate is in the range X'D800' through X'DBFF'. The code value of a low surrogate is in the range X'DC00' through X'DFFF'. Surrogate pairs provide for more characters than the 65,536 characters that fit in the Unicode 16-bit coded character set.

switch-status condition

The proposition, for which a truth value can be determined, that an UPSI switch, capable of being set to an 'on' or 'off' status, has been set to a specific status.

*** symbolic-character**

A user-defined word that specifies a user-defined figurative constant.

syntax

(1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. (4)

The relationship among symbols. (5) The rules for the construction of a statement.

*** system-name**

A COBOL word that is used to communicate with the operating environment.

T

*** table**

A set of logically consecutive items of data that are defined in the data division by means of the OCCURS clause.

*** table element**

A data item that belongs to the set of repeated items comprising a table.

text deck

Synonym for *object deck* or *object module*.

*** text-name**

A user-defined word that identifies library text.

*** text word**

A character or a sequence of contiguous characters between margin A and margin R in COBOL source code. A text word can be:

- A separator, except for: space; a pseudo-text delimiter; and the opening and closing delimiters for alphanumeric literals. The right parenthesis and left parenthesis characters, regardless of context within the library, source unit, or pseudo-text, are always considered text words.
- A literal including, in the case of alphanumeric literals, the opening quotation mark and the closing quotation mark that bound the literal.
- Any other sequence of contiguous COBOL characters except comment lines and the word 'COPY', bounded by separators, that are neither a separator nor a literal.

trailer-label

(1) A file label that follows the data records on a unit of recording medium.
(2) Synonym for end-of-file label.

*** truth value**

The representation of the result of the evaluation of a condition in terms of one of two values: true or false.

typed object reference

An object reference data item that can reference only an object of a specified class or one of its subclasses.

U*** unary operator**

A plus (+) or a minus (-) sign that precedes a variable or a left parenthesis in an arithmetic expression and that has the effect of multiplying the expression by +1 or -1, respectively.

Unicode

A coded character set that encodes all the characters required for the written expression of any of the languages of the modern world. There are multiple formats for representing Unicode, including UTF-8, UTF-16, and UTF-32. COBOL for AIX supports Unicode using UTF-16 big-endian format as the representation for the national data type.

unit A module of direct access, the dimensions of which are determined by IBM.

universal object reference

An object reference data item that can contain a reference to an object of any class.

*** unsuccessful execution**

The attempted execution of a statement that does not result in the execution of all the operations specified by that statement.

UPSI switch

A program switch that performs the functions of a hardware switch. Eight are provided: UPSI-0 through UPSI-7.

*** user-defined word**

A COBOL word that must be supplied by the user to satisfy the format of a clause or statement. The maximum length of a user-defined word is 30 bytes.

V*** variable**

A data item whose value can be changed by the application at run time.

variable-length item

A group item that contains a table described with the *DEPENDING* phrase of the *OCCURS* clause.

*** variable-length record**

A record associated with a file whose file description or sort-merge description entry permits records to contain a varying number of character positions.

*** variable-occurrence data item**

A variable-occurrence data item is a table element which is repeated a variable number of times. Such an item must contain an *OCCURS DEPENDING ON* clause in its data description entry, or be subordinate to such an item.

variably located group

A group item following, and not subordinate to, a variable-length table in the same level-01 record. A variably located group can be an alphanumeric group or a national group.

variably located item

A data item following, and not subordinate to, a variable-length table in the same level-01 record.

volume

A module of external storage. For tape devices it is a reel; for direct-access devices it is a unit.

volume switch procedures

System procedures executed automatically when the end of a unit or reel has been reached before end-of-file has been reached.

W**white space characters**

Characters that introduce space into a document. They are:

- Space
- Horizontal tabulation
- Carriage return
- Line feed
- Next line

as named in the Unicode Standard.

windowed date field

A date field containing a windowed (two-digit) year. See also *date field* and *windowed year*.

windowed year

A date field that consists only of a two-digit year. This two-digit year can be interpreted using a century window. For

example, 05 could be interpreted as 2005. See also *century window*. Compare with *expanded year*.

*** word**

A character-string that forms a user-defined word, a system-name, a reserved word, or a function-name.

*** working-storage section**

The section of the data division that describes working-storage data items, composed either of noncontiguous items or working-storage records, or both.

X

XML Extensible Markup Language. A metalanguage for defining markup languages that was derived from and is a subset of SGML. XML omits the more complex and less-used parts of SGML and makes it much easier to:

- Write applications to handle document types
- Author and manage structured information
- Transmit and share structured information across diverse computing systems

XML is being developed under the auspices of the World Wide Web Consortium (W3C).

XML data

Data that is organized into a hierarchical structure with XML elements. The data definitions are defined in XML element type declarations.

XML declaration

XML text that specifies characteristics of the XML document such as the version of XML being used and the encoding of the document.

XML document

A data object that is well formed as defined by the W3C XML specification.

XML namespace

A mechanism, defined by the W3C XML Namespace specifications, that limits the scope of a collection of element names and attribute names. A uniquely chosen XML namespace ensures the unique identity of an element name or attribute

name across multiple XML documents or multiple contexts within an XML document.

Z

zoned decimal data item

A data item described by a PICTURE character-string that contains valid combinations of picture symbols 9, S, P, and V. A zoned decimal data item is an external decimal data item that has usage DISPLAY. See *external decimal data item* and *national decimal data item*.

List of resources

COBOL for AIX

Installation Guide, GC27-3604

Language Reference, SC27-3602

Programming Guide, SC27-3601

VSAM File System Reply Messages, SC27-3603

Support

If you have a problem using COBOL for AIX, see the following site, which provides up-to-date support information: www.ibm.com/software/awdtools/cobol/aix/support/.

Related publications

AIX

Commands Reference: Volume 1, a - c, SC23-5243

Commands Reference: Volume 2, d - h, SC23-5244

Commands Reference: Volume 3, i - m, SC23-5245

Commands Reference: Volume 4, n - r, SC23-5246

Commands Reference: Volume 5, s - u, SC23-5247

Commands Reference: Volume 6, v - z, SC23-5248

General Programming Concepts: Writing and Debugging Programs, SC23-5259

DB2 for Linux, UNIX, and Windows

Command Reference, SC27-2439

Database Administration Concepts and Configuration Reference, SC27-2442

SQL Reference Volume 1, SC27-2456

SQL Reference Volume 2, SC27-2457

See also the Information Center:
publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp.

Java

IBM SDK for Java - Tools Documentation,
publib.boulder.ibm.com/infocenter/javasdk/tools/index.jsp

The Java Language Specification, Third Edition, by Gosling et al., java.sun.com/docs/books/jls/

The Java Native Interface, java.sun.com/j2se/1.5.0/docs/guide/jni/

Java Technology Edition SDK User Guides,
www.ibm.com/developerworks/java/jdk/aix/service.html

JDK 5.0 Documentation, java.sun.com/j2se/1.5.0/docs/

TXSeries for Multiplatforms

Encina Administration Guide Volume 2: Server Administration, SC09-4474

Encina SFS Programming Guide, SC09-4483

See also the Information Center:
publib.boulder.ibm.com/infocenter/txformp/v7r1/index.jsp.

Unicode and character representation

Unicode, www.unicode.org/

International Components for Unicode: Converter Explorer, demo.icu-project.org/icu-bin/convexp/

Character Data Representation Architecture: Reference and Registry, SC09-2190

XML

Extensible Markup Language (XML),
www.w3.org/XML/

Namespaces in XML 1.0, www.w3.org/TR/xml-names/

Namespaces in XML 1.1, www.w3.org/TR/xml-names11/

XML specification, www.w3.org/TR/xml/

Index

Special characters

- (minus)
 - insertion character 213, 214
 - SIGN clause 223
 - symbol in PICTURE clause 203
- , (comma)
 - insertion character 212
 - symbol in PICTURE clause 201, 203
- : (colon)
 - description 40
 - file concatenation 131
 - required use of 546
- / (slash)
 - insertion character 212
 - symbol in PICTURE clause 203
- (/ or *)> comment line 51
- (period) symbol in PICTURE clause 201
- \$ (default currency symbol)
 - in PICTURE clause 203
 - insertion character 213, 214
 - symbol in PICTURE clause 201
- * symbol in PICTURE clause 201
- *> (floating comment indicator) 51
- *CBL (*CONTROL) statement 538
- *CONTROL (*CBL) statement 538
- > (greater than) 268
- >= (greater than or equal to) 268
- < (less than) 268
- <= (less than or equal to) 268
- + (plus)
 - insertion character 213, 214, 215
 - SIGN clause 223
 - symbol in PICTURE clause 203
- = (equal) 268

Numerics

- 0
 - insertion character 212
 - symbol in PICTURE clause 203
- 0 symbol in PICTURE clause 201
- 66, RENAMES data description entry 220
- 66, renames level-number 159
- 77, elementary item level-number 159
- 88, condition-name data description entry 182
- 88, conditional variable level number 159
- 9 symbol in PICTURE clause 201
- 9, symbol in PICTURE clause 203

A

- A symbol in PICTURE clause 200
- abbreviated combined relation condition 617
 - examples 286
 - using parentheses in 284
- abend, definition 617

- ACCEPT statement
 - description and format 304
 - FROM phrase 304
 - mnemonic-name in 304
 - overlapping operands, unpredictable results 295
 - system information transfer 305
- access mode
 - description 137
 - dynamic
 - DELETE statement 332
 - description 138
 - READ statement 412
 - DYNAMIC 137
 - random
 - DELETE statement 332
 - description 138
 - READ statement 411
 - RANDOM 137
 - sequential
 - DELETE statement 331
 - description 138
 - READ statement 409
 - SEQUENTIAL 137
- ACCESS MODE clause 137
- accessibility
 - assistive technologies xix
 - keyboard navigation xix
 - of COBOL for AIX xix
 - of this information xix
- ACOS function 500
- ADD statement
 - common phrases 291
 - CORRESPONDING phrase 310
 - description and format 308
 - END-ADD phrase 311
 - GIVING phrase 309
 - NOT ON SIZE ERROR phrase 310
 - ON SIZE ERROR phrase 310
 - ROUNDED phrase 310
- ADDRESS OF special register 17
- ADVANCING phrase 465
- AFTER phrase
 - INSPECT statement 363
 - PERFORM statement 399
 - with REPLACING 360
 - with TALLYING 359
 - WRITE statement 465
- alignment rules 164
- ALL literal
 - figurative constant 15
 - STOP statement 444
 - STRING statement 446
- ALL phrase
 - INSPECT statement 359, 360
 - SEARCH statement 423
 - UNSTRING statement 456
- ALL subscripting 494
- ALPHABET clause 113
- alphabet-name 12, 55
 - description 113
- alphabet-name (*continued*)
 - MERGE statement 376
 - PROGRAM COLLATING SEQUENCE clause 109
 - SORT statement 436
- alphabetic category 162
- alphabetic character in ACCEPT 304
- ALPHABETIC class test 264
- alphabetic function arguments 492
- alphabetic items
 - alignment rules 164
 - elementary move rules 382
 - how to define 204
 - PICTURE clause 204
- ALPHABETIC-LOWER class test 264
- ALPHABETIC-UPPER class test 264
- alphanumeric category 162
- alphanumeric comparisons 271
- alphanumeric function arguments 492
- alphanumeric functions 490, 491
- alphanumeric group items 159
- alphanumeric items
 - alignment rules 164
 - elementary move rules 383
 - how to define 206
 - PICTURE clause 206
- alphanumeric literals 29, 31
 - control character restrictions 30
 - in hexadecimal notation 30
- alphanumeric operands, comparing 271
- alphanumeric-edited category 162
- alphanumeric-edited items
 - alignment rules 164
 - elementary move rules 383
 - how to define 207
 - PICTURE clause 207
- ALSO phrase
 - ALPHABET clause 114
 - EVALUATE statement 342
- ALTER statement
 - description and format 312
 - GO TO statement and 350
 - segmentation considerations 312
- altered GO TO statement 350
- alternate key data item 140
- ALTERNATE RECORD KEY clause 140
 - DUPLICATES phrase 141
- AND logical operator 280
- ANNUITY function 500
- APPLY WRITE-ONLY clause 147
- Area A (cols. 8-11) 46
- Area B (cols. 12-72 in fixed source format, or cols 12-252 in extended source format) 48
- arguments 492
- arithmetic expression
 - COMPUTE statement 328
 - description 257
 - EVALUATE statement 342
 - relation condition 267
- arithmetic operators 13

- arithmetic operators (*continued*)
 - description 258
 - permissible symbol pairs 259
- arithmetic statements
 - ADD 308
 - common phrases 291
 - COMPUTE 328
 - DIVIDE 335
 - list of 294
 - multiple results 295
 - MULTIPLY 388
 - operands 294
 - programming notes 295
 - SUBTRACT 450
- ASCENDING KEY phrase
 - collating sequence 194
 - description 374
 - MERGE statement 374
 - OCCURS clause 193
 - SORT statement 435
- ASCII
 - collating sequence 588
 - specifying in SPECIAL-NAMES paragraph 114
- ASIN function 501
- ASSIGN clause
 - description 126
 - format 122
 - SELECT clause and 126
- assigning index values 427
- assignment-name 12
 - ASSIGN clause 126
 - RECORD DELIMITER clause 136
 - RERUN clause 145
- assistive technologies xix
- asterisk (*)
 - comment line 51
 - insertion character 215
- AT END phrase
 - READ statement 408
 - RETURN statement 416
 - SEARCH statement 426
 - SEARCH statement (binary search) 423
 - SEARCH statement (serial search) 421
- AT END-OF-PAGE phrases 466
- at-end condition
 - READ statement 410
 - RETURN statement 416
- ATAN function 501
- ATTRIBUTES phrase 474
- AUTHOR paragraph
 - description 103
 - format 95

B

- B
 - insertion character 212
 - symbol in PICTURE clause 199
- basic character set 3
- BASIS statement 537
- basis-name 54
- batch compile 82
- BEFORE phrase
 - INSPECT statement 363

- BEFORE phrase (*continued*)
 - PERFORM statement 399
 - with REPLACING 360
 - with TALLYING 359
 - WRITE statement 465
- Bibliography 645
- big-endian 5
- binary arithmetic operators 258
- binary data item, DISPLAY statement 333
- BINARY phrase in USAGE clause 231
- binary search 423
- blank lines 52
- BLANK WHEN ZERO clause
 - description and format 184
 - INDEX phrase in USAGE clause 235
- BLOCK CONTAINS clause
 - description 173
 - format 167
- branching
 - GO TO statement 349
 - out-of-line PERFORM statement 397
- BY CONTENT phrase
 - CALL statement 317
- BY REFERENCE phrase
 - CALL statement 317
- BY VALUE phrase
 - CALL statement 318
 - INVOKE statement 368

C

- C01-C012 environment names 465
- call convention 561
- CALL statement
 - CANCEL statement and 322
 - description and format 314
 - linkage section 254
 - ON OVERFLOW phrase 314
 - procedure division header 251, 254
 - program termination 314
 - subprogram linkage 314
 - transfer of control 74
 - USING phrase 254
- called and calling programs, description 314
- CALLINTERFACE directive 561
- CANCEL statement 322
- category
 - of group items 159
 - relationship to classes of data 160
 - relationship to usages of data 160
- category descriptions 162
- category of data 160
 - alphabetic 162, 204
 - alphanumeric 162, 206
 - alphanumeric-edited 162, 207
 - DBCS 163, 207
 - external floating-point 163
 - internal floating-point 163
 - national 163, 208
 - national-edited 163, 209
 - numeric 164, 205
 - numeric-edited 164, 206
- category of functions 161
- category of literals 162
- CBL (PROCESS) statement 538

- century window (See also date field)
 - definition 77
- CHAR compiler option 5
- CHAR function 501
- character code set, specifying 114
- character encoding unit 6
- character sets 5
- character-strings 9
 - COBOL words 9
 - representation in PICTURE clause 203
 - size determination 165
- CHARACTERS BY phrase 360
- CHARACTERS phrase
 - BLOCK CONTAINS clause 173
 - INSPECT statement 359
 - MEMORY SIZE clause 109
 - USAGE clause and 173
- characters, valid in COBOL program 3
- checkpoint processing, RERUN clause 145
- class (object-oriented) 87
- class (of data)
 - of data items 160
 - of figurative constants 161
 - of functions 161
 - of group items 159
 - of literals 161
- CLASS clause 116
- class condition 263
- class definition
 - class procedure division 249
 - CLASS-ID paragraph 100
 - configuration section 107
 - description 87
 - effect of SELF and SUPER 366
 - factory procedure division 249
 - identification division 96
 - object procedure division 249
 - requirements for indexed tables 195
- class identification division 95, 100
- class procedure division 249
- CLASS-ID paragraph 100
- class-name 12, 55
- class-name class test 264
- class-name, OO 55
- clauses 44
 - definition 44
 - syntactical hierarchy 43
- CLOSE statement
 - format and description 324
- COBOL
 - class definition 87
 - language structure 3
 - method definition 91
 - program structure 81
 - reference format 45
 - extended source format 45
 - fixed source format 45
- COBOL classes 87
- COBOL for AIX
 - accessibility xix
- COBOL objects 87
- COBOL words 9
 - with DBCS characters 9
 - with multi-byte characters 9
 - with single-byte characters 9

- code page names 5
- code pages 5
- CODE-SET clause
 - ALPHABET clause and 114
 - description 180
 - format 167
- collating sequence
 - ASCENDING/DESCENDING KEY phrase and 194
 - ASCII 588
 - EBCDIC 585
 - locale definition 609
 - specified in OBJECT-COMPUTER paragraph 109
 - specified in SPECIAL-NAMES paragraph 113
- COLLATING SEQUENCE phrase 109
 - ALPHABET clause 113
 - MERGE statement 376
 - SORT statement 436
- colon character
 - description 40
 - required use of 546
- column 7
 - indicator area 48
 - specifying comments 50
- combined condition
 - description 282
 - evaluation rules 283
 - logical operators and evaluation results 283
 - order of evaluation 283
 - permissible element sequences 282
- comma (,)
 - DECIMAL-POINT IS COMMA clause 118
 - insertion character 212
- comment lines
 - description 50
 - in identification division 103
 - in library text 542
- comments 37
- COMMON clause 99
- common processing facilities 296
- COMP-1 through COMP-5 data items 232
- comparison tables 268
- comparison types 268
- comparisons
 - alphanumeric operands 271
 - cycle, INSPECT statement 364
 - date field 276
 - DBCS operands 273
 - function pointer operands 278
 - group operands 275
 - in EVALUATE statement 343
 - index data items 275
 - index-names 275
 - national operands 273
 - numeric operands 274
 - object reference operands 278
 - procedure pointer operands 278
 - rules for COPY statement 544
- compatible date field (See also date field)
 - definition 77
- compiler directive 537
- compiler limits 577
- compiler options 538
 - CHAR 5
 - controlling listing output 538
 - DATEPROC 75
 - NUMPROC 279
 - PGMNAME 322
 - specifying 538
 - THREAD 195
 - TRUNC 166
- compiler-directing statements 51, 537
 - *CBL (*CONTROL) 538
 - *CONTROL (*CBL) 538
 - BASIS 537
 - CBL (PROCESS) 538
 - COPY 540
 - DELETE 547
 - EJECT 548
 - ENTER 548
 - INSERT 549
 - PROCESS (CBL) 538
 - READY TRACE 550
 - REPLACE 550
 - RESET TRACE 550
 - SERVICE LABEL 554
 - SERVICE RELOAD 554
 - SKIP1 554
 - SKIP2 554
 - SKIP3 554
 - TITLE 555
 - USE 556
- complex conditions
 - abbreviated combined relation 284
 - combined condition 282
 - description 280
 - negated simple 281
- complex OCCURS DEPENDING ON (CODO) 198
- composite of operands 295
- COMPUTATIONAL data items 231
- COMPUTATIONAL phrases in USAGE clause 232
- COMPUTE statement
 - common phrases 292
 - description and format 328
- computer-name 12, 108, 109
- concatenation of files 131
- condition
 - abbreviated combined relation 284
 - class 263
 - combined 282
 - complex 280
 - condition-name 265
 - EVALUATE statement 342
 - IF statement 351
 - negated simple 281
 - PERFORM UNTIL statement 399
 - relation 267
 - SEARCH statement (binary search) 423
 - SEARCH statement (serial search) 423
 - sign 279
 - simple 263
 - switch-status 280
- condition-name 12, 54, 64
 - and conditional variable 182
 - description and format 265
- condition-name (*continued*)
 - rules for values 242
 - SEARCH statement 425
 - SET statement 430
 - SPECIAL-NAMES paragraph 113
 - switch status condition 113
- conditional expressions
 - DBCS operands 273
 - description 262
 - index-names and index data items 275
 - order of evaluation of operands 283
 - parentheses in abbreviated combined relation conditions 284
- conditional statements
 - description 288
 - GO TO statement 349
 - IF statement 351
 - list of 288
 - PERFORM statement 399
- conditional variable 182
- configuration section
 - description (programs, classes, methods) 107
 - REPOSITORY paragraph 118
 - SOURCE-COMPUTER paragraph 108
 - SPECIAL-NAMES paragraph 110
- conformance rules
 - SET..USAGE OBJECT REFERENCE 433
- contained programs 81
- continuation
 - area 45
 - lines 48, 50
- CONTINUE statement 330
- CONTROL statement (*CONTROL) 538
- conversion of data, DISPLAY statement 333
- CONVERTING phrase 362
- COPY libraries 60
- COPY statement
 - comparison rules 544
 - description and format 540
 - example 545
 - replacement rules 544
 - REPLACING phrase 543
 - SUPPRESS option 543
- CORRESPONDING (CORR) phrase
 - ADD statement 310
 - description 310
 - MOVE statement 380
 - SUBTRACT statement 451
 - with ON SIZE ERROR phrase 294
- COS function 502
- COUNT IN phrase
 - UNSTRING statement 457
 - XML GENERATE statement 472
- CR (credit)
 - insertion character 213
 - symbol in PICTURE clause 201
- cs (currency symbol)
 - in PICTURE clause 199
- CURRENCY SIGN clause
 - description 117
 - Euro currency sign 117
 - currency sign value 117

- currency symbol 203
 - in PICTURE clause 201
 - specifying in CURRENCY SIGN clause 117
- currency symbol, default (\$) 213
- CURRENT-DATE function 502
- customer support 645

D

data

- alignment 164
- categories 160, 204
- classes 160
- hierarchies used in qualification 157
- organization 133
- signed 166
- truncation of 166, 190
- data category
 - alphabetic 204
 - alphanumeric 206
 - alphanumeric-edited 207
 - DBCS 207
 - national 208
 - national-edited 209
 - numeric 205
 - numeric-edited 206
- data category descriptions 162
- data conversion, DISPLAY statement 333
- data description entry 181
 - BLANK WHEN ZERO clause 184
 - data-name 183
 - DATE FORMAT clause 184
 - FILLER phrase 183
 - GLOBAL clause 189
 - indentation and 159
 - JUSTIFIED clause 190
 - level-66 format (previously defined items) 182
 - level-88 format (condition-names) 182
 - level-number description 182
 - OCCURS clause 192
 - OCCURS DEPENDING ON (ODO) clause 195
 - PICTURE clause 198
 - REDEFINES clause 216
 - RENAMES clause 220
 - SIGN clause 222
 - SYNCHRONIZED clause 223
 - USAGE clause 229
 - USAGE IS NATIONAL clause and 190
 - VALUE clause 239
- data division
 - data description entry 181
 - data relationships 156
 - file description (FD) entry 172
 - in factory definition 151
 - in method definition 151
 - in object definition 151
 - in program definition 151
 - levels of data 157
 - linkage section 155
 - local-storage section 154
 - sort description (SD) entry 172

- data division (*continued*)
 - working-storage section 153
- data division names 61
- data flow
 - STRING statement 448
 - UNSTRING statement 458
- data item
 - characteristics 181
 - description entry definition 153
 - EXTERNAL clause 189
- data item description entry 154
- data items
 - categories 161
 - classes 161
- data manipulation statements
 - ACCEPT 304
 - INITIALIZE 353
 - list of 296
 - MOVE 380
 - overlapping operands 296
 - READ 406
 - RELEASE 413
 - RETURN 415
 - REWRITE 417
 - SET 427
 - STRING 445
 - UNSTRING 454
 - WRITE 462
- data organization
 - access modes and 138
 - indexed 134
 - line-sequential 134
 - relative 134
 - sequential 134
- DATA RECORDS clause
 - description 178
 - format 167
- data relationships
 - data division 156
- data transfer 304
- data units
 - factory data 156
 - file data 155
 - instance data 156
 - method data 156
 - overview 155
 - program data 156
- data-item-description-entry linkage section 155
- data-name
 - data description entry 183
 - definition 54
- DATE 306
- date field
 - addition 260
 - arithmetic 260
 - compatible 77
 - DATE FORMAT clause 184
 - DATEPROC compiler option 75
 - DATEVAL function 505
 - definition 76
 - expansion of windowed date fields before use 185
 - group items that are date fields 187
 - in relation conditions 276
 - in sign conditions 279
 - MOVE statement, behavior in 385

- date field (*continued*)
 - nondate 77
 - purpose 75
 - restrictions 186
 - size errors 261, 293
 - storing arithmetic results 261
 - subtraction 261
 - UNDATE function 529
 - windowed date field conditional variables 266
- date field comparisons 276
- date format (See also DATE FORMAT clause)
 - definition 76
- DATE FORMAT clause 184
 - combining with other clauses 186
- date functions 496
- DATE YYYYMMDD 306
- DATE-COMPILED paragraph
 - description 103
 - format 95
- DATE-OF-INTEGGER function 504
- DATE-TO-YYYYMMDD function 504
- DATE-WRITTEN paragraph
 - description 103
 - format 95
- DATEPROC compiler option 75
- DATEVAL function 505
- DAY 306
- DAY YYYYDDD 307
- DAY-OF-INTEGGER function 506
- DAY-OF-WEEK 307
- DAY-TO-YYYYDDD function 507
- DB (debit)
 - insertion character 213
 - symbol in PICTURE clause 201
- DB2 file system
 - FILE STATUS clause 142
- DB2 files
 - size limits 581
- DBCS (Double-Byte Character Set)
 - elementary move rules 383
 - using in comments 103
- DBCS category 163
- DBCS character set 3
- DBCS characters
 - in COBOL words 10
- DBCS class condition 264
- DBCS comparisons 273
- DBCS function arguments 492
- DBCS items
 - alignment rules 164
 - how to define 207
 - in ACCEPT 304
 - PICTURE clause 207
- DBCS literals 32
 - in ACCEPT 304
 - SOSI compiler option 33
- de-editing 384
- DEBUG-CONTENTS 18
- DEBUG-ITEM special register 17, 594
- DEBUG-LINE 18
- DEBUG-NAME 18
- debugging 593
- DEBUGGING declarative 556, 558
- debugging lines 52, 108, 593

- debugging mode
 - compile-time switch 594
 - object-time switch 594
- DEBUGGING MODE clause 108, 558, 593, 594
- debugging sections 593
- decimal point (.) 292
- DECIMAL-POINT IS COMMA clause
 - description 118
 - NUMVAL function 520
 - NUMVAL-C function 521
- declarative procedures
 - description and format 255
 - PERFORM statement 396
 - USE statement 255
- declaratives
 - DEBUGGING 558
 - EXCEPTION/ERROR 556
 - LABEL 558
 - precedence rules for nested programs 557
- DECLARATIVES key word
 - begin in Area A 47
 - description 255
- declaratives section 255
- DELETE statement
 - description and format 547
 - dynamic access 332
 - format and description 331
 - INVALID KEY phrase 332
 - random access 332
 - sequential access 331
- DELIMITED BY phrase
 - STRING 445
 - UNSTRING statement 456
- delimited scope statement 289
- delimiter
 - INSPECT statement 361
 - UNSTRING statement 456
- DELIMITER IN phrase, UNSTRING statement 457
- DEPENDING phrase
 - GO TO statement 349
 - OCCURS clause 195
- derived class 87
- DESCENDING KEY phrase 193
 - collating sequence 194
 - description 374
 - MERGE statement 374
 - SORT statement 435
- display floating-point 210
- DISPLAY phrase in USAGE clause 233
- DISPLAY statement
 - description and format 333
- DISPLAY-OF function 507
- DIVIDE statement
 - common phrases 292
 - description and format 335
 - REMAINDER phrase 338
- division header
 - format, environment division 107
 - format, identification division 95
 - format, procedure division 251
 - specification of 46
- DO-UNTIL structure, PERFORM statement 399

- DO-WHILE structure, PERFORM statement 399
- Double-Byte Character Set (DBCS)
 - PICTURE clause and 207
 - using in comments 103
- DOWN BY phrase, SET statement 428
- DUPLICATES phrase 141
 - SORT statement 436
- dynamic access mode
 - data organization and 138
 - DELETE statement 332
 - description 138
 - READ statement 412

E

- E symbol in PICTURE clause 200
- EBCDIC
 - code page 1140 585
 - CODE-SET clause and 180
 - collating sequence 585
 - specifying in SPECIAL-NAMES paragraph 114
- EBCDIC_CODEPAGE environment variable 5
- editing
 - fixed insertion 213
 - floating insertion 214
 - replacement 215
 - signs 166
 - simple insertion 212
 - special insertion 212
 - suppression 215
- editing sign control symbol 201
- editing signs 166
- eject page 51
- EJECT statement 548
- elementary items 157
 - alignment rules 164
 - basic subdivisions of a record 157
 - MOVE statement 381
 - size determination in program 165
 - size determination in storage 165
- elementary move rules 381
- ELSE NEXT SENTENCE phrase 351
- ENCODING phrase
 - XML GENERATE statement 473
- encoding units 6
- end class marker 47
- END DECLARATIVES key word 255
- end markers 47
- end method marker 47
- END PROGRAM 82
- end program marker 47
- END-ADD phrase 311
- END-CALL phrase 321
- END-IF phrase 351
- END-INVOKE phrase 371
- end-of-file processing 324
- END-OF-PAGE phrases 466
- END-PERFORM phrase 398
- END-SUBSTRACT phrase 453
- END-WRITE phrase 467
- END-XML phrase
 - XML GENERATE statement 475
 - XML PARSE statement 483

- entries
 - definition 43
 - syntactical hierarchy 43
- ENTRY statement
 - description and format 340
 - subprogram linkage 340
- environment division
 - configuration section
 - ALPHABET clause 113
 - CURRENCY SIGN clause 117
 - OBJECT-COMPUTER paragraph 108
 - REPOSITORY paragraph 118
 - SOURCE-COMPUTER paragraph 108
 - SPECIAL-NAMES paragraph 110, 116
 - SYMBOLIC CHARACTERS clause 116
 - input-output section
 - FILE-CONTROL paragraph 122
 - REPOSITORY paragraph 118
 - environment names with WRITE ADVANCING 465
 - environment variables
 - in ACCEPT statement 304
 - in DISPLAY statement 333
 - environment-name 12, 305
 - SPECIAL-NAMES paragraph 112, 113
- EOP phrases 466
- equal sign (=) 267
- EQUAL TO relational operator 267
- EUC 6
- Euro currency sign 586
 - specifying in CURRENCY SIGN clause 117
- EVALUATE statement
 - comparing operands 343
 - determining truth value 342
 - format and description 341
- evaluation rules
 - combined conditions 283
 - EVALUATE statement 343
 - nested IF statement 352
- EXCEPTION/ERROR declarative 556
 - CLOSE statement 325
 - DELETE statement 331
 - description and format 556
- execution flow
 - ALTER statement 312
 - PERFORM statement 396
- EXIT METHOD statement
 - format and description 346
- EXIT PROGRAM statement
 - format and description 347
- EXIT statement
 - format and description 345
 - PERFORM statement 397
- expanded date field (See also date field)
 - definition 76
- expanded year (See also date field)
 - definition 76
- expansion of windowed date fields before use 185
- explicit attributes, of data 72
- explicit scope terminators 290

- exponentiation
 - exponential expression 258
- expression, arithmetic 257
- EXTEND phrase
 - OPEN statement 392
- extended character set 3
- extension language elements 565
- EXTERNAL clause
 - with data item 189
 - with file name 172
- external decimal item
 - DISPLAY statement 333
- external floating-point
 - DISPLAY statement 333
- external floating-point category 163
- external floating-point in ACCEPT 304
- external floating-point items
 - alignment rules 164
 - how to define 210
 - PICTURE clause 210
- external-class-name 12, 120

F

- FACTORIAL function 509
- factory data 87
- factory data division 151
 - format 152
- factory data unit 156
- factory definition
 - FACTORY paragraph 101
 - format and description 89
- factory identification division 95, 101
- factory method 87, 92
- FACTORY paragraph 101
- factory procedure division 249
- factory procedure division header 251
- factory working-storage 153
- FALSE phrase 342
- FD (file description) entry
 - BLOCK CONTAINS clause 173
 - DATA RECORDS clause 178
 - description 172
 - format 167
 - GLOBAL clause 173
 - LABEL RECORDS clause 177
 - level indicator 157
 - VALUE OF clause 177
- figurative constant
 - DISPLAY statement 333
 - STOP statement 444
 - STRING statement 446
- figurative constants 13
 - ALL literal 15
 - HIGH-VALUE 14
 - HIGH-VALUES 14
 - LOW-VALUE 14
 - LOW-VALUES 14
 - NULL 15
 - NULLS 15
 - QUOTE 15
 - QUOTES 15
 - SPACE 14
 - SPACES 14
 - symbolic-character 15
 - ZERO 13
 - ZEROES 13
- figurative constants (*continued*)
 - ZEROS 13
- file
 - definition 155
 - labels 177
- file concatenation 131
- file organization
 - and access modes 137
 - definition 137
 - LINAGE clause 178
 - line-sequential 134
 - types of 134
- file position indicator
 - description 302
 - READ statement 410
- file section 152, 172
 - EXTERNAL clause 172
 - RECORD clause 174
- FILE STATUS clause
 - DELETE statement and 331
 - description 141
 - FILE STATUS clause 142
 - file status key 297
 - format 122
 - INVALID KEY phrase and 301
- file status key
 - common processing facility 297
 - value and meaning 297
- FILE-CONTROL paragraph
 - ASSIGN clause 126
 - description and format 122
 - FILE STATUS clause 141
 - ORGANIZATION clause 133
 - PADDING CHARACTER clause 136
 - RECORD KEY clause 139
 - RELATIVE KEY clause 141
 - RESERVE clause 133
 - SELECT clause 126
- file-description-entry 152
- file-name 54
- file-name, specifying on SELECT
 - clause 126
- files
 - SdU
 - size limits 581
 - size limits 581
 - STL
 - size limits 581
- FILLER phrase 181
 - CORRESPONDING phrase 183
 - data description entry 183
- fixed insertion editing 213
- fixed segments 256
- fixed-length
 - records 173
- floating comment indicator (*>)
 - comment lines 50
 - description 51
 - inline comment 51
- floating insertion editing 214
- floating-point
 - DISPLAY statement 333
- floating-point literals 32
- FOOTING phrase of LINAGE
 - clause 178
- FOR REMOVAL phrase 324, 326
- format notation, rules for xiii

- FROM phrase
 - ACCEPT statement 304
 - REWRITE statement 417
 - SUBTRACT statement 450
 - with identifier 301
 - WRITE statement 464
- function arguments 492
- function definitions 496
- function pointer
 - in SET statement 427
- function pointer data items 234
 - relation condition 278
- function type 490
- function-identifier 71
- function-names 12
- function-pointer data items
 - SET statement 431
- FUNCTION-POINTER phrase in USAGE
 - clause 234
- functions
 - arguments 492
 - categories 161
 - class and category of 161
 - classes 161
 - description 489
 - rules for usage 491
 - types of functions 490

G

- G symbol in PICTURE clause 200
- garbage collection 87
- GIVING phrase
 - ADD statement 309
 - arithmetic 292
 - DIVIDE statement 338
 - MERGE statement 377
 - MULTIPLY statement 389
 - SORT statement 438
 - SUBTRACT statement 452
- GLOBAL clause
 - with data item 189
 - with file name 173
- glossary 617
- GO TO statement 312
 - altered 350
 - conditional 349
 - format and description 349
 - MORE-LABELS 350
 - SEARCH statement 425, 426
 - unconditional 349
- GO TO, DEPENDING ON phrase 312
- GOBACK statement 348
- graphic character 6
- GREATER THAN OR EQUAL TO symbol
 - (>=) 267
- GREATER THAN symbol (>) 267
- group comparisons 275
- group items 157
 - alphanumeric 159
 - class and category of 159
 - description 157
 - MOVE statement 386
 - national 160, 191
 - usage of 159
- group move rules 386
- GROUP-USAGE clause 191

- GROUP-USAGE clause *(continued)*
 - description 191
 - format 191
- GROUP-USAGE NATIONAL clause 191
- groups
 - categories 161
 - classes 161

H

- halting execution 444
- hexadecimal notation
 - for alphanumeric literals 30
 - for national literals 35
- hiding 103
- hierarchy of data 157
- HIGH-VALUE figurative constant 14, 114
- HIGH-VALUES figurative constant 14, 114
- hyphen (-), in indicator area 48

I

- I-O-CONTROL paragraph
 - APPLY WRITE-ONLY clause 147
 - checkpoint processing in 145
 - description 121, 143
 - MULTIPLE FILE TAPE clause 147
 - order of entries 143
 - RERUN clause 145
 - SAME AREA clause 146
 - SAME RECORD AREA clause 146
 - SAME SORT AREA clause 146
 - SAME SORT-MERGE AREA clause 147
- IBM extensions xvi, 565
- identification division
 - CLASS-ID paragraph 100
 - FACTORY paragraph 101
 - format 95
 - format (program, class, method) 95
 - METHOD-ID paragraph 102
 - OBJECT paragraph 102
 - optional paragraphs 103
 - PROGRAM-ID paragraph 98
- identifier 257
- identifiers 62, 257
- IF statement 351
- imperative statement 286
- implementor-name 12
- implicit
 - redefinition of storage area 172, 218
 - scope terminators 290
- implicit attributes, of data 72
- in-line PERFORM statement 396
- indentation 48, 159
- independent segments 256
- index
 - data item 275, 381
 - relative indexing 68
 - SET statement 68
- index data item 65
- INDEX phrase in USAGE clause 234
- index-name 55, 65
 - assigning values 427

- index-name *(continued)*
 - comparisons 275
 - OCCURS clause 195
 - PERFORM statement 405
 - SET statement 427
- INDEXED BY phrase 195
- indexed files
 - CLOSE statement 326
 - DELETE statement 332
 - FILE-CONTROL paragraph
 - format 122
 - I-O-CONTROL paragraph
 - format 143
 - organization 134
 - permissible statements for 394
 - READ statement 410
 - REWRITE statement 418
 - START statement 442
- indexed organization
 - description 134
 - FILE-CONTROL paragraph
 - format 122
 - I-O-CONTROL paragraph
 - format 143
- indexing
 - description 67
 - MOVE statement evaluation 381
 - OCCURS clause 67, 192
 - relative 68
 - SET statement and 68
- indicator area 45
- inheritance 87, 101
- INHERITS clause 100
- INITIAL clause 99
- initial state of program 99
- INITIALIZE statement
 - format and description 353
 - overlapping operands, unpredictable results 295
- INPUT phrase
 - OPEN statement 392
 - USE statement 556
- INPUT PROCEDURE phrase
 - RELEASE statement 413
 - SORT statement 437
- Input-Output section
 - description 121
 - file control paragraph 121
 - FILE-CONTROL keyword 121
 - FILE-CONTROL paragraph 122
 - format 121
 - I-O-CONTROL paragraph 143
- input-output statements
 - ACCEPT 304
 - CLOSE 324
 - common processing facilities 296
 - DELETE 331
 - DISPLAY 333
 - EXCEPTION/ERROR
 - procedures 557
 - general description 296
 - OPEN 391
 - READ 406
 - REWRITE 417
 - START 441
 - WRITE 462
- INSERT statement 549

- insertion editing
 - fixed (numeric-edited items) 213
 - floating (numeric-edited items) 214
 - simple 212
 - special (numeric-edited items) 212
- INSPECT statement
 - AFTER phrase 361
 - BEFORE phrase 361
 - comparison cycle 364
 - CONVERTING phrase 362
 - overlapping operands, unpredictable results 295
 - REPLACING phrase 359
- INSTALLATION paragraph
 - description 103
 - format 95
- instance data 87, 91, 156
- instance definition
 - format and description 89
- instance method 87, 92
- instance variable 87
- integer arguments 492
- INTEGER function 509
- integer function arguments 493
- integer functions 491
- INTEGER-OF-DATE function 510
- INTEGER-OF-DAY function 510
- INTEGER-PART function 511
- internal floating-point
 - DISPLAY statement 333
 - size of items 165
- internal floating-point category 163
- internal floating-point items
 - alignment rules 164
 - how to define 204
- INTO phrase
 - DIVIDE statement 335
 - READ statement 406
 - RETURN statement 415
 - STRING statement 445
 - UNSTRING statement 457
 - with identifier 301
- intrinsic functions 489
 - ACOS 500
 - alphanumeric functions 490
 - ANNUITY 500
 - ASIN 501
 - ATAN 501
 - categories 161
 - CHAR 501
 - classes 161
 - COS 502
 - CURRENT-DATE 502
 - DATE-OF-INTEGER 504
 - DATE-TO-YYYYMMDD 504
 - DATEVAL 505
 - DAY-OF-INTEGER 506
 - DAY-TO-YYYYDDD 507
 - DISPLAY-OF 507
 - FACTORIAL 509
 - floating-point literals 493
 - INTEGER 509
 - integer functions 490
 - INTEGER-OF-DATE 510
 - INTEGER-OF-DAY 510
 - INTEGER-PART 511
 - LENGTH 511

intrinsic functions (*continued*)

- LOG 513
- LOG10 513
- LOWER-CASE 513
- MAX 514
- MEAN 515
- MEDIAN 516
- MIDRANGE 516
- MIN 517
- MOD 518
- national functions 490
- NATIONAL-OF 518
- numeric functions 490
- NUMVAL 519
- NUMVAL-C 520
- ORD 522
- ORD-MAX 523
- ORD-MIN 523
- PRESENT-VALUE 524
- RANDOM 524
- RANGE 525
- REM 526
- REVERSE 526
- SIN 527
- SQRT 527
- STANDARD-DEVIATION 528
- SUM 528
- summary of 497
- TAN 529
- UNDATE 529
- UPPER-CASE 530
- VARIANCE 530
- WHEN-COMPILED 531
- YEAR-TO-YYYY 532
- YEARWINDOW 533

invalid key condition 300

INVALID KEY phrase

- DELETE statement 332
- READ statement 408
- REWRITE statement 418
- START statement 442
- WRITE statement 466

INVOKE statement

- BY VALUE phrase 368
- format and description 366
- LENGTH OF special register 368
- NEW phrase 367
- NOT ON EXCEPTION phrase 370
- ON EXCEPTION phrase 370
- RETURNING phrase 369
- SELF special object identifier 367
- SUPER special object identifier 367
- USING phrase 368

J

Java

- class-name 120
- package 120

Java classes 87

Java interoperability 87

- data types 368, 371, 373
- literal types 368

Java interoperation 87

Java Native Interface (JNI) 19, 87, 631

Java objects 87

Java String data 87

java.lang.Object 87

JNI environment pointer 87

JNIENVPTR special register 19, 87

JUSTIFIED clause

- description and format 190
- effect on initial settings 190
- STRING statement 446
- truncation of data 190
- USAGE IS INDEX clause and 190
- VALUE clause and 239

K

Kanji 264

key of reference 134

KEY phrase

- OCCURS clause 193
- READ statement 408
- SEARCH statement 423
- SORT statement 435
- START statement 441

keyboard navigation xix

L

LABEL declarative 556, 558

LABEL RECORDS clause

- description 177
- format 167

language-name 12

LEADING phrase

- INSPECT statement 359, 360
- SIGN clause 223

LENGTH function 511

LENGTH OF special register 19

- INVOKE statement 368

LESS THAN OR EQUAL TO symbol

- (<=) 267

LESS THAN symbol (<) 267

level

- 01 item 157
- 02-49 item 157

level indicator

- (FD and SD) 47
- definition 156

level-number 50, 157, 181

- (01 and 77) 47
- 66, renames 159
- 77, elementary item 159
- 88, conditional variable 159
- definition 156
- description and format 182
- FILLER phrase 183

levels of data 156, 157

library-name 12, 54

- COPY statement 541

limits of the compiler 577

- file input-output 581

LINAGE clause 466

- description 178
- diagram of phrases 178
- format 167

LINAGE-COUNTER special register

- description 20
- WRITE statement 465

LINE

- WRITE statement 465

line advancing 465

line-sequential file organization 134

line-sequential files

- size limits 581

LINES

- WRITE statement 465

LINES AT BOTTOM phrase 178

LINES AT TOP phrase 178

linkage section

- called subprogram 254
- description 155
- requirement for indexed items 195
- VALUE clause 239

List of resources 645

literals

- and arithmetic expressions 257
- ASSIGN clause 126
- categories 162
- classes 162
- CODE-SET clause and ALPHABET clause 114
- CURRENCY SIGN clause 117
- DBCS 32
- description 28
- null-terminated alphanumeric 31
- STOP statement 444
- VALUE clause 240
- Z literals 31

literals, class and category of 161

local-storage 154

- defining with RECURSIVE clause 99
- requirement for indexed items 195

locale 609

LOG function 513

LOG10 function 513

logical operator

- complex condition 280
- in evaluation of combined conditions 283
- list of 280

logical record

- definition 155
- file data 155
- program data 156
- record description entry and 156
- RECORDS phrase 174

LOW-VALUE figurative constant 14, 114

LOW-VALUES figurative constant 14, 114

LOWER-CASE function 513

lowercase letters

- in PICTURE clause 199

M

MAX function 514

maximum file size 581

maximum index value 68

maximum record size 581

MEAN function 515

MEDIAN function 516

MEMORY SIZE clause 109

MERGE statement

- ASCENDING/DESCENDING KEY phrase 374

MERGE statement (*continued*)
 COLLATING SEQUENCE
 phrase 376
 format and description 374
 GIVING phrase 377
 OUTPUT PROCEDURE phrase 378
 segmentation considerations 379
 USING phrase 377
 method data 156
 method data division 151
 format 151
 method definition
 effect of SELF and SUPER 366
 format and description 91
 identification division 98
 inheritance rules 101
 method procedure division 249
 METHOD-ID paragraph 102
 method file section 153
 method hiding 103
 method identification division 95, 102
 method local-storage 154, 155
 method overloading 102
 method overriding 102
 method procedure division 249, 250
 method procedure division header 252
 method working-storage 153
 METHOD-ID paragraph 102
 method-name 54
 methods
 available to subclasses 101
 exiting 346
 invoking 366
 recursively reentering 99
 reusing 100
 MIDRANGE function 516
 millennium language extensions
 syntax 75
 millennium language extensions (MLE)
 (See also date field)
 description 75
 MIN function 517
 minus sign (-)
 COBOL character 3
 fixed insertion symbol 213
 floating insertion symbol 214, 215
 SIGN clause 223
 mnemonic-name 12, 55, 305
 ACCEPT statement 304
 DISPLAY statement 333
 SET statement 429
 SPECIAL-NAMES paragraph 113
 WRITE statement 465
 MOD function 518
 MORE-LABELS GO TO statement 350
 MOVE statement
 CORRESPONDING phrase 380
 elementary moves 381
 format and description 380
 group moves 386
 record area 386
 multibyte characters
 in COBOL words 10
 MULTIPLE FILE TAPE clause 147
 multiple record processing, READ
 statement 408

multiple results, arithmetic
 statements 295
 MULTIPLY statement
 common phrases 292
 format and description 388

N

N symbol in PICTURE clause 200
 NAMESPACE phrase 474
 NAMESPACE-PREFIX phrase 474
 national category 163
 national comparisons 273
 national data items 235
 elementary move rules 383
 in a class condition 263
 in ACCEPT 304
 in UNSTRING statement 454
 SEARCH statement 423
 national floating-point 210
 national function arguments 492
 national functions 490, 491
 national groups 191
 CORRESPONDING phrase 191
 description 191
 INITIALIZE statement 191
 qualification of data-names 191
 RENAMES clause 191
 where processed as group 191
 XML GENERATE statement 191
 national items
 alignment rules 164
 how to define 208
 PICTURE clause 208
 national literals 3, 34
 in ACCEPT 304
 national literals in hexadecimal
 notation 35
 NATIONAL phrase in USAGE
 clause 235
 national-edited category 163
 national-edited items 209
 alignment rules 164
 how to define 209
 NATIONAL-OF function 518
 native binary data item 232
 native character set 114
 native collating sequence 114
 negated combined condition 282
 negated simple condition 281
 NEGATIVE in sign condition 279
 nested IF structure
 description 352
 EVALUATE statement 341
 nested programs
 description 81
 precedence rules for 557
 NEW phrase
 INVOKE statement 367
 next executable statement 73
 NEXT RECORD phrase, READ
 statement 406
 NEXT SENTENCE phrase
 IF statement 351
 SEARCH statement 426
 SEARCH statement (binary
 search) 425

NEXT SENTENCE phrase (*continued*)
 SEARCH statement (serial
 search) 421
 NO ADVANCING phrase, DISPLAY
 statement 334
 NO REWIND phrase 324
 OPEN statement 392
 nondate (See also date field)
 definition 77
 nonreel file, definition 326
 NOT AT END phrase
 READ statement 408
 RETURN statement 416
 NOT END-OF-PAGE phrase 466
 NOT INVALID KEY phrase
 DELETE statement 332
 READ statement 408
 REWRITE statement 418
 START statement 442
 NOT ON EXCEPTION phrase
 CALL statement 320
 INVOKE statement 370
 XML GENERATE statement 475
 XML PARSE statement 483
 NOT ON OVERFLOW phrase
 STRING statement 447
 UNSTRING statement 458
 NOT ON SIZE ERROR phrase
 ADD statement 310
 DIVIDE statement 338
 general description 293
 MULTIPLY statement 390
 SUBTRACT statement 452
 NSYMBOL compiler option 3
 NULL
 figurative constant 15
 null block branch, CONTINUE
 statement 330
 null-terminated alphanumeric literals 31
 NULL/NULLS
 data pointer 277, 430
 figurative constant 244
 function-pointer 278, 432
 object reference 278, 433
 procedure-pointer 278, 432
 NULLS
 figurative constant 15
 numeric arguments 492, 493
 numeric category 164
 NUMERIC class test 264
 numeric comparisons 274
 numeric function arguments 493
 numeric functions 490, 491
 numeric items 205
 alignment rules 164
 how to define 205
 millennium dates 205
 PICTURE clause 205
 numeric literals 31
 numeric-edited category 164
 numeric-edited item
 editing signs 166
 elementary move rules 384
 numeric-edited items
 alignment rules 164
 how to define 206
 PICTURE clause 206

NUMVAL function 519
NUMVAL-C function 520

O

object data division 151
 format 152
object definition
 OBJECT paragraph 102
object identification division 95, 102
object instance data 156
OBJECT paragraph 102
object procedure division 249
object program 81
object reference 87
 in SET statement 427
OBJECT REFERENCE phrase 235
object working-storage 153
OBJECT-COMPUTER paragraph 108
object-oriented class-name 55
object-oriented COBOL
 class definition 87
 comparison rules 278
 conformance rules
 SET...USAGE OBJECT
 REFERENCE 433
 effect of VALUE clause 153
 factory definition 89
 identification division (class and
 method) 95
 INHERITS clause 100
 INVOKE statement 366
 method definition 91
 method-name 54
 object definition 89
 OBJECT REFERENCE phrase in
 USAGE clause 235
 OO class name 55
 procedure division (classes and
 methods) 249
 REPOSITORY paragraph 118
 SELF and SUPER special object
 identifiers 13
 specifying configuration section 107
 subclasses and methods 101
objects in EVALUATE statement 342
obsolete language elements xvi
OCCURS clause
 ASCENDING/DESCENDING KEY
 phrase 193
 description 192
 INDEXED BY phrase 195
 restrictions 192
 variable-length tables format 195
OCCURS DEPENDING ON (ODO)
 clause
 complex 198
 description 196
 format 195
 object of 196
 RECORD clause 174
 REDEFINES clause and 192
 SEARCH statement and 192
 subject and object of 196
 subject of 192, 196
 subscripting 65
OFF phrase, SET statement 429
OMITTED phrase 317, 318
ON EXCEPTION phrase
 CALL statement 320
 INVOKE statement 370
 XML GENERATE statement 475
 XML PARSE statement 482
ON OVERFLOW phrase
 CALL statement 320
 STRING statement 447, 458
ON phrase, SET statement 429
ON SIZE ERROR phrase
 ADD statement 310
 arithmetic statements 293
 COMPUTE statement 329
 DIVIDE statement 338
 MULTIPLY statement 390
 SUBTRACT statement 452
OPEN statement
 format and description 391
 I-O phrase 392
 phrases 391
 programming notes 393
 system dependencies 394
operands
 comparison of alphanumeric 271
 comparison of DBCS 273
 comparison of group 275
 comparison of national 273
 comparison of numeric 274
 composite of 295
 overlapping 295, 296
operation of XML GENERATE
 statement 476
operational sign
 algebraic, description of 166
 SIGN clause and 166
 USAGE clause and 166
operational signs 166
optional words, syntax notation xiii
ORD function 522
ORD-MAX function 523
ORD-MIN function 523
order of entries
 clauses in FILE-CONTROL
 paragraph 122
 I-O-CONTROL paragraph 143
order of evaluation in combined
 conditions 283
ORGANIZATION clause
 description 133
 format 122
 INDEXED phrase 133
 LINE SEQUENTIAL phrase 133
 RELATIVE phrase 133
 SEQUENTIAL phrase 133
out-of-line PERFORM statement 397
outermost programs, debugging 558
OUTPUT phrase 392
OUTPUT PROCEDURE phrase
 MERGE statement 378
 RETURN statement 415
 SORT statement 439
OVERFLOW phrase
 CALL statement 320
 STRING statement 447, 458
overlapping operands invalid in
 arithmetic statements 295

overlapping operands invalid in
 (continued)
 data manipulation statements 296
overloading 102
overriding 102

P

P symbol in PICTURE clause 200, 202
PACKED-DECIMAL phrase in USAGE
 clause 232
PADDING CHARACTER clause 136
PAGE
 WRITE statement 465
page eject 51
paragraph
 header, specification of 47
 termination, EXIT statement 345
paragraph-name 11, 54
 description 257
 specification of 47
paragraphs
 description 43, 257
 syntactical hierarchy 43
parent class 87
parentheses
 combined conditions, use 282
 in arithmetic expressions 258
partial listings 538
PASSWORD clause
 description 141
PERFORM statement
 branching 397
 conditional 399
 END-PERFORM phrase 398
 EVALUATE statement 341
 execution sequences 397
 EXIT statement 345
 format and description 396
 in-line 397
 out-of-line 397
 TIMES phrase 398
 VARYING phrase 400, 402
period (.)
 actual decimal point 212
PGMNAME compiler option
 CANCEL statement 322
phrases
 definition 44
 syntactical hierarchy 43
physical record
 BLOCK CONTAINS clause 173
 definition 155
 file data 155
 file description entry and 156
 RECORDS phrase 174
PICTURE character-strings 36
PICTURE clause
 and class condition 263
 computational items and 231
 CURRENCY SIGN clause 117
 data categories 204
 DECIMAL-POINT IS COMMA
 clause 118, 199
 description 198
 editing 211
 format 198

PICTURE clause (*continued*)
 symbols used in 199
 PICTURE SYMBOL phrase 117
 picture symbols 199
 - 201
 , 201
 / 201
 . 201
 \$ (currency symbol) 201
 * 201
 + 201
 0 201
 9 201
 A 200
 asterisk 201
 B 200
 comma 201
 CR 201
 currency symbol (cs) 201, 203
 DB 201
 E 200
 G 200
 minus 201
 N 200
 P 200, 202
 period 201
 plus 201
 S 200
 sequence of 201
 slash 201
 V 200
 X 201
 Z 201
 plus (+)
 fixed insertion symbol 213
 floating insertion symbol 214, 215
 insertion character 215
 SIGN clause 223
 pointer data items
 relation condition 277
 SET statement 430
 USAGE clause 236
 POINTER phrase
 STRING statement 446
 UNSTRING statement 457
 POINTER phrase in USAGE clause 236
 POSITIVE in sign condition 279
 PRESENT-VALUE function 524
 PREVIOUS RECORD phrase, READ
 statement 407
 priority-number 110, 256
 procedure branching
 GO TO statement 349
 statements, executed sequentially 303
 procedure branching statements 303
 procedure division
 declarative procedures 255
 description 249
 format (programs, methods,
 classes) 249
 header 251
 statements 303
 procedure division header
 RETURNING phrase 254
 USING phrase 252
 procedure division names 61

procedure pointer data items
 relation condition 278
 procedure-name
 GO TO statement 349
 MERGE statement 378
 PERFORM statement 396
 SORT statement 437
 procedure-pointer data items
 SET statement 431
 USAGE clause 237
 PROCEDURE-POINTER phrase in
 USAGE clause 237
 procedures, description 256
 PROCESS (CBL) statement 538
 PROCESSING PROCEDURE phrase, in
 XML PARSE 481
 product support 645
 PROGRAM COLLATING SEQUENCE
 clause
 ALPHABET clause 113
 SPECIAL-NAMES paragraph
 and 109
 program data 156
 program data division 151
 format 151
 program definition
 program procedure division 249
 program identification division 95
 program local-storage 154
 program procedure division 249
 program procedure division header 251
 program termination
 GOBACK statement 348
 STOP statement 444
 program working-storage 153
 PROGRAM-ID paragraph
 description 98
 format 95
 program-name 54
 program-name, rules for referencing 84
 program, separately compiled 81
 Programming interface information 614
 programming notes
 ACCEPT statement 304
 altered GO TO statement 312
 arithmetic statements 295
 data manipulation statements 445,
 454
 DELETE statement 331
 EXCEPTION/ERROR
 procedures 557
 OPEN statement 393
 PERFORM statement 398
 RECORD clause 174
 STRING statement 445
 UNSTRING statement 454
 programming structures 399
 programs, recursive 99
 pseudo-text
 continuation rules 552
 COPY statement operand 543
 description 52
 pseudo-text delimiters 41

Q

qualification 59
 quotation mark character 49
 QUOTE figurative constant 15
 QUOTES figurative constant 15

R

railroad track format, how to read xiii
 random access mode
 data organization and 138
 DELETE statement 332
 description 138
 READ statement 411
 RANDOM function 524
 RANGE function 525
 READ statement
 AT END phrases 408
 dynamic access mode 412
 format and description 406
 INTO identifier phrase 301, 407
 INVALID KEY phrase 300, 408
 KEY phrase 408
 multiple record processing 408
 NEXT RECORD phrase 406
 overlapping operands, unpredictable
 results 295
 programming notes 412
 random access mode 411
 READY TRACE statement 550
 receiving field
 COMPUTE statement 328
 MOVE statement 380
 multiple results rules 295
 SET statement 427
 STRING statement 445
 UNSTRING statement 457
 record
 area description 174
 elementary items 157
 fixed-length 173
 logical, definition of 155
 physical, definition of 155
 size limits 581
 record area
 MOVE statement 386
 RECORD clause
 description and format 174
 omission of 174
 RECORD CONTAINS 0
 CHARACTERS 175
 RECORD DELIMITER clause 136
 record description entry 154, 181
 levels of data 157
 logical record 156
 RECORD KEY clause
 description 139
 format 122
 record key in indexed file 332
 record-description entry
 linkage section 155
 record-description-entry 153
 record-name 54
 RECORDING MODE clause 180
 RECORDS phrase
 BLOCK CONTAINS clause 173

- RECORDS phrase (*continued*)
 - RERUN clause 145
 - RECURSIVE clause 99
 - recursive methods 366
 - recursive programs 99
 - requirement for indexed items 195
 - REDEFINES clause
 - description 216
 - examples of 218
 - format 216
 - general considerations 218
 - OCCURS clause restriction 216
 - undefined results 219
 - VALUE clause and 216
 - redefinition, implicit 172
 - REEL phrase 324, 326
 - reference format 45
 - extended source format 45
 - fixed source format 45
 - reference-modification
 - description 68
 - MOVE statement evaluation 381
 - reference, methods of
 - simple data 61
 - relation character
 - COPY statement 543
 - INITIALIZE statement 353
 - INSPECT statement 359
 - relation conditions
 - abbreviated combined 284
 - alphanumeric comparisons 268, 271
 - comparison operations 268
 - data pointer 277
 - DBCS comparisons 268, 273
 - description 266
 - function-pointer operands 278
 - general relation 267
 - group comparisons 268
 - national comparisons 268
 - numeric comparisons 268
 - object reference 278
 - procedure pointer operands 278
 - relational operator
 - in abbreviated combined relation
 - condition 284
 - meaning of each 268
 - relation condition use 267
 - relational operators 13
 - relative files
 - access modes allowed 138
 - CLOSE statement 326
 - DELETE statement 332
 - FILE-CONTROL paragraph
 - format 122
 - I-O-CONTROL paragraph
 - format 143
 - organization 134
 - permissible statements for 394
 - READ statement 409
 - RELATIVE KEY clause 138, 141
 - REWRITE statement 419
 - START statement 443
 - RELATIVE KEY clause
 - description 141
 - format 122
 - relative organization
 - access modes allowed 138
 - relative organization (*continued*)
 - description 134
 - FILE-CONTROL paragraph
 - format 122
 - I-O-CONTROL paragraph
 - format 143
 - RELEASE statement 295, 413
 - REM function 526
 - REMAINDER phrase of DIVIDE
 - statement 338
 - RENAMES clause 159
 - description and format 220
 - INITIALIZE statement 354
 - level 66 item 159, 220
 - PICTURE clause 199
 - repeated words, syntax notation xiv
 - REPLACE statement
 - comparison operation 552
 - continuation rules for
 - pseudo-text 552
 - description and format 550
 - special notes 553
 - replacement editing 215
 - replacement rules for COPY
 - statement 544
 - REPLACING phrase
 - COPY statement 543
 - INITIALIZE statement 353, 354
 - REPOSITORY paragraph 118, 120
 - required words, syntax notation xiii
 - RERUN clause
 - checkpoint processing 145
 - description 145
 - format 143
 - RECORDS phrase 145
 - sort/merge 145
 - RESERVE clause
 - description 133
 - format 122
 - reserved words 12, 595
 - RESET TRACE statement 550
 - resolution of names 56
 - result field
 - GIVING phrase 292
 - NOT ON SIZE ERROR phrase 293
 - ON SIZE ERROR phrase 293
 - ROUNDED phrase 292
 - RETURN statement
 - AT END phrase 416
 - description and format 415
 - overlapping operands, unpredictable
 - results 295
 - RETURN-CODE special register 21
 - RETURNING phrase
 - CALL statement 319
 - INVOKE statement 369
 - procedure division header 254
 - reusing logical records 418
 - REVERSE function 526
 - REWRITE statement
 - description and format 417
 - FROM identifier phrase 301
 - INVALID KEY phrase 418
 - ROUNDED phrase
 - ADD statement 310
 - COMPUTE statement 329
 - description 292
 - ROUNDED phrase (*continued*)
 - DIVIDE statement 338
 - MULTIPLY statement 390
 - size error checking and 294
 - SUBTRACT statement 452
 - RSD file 174, 326
 - WRITE statement 465
 - RSD files
 - size limits 581
 - rules for condition-name entries 242
 - rules for syntax notation xiii
 - run unit
 - description 81
 - termination with CANCEL
 - statement 323
 - runtime options
 - DEBUG 594
 - NODEBUG 594
- ## S
- S symbol in PICTURE clause 200
 - S01-S05 environment names 465
 - SAME clause 146
 - SAME RECORD AREA clause
 - description 146
 - format 143
 - SAME SORT AREA clause
 - description 146
 - format 143
 - SAME SORT-MERGE AREA clause
 - description 147
 - format 143
 - scope of names 53
 - scope terminator
 - explicit 290
 - implicit 290
 - scu (source conversion utility) 583
 - SD (sort file description) entry
 - data division 172
 - DATA RECORDS clause 178
 - description 172
 - level indicator 157
 - SD (Sort File Description) entry
 - description 167
 - SdU files
 - size limits 581
 - SEARCH statement
 - AT END phrase 421, 426
 - binary search 423
 - description and format 420
 - serial search 421
 - SET statement 421
 - VARYING phrase 422
 - WHEN phrase 425
 - SEARCH STATEMENT
 - NEXT SENTENCE phrase 426
 - section header
 - description 256
 - specification of 46
 - section-name 11, 54
 - description 256
 - in EXCEPTION/ERROR
 - declarative 556
 - sections 43, 256
 - SECURITY paragraph
 - description 103

SECURITY paragraph (*continued*)
 format 95
 SEGMENT-LIMIT clause 110
 segmentation 256
 segmentation considerations 312, 379, 440
 SELECT clause
 ASSIGN clause and 126
 format 122
 specifying a file name 126
 SELECT OPTIONAL clause
 CLOSE statement 326
 description 126
 format 122
 specification for sequential I-O files 126
 selection objects in EVALUATE statement 342
 selection subjects in EVALUATE statement 342
 SELF 278
 SELF special object identifier 13, 367
 sending field
 MOVE statement 380
 SET statement 427
 STRING statement 445
 UNSTRING statement 454
 sentences
 definition 44
 description 257
 syntactical hierarchy 43
 SEPARATE CHARACTER phrase of
 SIGN clause 223
 separately compiled program 81
 separators 39, 242
 separators, rules for 39
 sequence number area (cols. 1-6) 45
 sequential access mode
 data organization and 138
 DELETE statement 331
 description 138
 READ statement 409
 REWRITE statement 418
 sequential files
 access mode allowed 138
 CLOSE statement 324, 326
 description 134
 file description entry 167
 FILE-CONTROL paragraph
 format 122
 LINAGE clause 178
 OPEN statement 391
 PASSWORD clause valid with 141
 permissible statements for 394
 READ statement 409
 REWRITE statement 418
 SELECT OPTIONAL clause 126
 serial search 421
 PERFORM statement 400
 SERVICE LABEL statement 554
 SERVICE RELOAD statement 554
 SET statement
 description and format 427
 DOWN BY phrase 428
 function-pointer data items 234, 431
 index data item 234
 object reference data items 433

SET statement (*continued*)
 OFF phrase 429
 ON phrase 429
 overlapping operands, unpredictable results 295
 pointer data items 430
 procedure-pointer data items 431
 requirement for indexed items 195
 SEARCH statement 428
 TO phrase 427
 TO TRUE phrase 430
 UP BY phrase 428
 SFS and SdU file system
 FILE STATUS clause 142
 sharing data 189
 sharing files 173
 SHIFT-IN special register 21
 SHIFT-OUT special register 21
 sibling program 81
 SIGN clause 222
 sign condition 279
 SIGN IS SEPARATE clause 223
 signed
 numeric item, definition 205
 operational signs 166
 simple condition
 combined 282
 description and types 263
 negated 281
 simple data reference 61
 simple insertion editing 212
 SIN function 527
 single-byte ASCII 6
 single-byte EBCDIC 6
 size-error condition 293
 skip to next page 51
 SKIP1 statement 554
 SKIP2 statement 554
 SKIP3 statement 554
 slack bytes
 between 228
 within 226
 slash (/)
 comment lines 50
 insertion character 212
 symbol in PICTURE clause 201
 SORT statement
 ASCENDING KEY phrase 435
 COLLATING SEQUENCE
 phrase 436
 DESCENDING KEY phrase 435
 description and format 434
 DUPLICATES phrase 436
 GIVING phrase 438
 INPUT PROCEDURE phrase 437
 OUTPUT PROCEDURE phrase 439
 segmentation considerations 440
 USING phrase 437
 SORT-CONTROL special register 22, 440
 SORT-CORE-SIZE special register 22, 439
 SORT-FILE-SIZE special register 23, 439
 SORT-MESSAGE special register 23, 439
 SORT-MODE-SIZE special register 23, 440
 SORT-RETURN special register 24, 440

Sort/Merge feature
 I-O-CONTROL paragraph
 format 143
 MERGE statement 374
 RELEASE statement 413
 RERUN clause 145
 RETURN statement 415
 SAME SORT AREA clause 146
 SAME SORT-MERGE AREA
 clause 147
 SORT statement 434
 Sort/Merge file statement phrases
 ASCENDING/DESCENDING KEY
 phrase 374
 COLLATING SEQUENCE
 phrase 376
 GIVING phrase 377
 OUTPUT PROCEDURE phrase 378
 USING phrase 377
 SOSI compiler option 33
 source code
 library, programming notes 545
 listing 540
 source code format 45
 source conversion utility (scu) 583
 source language debugging 593
 source program
 standard COBOL reference format 45
 SOURCE-COMPUTER paragraph 108
 SPACE figurative constant 14
 SPACES figurative constant 14
 special insertion editing 212
 special object identifiers
 SELF 13
 SUPER 13
 special registers 16
 ADDRESS OF 17
 DEBUG-ITEM 17
 JNENVPTR 19
 LENGTH OF 19
 LINAGE-COUNTER 20
 RETURN-CODE 21
 SHIFT-OUT, SHIFT-IN 21
 SORT-CONTROL 22
 SORT-CORE-SIZE 22
 SORT-FILE-SIZE 23
 SORT-MESSAGE 23
 SORT-MODE-SIZE 23
 SORT-RETURN 24
 TALLY 24
 WHEN-COMPILED 24
 XML-CODE 25
 XML-EVENT 26
 XML-NTEXT 27
 XML-TEXT 28
 SPECIAL-NAMES paragraph
 ACCEPT statement 304
 ALPHABET clause 113
 CLASS clause 116
 CODE-SET clause and 180
 CURRENCY SIGN clause 117
 DECIMAL-POINT IS COMMA
 clause 118
 description 110
 format 110
 mnemonic-name 113
 SQR function 527

- standard alignment
 - JUSTIFIED clause 190
- standard alignment rules 164
- STANDARD-1
 - RECORD DELIMITER clause 136
- STANDARD-1 phrase 114
- STANDARD-2 phrase 114
- STANDARD-DEVIATION function 528
- START statement
 - description and format 441
 - indexed file 442
 - INVALID KEY phrase 300, 442
 - relative files 443
 - status key considerations 442
- statement operations
 - common phrases 291
 - file position indicator 302
 - INTO and FROM phrases 301
- statements
 - categories of 286
 - conditional 288
 - data manipulation 296
 - definition 44
 - delimited scope 289
 - description 257
 - imperative 286
 - input-output 296
 - procedure branching 303
 - syntactical hierarchy 43
 - types of 44
- static data 87
- static method 87
- status key
 - common processing facility 297
 - file processing 557
- STL files
 - size limits 581
- STOP RUN statement 444
- STOP statement 444
- storage
 - map listing 540
 - MEMORY SIZE clause 109
 - REDEFINES clause 216
- STRING statement
 - description and format 445
 - execution of 447
 - overlapping operands, unpredictable results 295
- structure of the COBOL language 3
- structured programming
 - DO-WHILE and DO-UNTIL 399
- subclass 87
- subclasses and methods 101
- subjects in EVALUATE statement 342
- subprogram linkage
 - CALL statement 314
 - CANCEL statement 322
 - ENTRY statement 340
- subprogram termination
 - CANCEL statement 322
 - EXIT PROGRAM statement 347
 - GOBACK statement 348
- subscripting
 - definition and format 65
 - INDEXED BY phrase of OCCURS clause 195
 - MOVE statement evaluation 381

- subscripting (*continued*)
 - OCCURS clause specification 192
 - table references 65
 - using data-names 67
 - using index-names (indexing) 67
 - using integers 67
- substitution characters
 - DISPLAY-OF 508
 - NATIONAL-OF 519
- substitution field of INSPECT
 - REPLACING 359
- substrings, specifying
 - (reference-modification) 68
- SUBTRACT statement
 - common phrases 291
 - description and format 450
- SUM function 528
- SUPER special object identifier 13, 367
- superclass 87
- support 645
- SUPPRESS option, COPY 543
- suppress output 538
- suppression editing 215
- switch-status condition 280
- SYMBOLIC CHARACTERS clause 116
- symbolic-character 12, 55
- symbolic-character figurative
 - constant 15
- symbols in PICTURE clause 199
- SYNCHRONIZED clause 223
 - effect on other language elements 224
 - VALUE clause and 239
- syntax notation, rules for xiii
- system considerations, subprogram linkage
 - CALL statement 314
 - CANCEL statement 322
- system information transfer, ACCEPT statement 305
- system-names 12, 108
 - computer-name 108
 - SOURCE-COMPUTER paragraph 108

T

- table references
 - indexing 67
 - subscripting 65
- TALLY special register 24
- TALLYING phrase
 - INSPECT statement 359
 - UNSTRING statement 457
- TAN function 529
- termination of execution
 - EXIT METHOD statement 346
 - EXIT PROGRAM statement 347
 - GOBACK statement 348
 - STOP RUN statement 444
- terminators, scope 290
- text words 542
- text-name 12, 55
 - literal-1 541
- THREAD compiler option 195
 - requirement for indexed items 195

- THROUGH (THRU) phrase
 - ALPHABET clause 114
 - CLASS clause 116
 - EVALUATE statement 342
 - PERFORM statement 396
 - RENAMES clause 220
 - VALUE clause 241
- TIME 307
- TIMES phrase of PERFORM statement 398
- TITLE statement 555
- TO phrase, SET statement 427
- TO TRUE phrase, SET statement 430
- transfer of control
 - ALTER statement 312, 313
 - explicit 73
 - GO TO statement 349
 - IF statement 352
 - implicit 73
 - PERFORM statement 396
 - XML PARSE statement 480
- transfer of data
 - ACCEPT statement 304
 - MOVE statement 380
 - STRING statement 445
 - UNSTRING statement 454
- trimming of generated XML data 478
- TRUNC compiler option 166
- truncation of data
 - arithmetic item 166
 - JUSTIFIED clause 190
 - ROUNDED phrase 292
 - TRUNC compiler option 166
- truth value
 - complex conditions 280
 - EVALUATE statement 342
 - IF statement 351
 - of complex condition 281
 - sign condition 279
 - with conditional statement 288
- type conformance
 - SET..USAGE OBJECT REFERENCE 433
- types of functions 490

U

- unary operator 258
- unconditional GO TO statement 349
- UNDATE function 529
- Unicode 3, 6
- unique names 159
- uniqueness of reference 59
- unit file, definition 326
- UNIT phrase 324
- universal object reference 235
- unsigned numeric item, definition 205
- UNSTRING statement
 - description and format 454
 - execution 458
 - overlapping operands, unpredictable results 295
 - receiving field 457
 - sending field 454
- UP BY phrase, SET statement 428
- UPON phrase, DISPLAY 333
- UPPER-CASE function 530

- UPSI-0 through UPSI-7, program switches
 - and switch-status condition 280
 - condition-name 113
 - processing special conditions 113
 - SPECIAL-NAMES paragraph 113
- USAGE clause
 - BINARY phrase 231
 - CODE-SET clause and 180
 - COMPUTATIONAL phrases 232
 - description 229
 - DISPLAY phrase 233
 - DISPLAY-1 phrase 234
 - format 229
 - FUNCTION-POINTER phrase 234
 - INDEX phrase 234
 - NATIONAL phrase 191, 235
 - operational signs and 166
 - PACKED-DECIMAL phrase 232
 - POINTER phrase 236
 - PROCEDURE-POINTER phrase 237
 - VALUE clause and 239
- USAGE COMP-1
 - size of items 165
- USAGE COMP-2
 - size of items 165
- USAGE DISPLAY
 - size of items 165
 - STRING statement and 446
- USAGE DISPLAY-1
 - size of items 165
 - STRING statement and 446
- USAGE NATIONAL
 - size of items 165
 - STRING statement and 446
- USAGE OBJECT REFERENCE
 - phrase 366
- USE FOR DEBUGGING declarative 594
- USE statement
 - format and description 556
- user labels
 - DEBUGGING declarative 558
 - LABEL declarative 558
- user-defined words 11
- USING phrase
 - CALL statement 316
 - in procedure division header 251
 - INVOKE statement 368
 - MERGE statement 377
 - procedure division header 252
 - SORT statement 437
 - subprogram linkage 254
- UTF-16 3, 6
- UTF-8 6

V

- V symbol in PICTURE clause 200
- VALUE clause
 - condition-name 241
 - effect on object-oriented programs 153
 - format 239, 241
 - level 88 item 159
 - NULL/NULLS figurative constant 234, 244
 - rules for condition-name entries 242
 - rules for literal values 240

- VALUE OF clause
 - description 177
 - format 167
- variable-length tables 195, 196
- VARIANCE function 530
- VARYING phrase
 - PERFORM statement 400
 - SEARCH statement 422

W

- WHEN phrase
 - EVALUATE statement 342
 - SEARCH statement (binary search) 424
 - SEARCH statement (serial search) 423
- WHEN-COMPILED function 531
- WHEN-COMPILED special register 24
- windowed date field
 - expansion before use 185
- windowed date field (See also date field)
 - definition 76
- WITH DEBUGGING MODE clause 108, 558, 593
- WITH DUPLICATES phrase, SORT statement 436
- WITH FOOTING phrase 178
- WITH NO ADVANCING phrase 334
- WITH NO REWIND phrase, CLOSE statement 326
- WITH POINTER phrase
 - STRING statement 445
 - UNSTRING statement 457
- working-storage section 153
- WRITE statement
 - AFTER ADVANCING 465
 - ALTERNATE RECORD KEY 468
 - BEFORE ADVANCING 465
 - description 462
 - END-OF-PAGE phrase 466
 - format 462
 - FROM identifier phrase 301
 - indexed files 468
 - NOT END-OF-PAGE phrase 466
 - relative files 468
 - sequential files 467

X

- X symbol in PICTURE clause 201
- X'00' - X'1F' control characters 30
- XML GENERATE statement
 - ATTRIBUTES phrase 474
 - COUNT IN phrase 472
 - description 470
 - element name formation 479
 - ENCODING phrase 473
 - END-XML phrase 475
 - exception event 475
 - format 470
 - format conversion 477
 - NAMESPACE phrase 474
 - NAMESPACE-PREFIX phrase 474
 - NOT ON EXCEPTION phrase 475
 - ON EXCEPTION phrase 475

- XML GENERATE statement (*continued*)
 - operation 476
 - trimming 478
 - XML-DECLARATION phrase 473
- XML PARSE statement
 - control flow 484
 - description 480
 - exception event 482
 - format 480
 - nested XML GENERATE 483
 - nested XML PARSE 483
 - ON EXCEPTION phrase 482
 - PROCESSING PROCEDURE phrase 481
- XML processing
 - ENCODING phrase, in XML GENERATE 473
 - PROCESSING PROCEDURE phrase, in XML PARSE 481
 - XML-CODE special register 25, 480
 - XML-EVENT special register 26, 480
 - XML-NTEXT special register 27, 480
 - XML-TEXT special register 28, 480
- XML-CODE special register 25
 - use in XML GENERATE 475
 - use in XML PARSE 482
- XML-DECLARATION phrase 473
- XML-EVENT special register 26, 484
- XML-NTEXT special register 27, 484
- XML-TEXT special register 28, 484
- XML, definition 644

Y

- year-last date field (See also date field)
 - definition 76
- YEAR-TO-YYYY function 532
- YEARWINDOW compiler option
 - century window 77
- YEARWINDOW function 533

Z

- Z
 - insertion character 215
 - null-terminated literals 31
 - symbol in PICTURE clause 201
 - Z literals 31
- zero
 - filling, elementary moves 381
 - suppression and replacement editing 215
- ZERO figurative constant 13
- ZERO in sign condition 279
- ZEROES figurative constant 13
- ZEROS figurative constant 13

Readers' Comments — We'd Like to Hear from You

COBOL for AIX
Language Reference
Version 4.1

Publication No. SC27-3602-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Send your comments to the address on the reverse side of this form.

If you would like a response from IBM, please fill in the following information:

Name

Address

Company or Organization

Phone No.

E-mail address



Cut or Fold
Along Line

Fold and Tape

Please do not staple

Fold and Tape



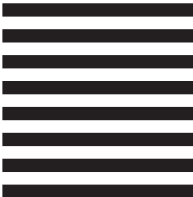
NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Reader Comments
DTX/E269
555 Bailey Avenue
San Jose, CA
U.S.A. 95141-9989



Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold
Along Line



Program Number: 5724-Z87

Printed in USA

SC27-3602-00

