

Enterprise PL/I for z/OS



プログラミング・ガイド

バージョン 3 リリース 8

Enterprise PL/I for z/OS



プログラミング・ガイド

バージョン 3 リリース 8

— お願い —

本書および本書で紹介する製品をご使用になる前に、523 ページの『特記事項』に記載されている情報をお読みください。

本書は、Enterprise PL/I for z/OS のバージョン 3 リリース 8 (5655-H31)、および新しい版または TNL で明記されていない限り、以降のすべてのリリースに適用されます。製品のレベルに合った正しい版をご使用ください。

資料のご注文方法については、<http://www.ibm.com/jp/manuals> の「ご注文について」をご覧ください。（URL は、変更になる場合があります）

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC27-1457-08
Enterprise PL/I for z/OS
Programming Guide
Version 3 Release 8

発行： 日本アイ・ピー・エム株式会社

担当： ナショナル・ランゲージ・サポート

第10版第1刷 2008.12

© Copyright International Business Machines Corporation 1999, 2008. All rights reserved.

目次

表	ix
-------------	----

図	xi
-------------	----

概要	xiii
--------------	------

本書について	xiii
------------------	------

Enterprise PL/I for z/OS のランタイム環境	xiii
---	------

資料の使用	xiv
-----------------	-----

PL/I 情報	xiv
-------------------	-----

本書で用いられる表記規則	xv
------------------------	----

使用する規則	xv
------------------	----

構文記法の読み方	xv
--------------------	----

表記記号の読み方	xviii
--------------------	-------

本リリースにおける機能強化	xix
-------------------------	-----

パフォーマンス向上	xix
---------------------	-----

使いやすさの向上	xix
--------------------	-----

V3R7 からの機能拡張	xx
------------------------	----

デバッグの向上	xx
-------------------	----

パフォーマンス向上	xx
---------------------	----

使いやすさの向上	xx
--------------------	----

V3R6 からの機能拡張	xxi
------------------------	-----

DB2 V9 サポート	xxi
-----------------------	-----

デバッグの向上	xxii
-------------------	------

パフォーマンス向上	xxii
---------------------	------

使いやすさの向上	xxii
--------------------	------

V3R5 からの機能拡張	xxiii
------------------------	-------

デバッグの向上	xxiii
-------------------	-------

パフォーマンス向上	xxiii
---------------------	-------

使いやすさの向上	xxiii
--------------------	-------

V3R4 からの機能拡張	xxiv
------------------------	------

マイグレーションの強化	xxiv
-----------------------	------

パフォーマンス向上	xxiv
---------------------	------

使いやすさの向上	xxv
--------------------	-----

デバッグの向上	xxv
-------------------	-----

V3R3 からの機能拡張	xxvi
------------------------	------

XML サポートの強化	xxvi
-----------------------	------

パフォーマンスの改善	xxvi
----------------------	------

容易なマイグレーション	xxvi
-----------------------	------

使いやすさの向上	xxvii
--------------------	-------

デバッグ・サポートの向上	xxvii
------------------------	-------

V3R2 からの機能拡張	xxvii
------------------------	-------

パフォーマンスの改善	xxvii
----------------------	-------

容易なマイグレーション	xxviii
-----------------------	--------

使いやすさの向上	xxix
--------------------	------

V3R1 からの機能拡張	xxix
------------------------	------

VisualAge PL/I からの機能拡張	xxx
----------------------------------	-----

第 1 部 プログラムのコンパイル . . . 1

第 1 章 コンパイラー・オプションと機能の使用 5

コンパイル時オプションの説明	5
--------------------------	---

AGGREGATE	8
---------------------	---

ARCH	9
----------------	---

ATTRIBUTES	10
----------------------	----

BACKREG	10
-------------------	----

BIFPREC	11
-------------------	----

BLANK	12
-----------------	----

BLKOFF	12
------------------	----

CEESTART	12
--------------------	----

CHECK	13
-----------------	----

CMPAT	14
-----------------	----

CODEPAGE	15
--------------------	----

COMMON	16
------------------	----

COMPACT	16
-------------------	----

COMPILE	17
-------------------	----

COPYRIGHT	18
---------------------	----

CSECT	18
-----------------	----

CSECTCUT	19
--------------------	----

CURRENCY	19
--------------------	----

DBCS	19
----------------	----

DD	20
--------------	----

DDSQL	20
-----------------	----

DECIMAL	20
-------------------	----

DEFAULT	22
-------------------	----

DISPLAY	31
-------------------	----

DLLINIT	32
-------------------	----

EXIT	32
----------------	----

EXTRN	32
-----------------	----

FLAG	33
----------------	----

FLOAT	33
-----------------	----

FLOATINMATH	36
-----------------------	----

GOFF	36
----------------	----

GONUMBER	37
--------------------	----

GRAPHIC	37
-------------------	----

HGPR	38
----------------	----

INCAFTER	38
--------------------	----

INCDIR	39
------------------	----

INCPDS	39
------------------	----

INITAUTO	40
--------------------	----

INITBASED	40
---------------------	----

INITCTL	41
-------------------	----

INITSTATIC	41
----------------------	----

INSOURCE	42
--------------------	----

INTERRUPT	42
---------------------	----

LANGLVL	43
-------------------	----

LIMITS	44
------------------	----

LINECOUNT	45
---------------------	----

LINEDIR	45
-------------------	----

LIST	45	WINDOW	84
LISTVIEW	46	WRITABLE	84
MACRO	47	XINFO	85
MAP	47	XML	88
MARGINI	47	XREF	88
MARGINS	48	オプションの中のブランク、コメント、およびスト リング	89
MAXMEM	49	デフォルト・オプションの変更	89
MAXMSG	50	%PROCESS ステートメントまたは *PROCESS ステ ートメントでのオプションの指定	90
MAXNEST	50	% ステートメントの使用	91
MAXSTMT	51	%INCLUDE ステートメントの使用	92
MAXTEMP	51	%OPTION ステートメントの使用	93
MDECK	51	コンパイラー・リストの使用	94
NAME	52	見出し情報	94
NAMES	52	コンパイルに使用するオプション	95
NATLANG	52	プリプロセッサ入力	95
NEST	53	SOURCE プログラム	95
NOT	53	ステートメントのネスト・レベル	95
NUMBER	53	ATTRIBUTE と相互参照テーブル	96
OBJECT	54	集合長さテーブル	97
OFFSET	54	ステートメント・オフセット・アドレス	97
OPTIMIZE	55	ストレージ・オフセット・リスト	100
OPTIONS	56	ファイル参照テーブル	101
OR	56	メッセージと戻りコード	102
PP	57	第 2 章 PL/I プリプロセッサ 105	
PPCICS	58	インクルード・プリプロセッサ	106
PPINCLUDE	58	マクロ・プリプロセッサ	107
PPMACRO	59	マクロ・プリプロセッサのオプション	107
PPSQL	60	マクロ・プリプロセッサの例	109
PPTRACE	60	SQL プリプロセッサ	110
PRECTYPE	60	プログラミングとコンパイルに関する考慮事項	111
PREFIX	61	SQL プリプロセッサ・オプション	112
PROCEED	61	PL/I アプリケーション内での SQL ステートメ ントのコーディング	119
PROCESS	62	ラージ・オブジェクト (LOB) サポートに関する 追加情報	128
QUOTE	63	SQL データ・タイプと PL/I データ・タイプの 互換性の判別	131
REDUCE	63	ホスト構造体の使用	131
RENT	64	標識変数の使用	132
RESEXP	65	ホスト構造体の例	133
RESPECT	65	コンパイラー・ユーザー出口 (IBMUEXIT) での SQL プリプロセッサの使用	133
RULES	66	DECLARE STATEMENT ステートメント	134
SEMANTIC	72	CICS プリプロセッサ	134
SERVICE	73	プログラミングとコンパイルに関する考慮事項	135
SOURCE	73	CICS プリプロセッサのオプション	136
SPILL	73	PL/I アプリケーション内での CICS ステートメ ントのコーディング	136
STATIC	74	PL/I を使用した CICS トランザクションの作成 エラー処理	136
STDSYS	74	第 3 章 PL/I カタログ式プロシージャー の用法 139	
STMT	74		
STORAGE	75		
STRINGOFGRAPHIC	75		
SYNTAX	75		
SYS Parm	76		
SYSTEM	77		
TERMINAL	78		
TEST	78		
TUNE	82		
USAGE	82		
WIDECHAR	83		

IBM 提供のカatalog式プロシージャー	139
コンパイルのみ (IBMZC)	140
コンパイルおよびバインド (IBMZCB)	142
コンパイル、バインド、および実行 (IBMZCBG)	144
コンパイル、プリリンク、およびリンク・エディット (IBMZCPL)	146
コンパイル、プリリンク、リンク・エディット、および実行 (IBMZCPLG)	148
コンパイル、プリリンク、ロード、および実行 (IBMZCPG)	150
カatalog式プロシージャーの呼び出し	152
複数呼び出しの指定	153
PL/I カatalog式プロシージャーの変更	154
EXEC ステートメント	154
DD ステートメント	155

第 4 章 プログラムのコンパイル . . . 157

z/OS UNIX の下でのコンパイラーの呼び出し	157
入力ファイル	157
z/OS UNIX の下でのコンパイル時オプションの指定	158
-qoption_keyword	158
単一フラグおよび複数文字フラグ	159
JCL を使用した z/OS の下でのコンパイラーの呼び出し	160
EXEC ステートメント	160
標準データ・セット用の DD ステートメント	160
リスト (SYSPRINT)	162
ソース・ステートメント・ライブラリー (SYSLIB)	163
オプションの指定	163
EXEC ステートメントでのオプションの指定	164
オプション・ファイルを使用した EXEC ステートメントでのオプションの指定	164

第 5 章 リンク・エディットと実行 . . 167

リンク・エディットに関する考慮事項	167
バインダーの使用	167
プリリンカーの使用	168
ENTRY カードの使用	168
実行時の考慮事項	168
PRINT ファイルのフォーマット設定に関する規則	168
PRINT ファイルでのフォーマットの変更	169
自動プロンプト	170
長い入力行の句読法	171
GET LIST ステートメントと GET DATA ステートメントの句読法	171
ENDFILE	172
SYSPRINT の考慮事項	172
ルーチン内での FETCH の使用	174
Enterprise PL/I ルーチンのフェッチ	174
z/OS C ルーチンのフェッチ	182
アセンブラー・ルーチンのフェッチ	182
z/OS UNIX を指定した場合の MAIN の呼び出し	182

第 2 部 入出力機能の使用 185

第 6 章 データ・セットとファイルの使用 187

z/OS でのデータ・セットとファイルの関連付け	187
複数のファイルと 1 つのデータ・セットの関連付け	189
複数のデータ・セットと 1 つのファイルの関連付け	190
複数のデータ・セットの連結	190
z/OS での HFS ファイルへのアクセス	190
z/OS UNIX でのデータ・セットとファイルの関連付け	191
環境変数の使用	191
OPEN ステートメントの TITLE オプションの使用	192
データ・セットに関連付けられていないファイルの使用の試み	193
PL/I によるデータ・セットの検索方法	194
DD_DDNAME 環境変数の使用による特性の指定	194
データ・セット特性の設定	200
ブロックおよびレコード	200
レコード・フォーマット	201
データ・セットの編成	203
ラベル	204
データ定義 (DD) ステートメント	204
OPEN ステートメントの TITLE オプションの使用	206
PL/I ファイルとデータ・セットの関連付け	207
ENVIRONMENT 属性での特性の指定	208

第 7 章 ライブラリーの使用 221

ライブラリーのタイプ	221
ライブラリーの用法	222
ライブラリーの作成	222
SPACE パラメーター	222
ライブラリー・メンバーの作成と更新	223
例	224
ライブラリー・ディレクトリーにある情報の取り出し	226

第 8 章 連続データ・セットの定義と使用 227

ストリーム指向データ伝送の使用	227
ストリーム入出力の使用によるファイルの定義	228
ENVIRONMENT オプションの指定	228
ストリーム入出力によるデータ・セットの作成	230
ストリーム入出力によるデータ・セットへのアクセス	233
ストリーム入出力による PRINT ファイルの使用	235
SYSIN ファイルおよび SYSPRINT ファイルの使用	240
使用方法	240
端末からの入力の制御	241
データのフォーマット	241
ストリーム・ファイルおよびレコード・ファイル	242

大文字と小文字.	243
ファイルの終わり.	243
GET ステートメントの COPY オプション	243
端末への出力の制御	243
PRINT ファイルのフォーマット	243
ストリーム・ファイルおよびレコード・ファイル	244
大文字と小文字.	244
PUT EDIT コマンドの出力.	244
レコード単位データ伝送の使用	244
レコード・フォーマットの指定	246
レコード入出力の使用によるファイルの定義	246
ENVIRONMENT オプションの指定	246
レコード入出力によるデータ・セットの作成	249
レコード入出力によるデータ・セットへのアクセ スと更新.	250

第 9 章 領域データ・セットの定義と使用 257

領域データ・セット用のファイルの定義	259
ENVIRONMENT オプションの指定	260
REGIONAL データ・セットでのキーの使用	260
REGIONAL(1) データ・セットの使用	261
ダミー・レコード.	261
REGIONAL(1) データ・セットの作成	261
REGIONAL(1) データ・セットへのアクセスと更 新.	263
領域データ・セットの作成時、および領域データ・ セットへのアクセス時の必須情報.	267

第 10 章 VSAM データ・セットの定義と使用 271

VSAM データ・セットの使用.	271
VSAM データ・セットでのプログラムの実行	271
代替索引パスとファイルのベア化.	271
VSAM 編成.	272
VSAM データ・セットのキー.	274
データ・セット・タイプの選択	275
VSAM データ・セットのファイルの定義	277
ENVIRONMENT オプションの指定	278
パフォーマンス・オプション	281
代替索引パス用のファイルの定義.	281
VSAM データ・セットの定義.	282
入力順データ・セット	283
ESDS のロード.	284
SEQUENTIAL ファイルの使用による ESDS へ のアクセス	284
非 VSAM データ・セット用に定義されたファイル の使用.	306
共用データ・セットの使用	306

第 3 部 プログラムの改良 307

第 11 章 パフォーマンスの向上 309

最適なパフォーマンスのためのコンパイラー・オブ ションの選択	309
---	-----

OPTIMIZE	309
GONUMBER.	310
ARCH.	310
REDUCE.	310
RULES	310
PREFIX	311
DEFAULT	312
パフォーマンスを向上させるコンパイラー・オブ ションの要約	315
パフォーマンス向上のためのコーディング.	316
DATA ディレクティブ入出力.	316
入力専用パラメーター	317
GOTO ステートメント	317
ストリングの割り当て	317
ループ制御変数.	318
PACKAGE 対ネストされた PROCEDURE	318
REDUCIBLE 関数.	319
DESCLOCATOR または DESCLIST	320
DEFINED 対 UNION.	320
名前付き定数対静的変数.	320
ライブラリー・ルーチンの呼び出しの回避.	322
ライブラリー・ルーチンのプリロード	323

第 4 部 他の製品に対するインターフェースの使用 325

第 12 章 ソート・プログラムの使用 327

ソート・プログラムの使用準備	328
ソート・タイプの選択	328
ソート・フィールドの指定.	331
ソートするレコードの指定.	333
ソート・プログラムに必要なストレージの決定	334
ソート・プログラムの呼び出し	334
例 1	336
例 2	336
例 3	337
例 4	337
例 5	337
ソートが成功したかどうかの判別.	337
ソート・プログラム用のデータ・セットの確立	338
ソート・データの入出力.	339
データ入出力処理ルーチン.	339
E15 - 入力処理ルーチン (ソート出口 E15)	340
E35 - 出力処理ルーチン (ソート出口 E35)	343
PLISRTA の呼び出し例	344
PLISRTB の呼び出し例	346
PLISRTC の呼び出し例	347
PLISRTD の呼び出し例	348
可変長レコードのソートの例	350

第 13 章 C との ILC 353

同等なデータ・タイプ	353
単純なタイプの一致	353
struct タイプの一致	354
enum タイプの一致	354

ファイル・タイプの一致	355
C 関数の使用	355
一致する単純パラメーター・タイプ	357
一致するストリング・パラメーター・タイプ	359
ENTRY を戻す関数	361
リンケージ	362
出力の共用	364
要約	364

第 14 章 Java とのインターフェース 367

Java Native Interface (JNI) の概要	367
JNI サンプル・プログラム #1 - 'Hello World'	368
Java サンプル・プログラム #1 の作成	368
JNI サンプル・プログラム #2 - スtringの引き渡し	372
Java サンプル・プログラム #2 の作成	372
JNI サンプル・プログラム #3 - 整数の引き渡し	376
Java サンプル・プログラム #3 の作成	376
JNI サンプル・プログラム #4 - Java 呼び出し API	381
Java サンプル・プログラム #4 の作成	381
Java および PL/I の同等なデータ・タイプの判別	385

第 5 部 特殊プログラミング・タスク 387

第 15 章 PLISAXA および PLISAXB XML パーサーの使用 389

概要	389
PLISAXA 組み込みサブルーチン	390
PLISAXB 組み込みサブルーチン	390
SAX イベント構造体	391
start_of_document	392
version_information	392
encoding_declaration	392
standalone_declaration	392
document_type_declaration	392
end_of_document	392
start_of_element	392
attribute_name	392
attribute_characters	393
attribute_predefined_reference	393
attribute_character_reference	393
end_of_element	393
start_of_CDATA_section	393
end_of_CDATA_section	393
content_characters	394
content_predefined_reference	394
content_character_reference	394
processing_instruction	394
comment	394
unknown_attribute_reference	395
unknown_content_reference	395
start_of_prefix_mapping	395
end_of_prefix_mapping	395
exception	395

イベント関数に渡されるパラメーター	395
XML 文書のコード化文字セット	396
サポートされる EBCDIC コード・ページ	396
サポートされる ASCII コード・ページ	397
コード・ページの指定	397
例外	398
例	399
継続可能な例外コード	411
例外コードの終了	415

第 16 章 PLISAXC XML パーサーの使用 419

概要	419
PLISAXC 組み込みサブルーチン	420
SAX イベント構造体	420
start_of_document	421
version_information	421
encoding_declaration	421
standalone_declaration	421
document_type_declaration	421
end_of_document	422
start_of_element	422
attribute_name	422
attribute_characters	422
end_of_element	422
start_of_CDATA_section	422
end_of_CDATA_section	423
content_characters	423
processing_instruction	423
comment	423
namespace_declare	423
end_of_input	423
unresolved_reference	424
exception	424
イベント関数に渡されるパラメーター	424
イベントにおける差異	426
XML 文書のコード化文字セット	426
サポートされるコード・ページ	427
コード・ページの指定	427
例外	428
単純な文書での例	428

第 17 章 PLIDUMP の用法 439

PLIDUMP の使用上の注意	440
PLIDUMP 出力内の変数の検出	441
AUTOMATIC 変数の検出	441
STATIC 変数の検出	442
CONTROLLED 変数の検出	444
保存されたコンパイル・データ	447
タイム・スタンプ	447
保存されたオプション・ストリング	448

第 18 章 割り込みとアテンションの処理 453

ATTENTION ON ユニットの使用	454
デバッグ・ツールとの対話	454

第 19 章 チェックポイント/再始動機能の使用	455
チェックポイント・レコードの要求	455
チェックポイント・データ・セットの定義	456
再始動の要求	457
システム障害後の自動再始動	457
プログラム内の自動再始動	458
据え置き再始動	458
チェックポイント/再始動活動の変更	458
第 20 章 ユーザー出口の用法	459
コンパイラー・ユーザー出口によって実行されるプロセス	459
コンパイラー・ユーザー出口の活動化	460
IBM 提供のコンパイラー出口、IBMUEXIT	460
コンパイラー・ユーザー出口のカスタマイズ	461
SYSUEXIT の変更	461
独自のコンパイラー出口の作成	461
グローバル制御ブロックの構造	462
初期化プロセスの作成	463
メッセージ・フィルター操作プロセスの作成	463
終了プロセスの作成	465
第 21 章 PL/I 記述子	467
引数の引き渡し	467
記述子リストによる引数の引き渡し	467
記述子ロケーターによる引数の引き渡し	468
CMPAT(V*) 記述子	468
ストリング記述子	468
配列記述子	470
CMPAT(LE) 記述子	470
ストリング記述子	471
配列記述子	471
第 6 部 付録	473

付録. SYSADATA メッセージ情報	475
SYSADATA ファイルについて	475
サマリー・レコード	477
カウンター・レコード	477
リテラル・レコード	478
ファイル・レコード	478
メッセージ・レコード	479
SYSADATA シンボル情報について	479
序数タイプ・レコード	480
序数エレメント・レコード	481
シンボル・レコード	481
SYSADATA 構文情報について	497
ソース・レコード	497
トークン・レコード	497
構文レコード	498

特記事項	523
商標	524

参考文献	525
Enterprise PL/I 資料	525
PL/I for MVS & VM	525
z/OS 言語環境プログラム	525
CICS Transaction Server	525
DB2 UDB (OS/390 版および z/OS 版)	525
DFSORT	526
IMS/ESA	526
z/OS MVS	526
z/OS UNIX システム・サービス	526
z/OS TSO/E	526
z/Architecture	526
Unicode および文字表現	527

用語集	529
------------	------------

索引	545
-----------	------------

表

1. Enterprise PL/I 付属資料の使用法	xiv	21. 領域データ・セットの作成と領域データ・セ	
2. z/OS 言語環境プログラム 付属資料の使用法	xiv	ットへのアクセスで利用できるステートメン	
3. コンパイル時オプション、省略形、および IBM		トとオプション	258
提供のデフォルト値	5	22. 領域データ・セットの作成: DD ステートメン	
4. SYSTEM オプション・テーブル	77	トの必須パラメーター	268
5. PL/I エラー・コードと戻りコードの説明	103	23. 領域データ・セットの DCB サブパラメータ	
6. リストされたメッセージの最低重大度を選択		ー	269
するための FLAG オプションの使用	103	24. 領域データ・セットへのアクセス: DD ステー	
7. PL/I 宣言から生成される SQL データ・タイ		トメントの必須パラメーター	269
プ	126	25. VSAM データ・セットのタイプと利点	274
8. メタ PL/I 宣言から生成される SQL データ・		26. 代替索引パスで実行できる処理	277
タイプ	126	27. VSAM 入力順データ・セットのロードと入力	
9. SQL データ・タイプと PL/I 宣言の対応	127	順データ・セットへのアクセスで利用できる	
10. SQL データ・タイプとメタ PL/I 宣言の対応	127	ステートメントとオプション	283
11. z/OS UNIX の下で Enterprise PL/I によりサポ		28. VSAM 索引付きデータ・セットのロードとそ	
ートされるコンパイル時オプション・フラグ	159	れへのアクセスに使用できるステートメント	
12. コンパイラーの標準データ・セット	161	とオプション	286
13. PL/I ファイル宣言の属性	209	29. VSAM 相対レコード・データ・セットのロー	
14. PL/I レコード入出力で利用できるデータ・セ		ドとそれへのアクセスに使用できるステート	
ット・タイプの比較	217	メントとオプション	299
15. ライブラリー作成時に必要な情報	222	30. 各エントリー・ポイントと PLISRTx (x =	
16. 連続データ・セットの作成と連続データ・セ		A、B、C、または D) への引数	335
ットへのアクセスで利用できるステートメン		31. C と PL/I の同等なタイプ	353
トとオプション	245	32. Java の基本的なタイプと同等な PL/I ネイテ	
17. IBM マシン・コード印刷制御文字 (CTL360)	248	ィブのタイプ	385
18. LEAVE および REREAD オプションの影響	249	33. 継続可能な例外	411
19. レコード入出力による連続データ・セットの		34. 終了例外	415
作成: DD ステートメントの必須パラメーター	250		
20. レコード入出力による連続データ・セットへ			
のアクセス: DD ステートメントの必須パラメ			
ーター	251		



1. ライブラリーからのソース・ステートメントの 組み込み	93	34. VSAM データ・セットと使用できるファイル 属性	277
2. ステートメント番号の検索 (コンパイラー・リ ストの例)	99	35. 入力順データ・セット (ESDS) の定義とロー ド	285
3. ステートメント番号の検索 (ランタイム・メッ セージの例)	99	36. ESDS の更新	286
4. ソース・デックを作成するためのマクロ・プ リプロセッサの使用	110	37. キー順データ・セット (KSDS) の定義とロー ド	289
5. SQLCA の PL/I 宣言	120	38. KSDS の更新	291
6. SQL 記述子域の PL/I 宣言	121	39. ESDS 用の固有キー代替索引パスの作成	293
7. カタログ式プロシージャの呼び出し	140	40. ESDS 用の非固有キー代替索引パスの作成	294
8. カタログ式プロシージャ IBMZC	142	41. KSDS 用の固有キー代替索引パスの作成	295
9. カタログ式プロシージャ IBMZCB	143	42. ESDS での代替索引パスと逆方向読み取り	297
10. カタログ式プロシージャ IBMZCBG	145	43. KSDS アクセス用の固有代替索引パスの使用	299
11. カタログ式プロシージャ IBMZCPL	147	44. 相対レコード・データ・セット (RRDS) の定 義とロード	303
12. カタログ式プロシージャ IBMZCPLG	149	45. RRDS の更新	305
13. カタログ式プロシージャ IBMZCPG	151	46. ソート・プログラムの制御の流れ	330
14. PLITABS の宣言	169	47. 入力および出力処理サブルーチンのフローチ ャート	341
15. PAGELength および PAGESIZE	170	48. 入力プロシージャ用の骨組みコード	342
16. ユーザー出口のコンパイル、リンク、および 呼び出しのためのサンプル JCL	176	49. 出力処理プロシージャ用の骨組みコード	344
17. z/OS UNIX 引数と環境変数を表示するサンプ ル・プログラム	183	50. PLISRTA - 入力データ・セットから出力デー タ・セットへのソート	345
18. 固定長レコード	202	51. PLISRTB - 入力処理ルーチンから出力デー タ・セットへのソート	346
19. オペレーティング・システムによる DCB へ の情報の組み込みの方法	208	52. PLISRTC - 入力データ・セットから出力処理 ルーチンへのソート	347
20. コンパイルされたオブジェクト・モジュール 用の新規ライブラリーの作成	224	53. PLISRTD - 入力処理ルーチンから出力処理ル ーチンへのソート	348
21. ロード・モジュールの既存ライブラリーへの 配置	225	54. 入出力処理ルーチンを使った可変長レコード のソート	350
22. PL/I プログラム内でのライブラリー・メンバ ーの作成	225	55. 単純なタイプ的一致	354
23. ライブラリー・メンバーの更新	226	56. struct タイプの一致の例	354
24. ストリーム指向データ伝送によるデータ・セ ットの作成	232	57. enum タイプの一致の例	355
25. グラフィック・データのストリーム・ファイ ルへの書き込み	233	58. FILE タイプの C 宣言の開始	355
26. ストリーム指向データ伝送によるデータ・セ ットへのアクセス	235	59. C ファイルと一致する PL/I	355
27. ストリーム・データ伝送による印刷ファイル の作成	238	60. fopen と fread を使用してファイルをダンプす るコードの例	356
28. 事前設定済みのタブ設定を変更する場合の PL/I 構造体 PLITABS	240	61. filedump プログラムの宣言	356
29. 米国標準規格の印刷およびカード穿孔制御文 字 (CTLASA)	248	62. fread の C 宣言	357
30. 連続データ・セットのマージ、ソート、作成 と連続データ・セットへのアクセス	253	63. fread の誤った宣言 (その 1)	357
31. レコード単位データ伝送の印刷	255	64. fread の誤った宣言 (その 2)	357
32. REGIONAL(1) データ・セットの作成	263	65. fread の誤った宣言 (その 3)	357
33. REGIONAL(1) データ・セットの更新	266	66. RETURNS BYADDR に対して生成されるコー ド	358
		67. fread の正しい宣言	358
		68. RETURNS BYVALUE に対して生成されるコ ード	359
		69. fopen の誤った宣言 (その 1)	359
		70. fopen の誤った宣言 (その 2)	359
		71. fopen の正しい宣言	360

72. fopen の正しく最適な宣言	360	98. PLISAXC のコーディング例 - イベント・ルーチン	432
73. fclose の宣言	360	99. PLISAXC のコーディング例 - プログラム出力	438
74. filedump をコンパイルして実行するためのコマンド	360	100. PLIDUMP を呼び出す PL/I ルーチンの例	439
75. filedump の実行結果の出力	361	101. 保存されたオプション・ストリングの宣言	449
76. C qsort 関数の比較ルーチンの例	361	102. ATTENTION ON ユニットの使用	454
77. C qsort 関数を使用するためのコード例	361	103. PL/I コンパイラー・ユーザー出口のプロシージャ	460
78. qsort の誤った宣言	362	104. ユーザー出口入力ファイルの例	461
79. qsort の正しい宣言	362	105. 序数値としてエンコードされたレコード・タプル	476
80. パラメーターが BYADDR である場合のコード	363	106. レコードのヘッダー部分の宣言	476
81. パラメーターが BYVALUE である場合のコード	364	107. サマリー・レコードの宣言	477
82. Java サンプル・プログラム #2 - ストリングの引き渡し	373	108. カウンター・レコードの宣言	477
83. PL/I サンプル・プログラム #2 - ストリングの引き渡し	375	109. リテラル・レコードの宣言	478
84. Java サンプル・プログラム #3 - 整数の引き渡し	378	110. ファイル・レコードの宣言	478
85. PL/I サンプル・プログラム #3 - 整数の引き渡し	380	111. メッセージ・レコードの宣言	479
86. Java サンプル・プログラム #4 - ストリングの受け取りおよび出力	381	112. 序数タイプ・レコードの宣言	480
87. PL/I サンプル・プログラム #4 - Java 呼び出し API の呼び出し	384	113. 序数エレメント・レコードの宣言	481
88. サンプル XML 文書	391	114. 構造体のエレメントに割り当てられたシンボル索引	482
89. PLISAXA のコーディング例 - 型宣言	400	115. 変数のデータ・タイプ	484
90. PLISAXA のコーディング例 - イベント構造体	401	116. シンボル・レコードの宣言	485
91. PLISAXA のコーディング例 - メインルーチン	402	117. xin_Bif_Kind の宣言	492
92. PLISAXA のコーディング例 - イベント・ルーチン	403	118. ソース・レコードの宣言	497
93. PLISAXA のコーディング例 - プログラム出力	411	119. トークン・レコードの宣言	498
94. サンプル XML 文書	421	120. トークン・レコードの種類の宣言	498
95. PLISAXC のコーディング例 - 型宣言	429	121. プログラムのブロックに割り当てられるノード索引	499
96. PLISAXC のコーディング例 - イベント構造体	430	122. 構文レコードの宣言	500
97. PLISAXC のコーディング例 - メインルーチン	431	123. 構文レコードの種類の宣言	503
		124. プログラムの構文レコードに割り当てられるノード索引	504
		125. 式の種類の宣言	505
		126. 数値の種類の宣言	506
		127. 字句の種類の宣言	507
		128. 語彙の種類の宣言	508

概要

本書について	xiii	V3R5 からの機能拡張	xxiii
Enterprise PL/I for z/OS のランタイム環境	xiii	デバッグの向上	xxiii
資料の使用	xiv	パフォーマンス向上	xxiii
PL/I 情報	xiv	使いやすさの向上	xxiii
言語環境プログラム情報	xiv	V3R4 からの機能拡張	xxiv
本書で用いられる表記規則	xv	マイグレーションの強化	xxiv
使用する規則	xv	パフォーマンス向上	xxiv
構文記法の読み方	xv	使いやすさの向上	xxv
表記記号の読み方	xviii	デバッグの向上	xxv
表記例	xviii	V3R3 からの機能拡張	xxvi
本リリースにおける機能強化	xix	XML サポートの強化	xxvi
パフォーマンス向上	xix	パフォーマンスの改善	xxvi
使いやすさの向上	xix	容易なマイグレーション	xxvi
V3R7 からの機能拡張	xx	使いやすさの向上	xxvii
デバッグの向上	xx	デバッグ・サポートの向上	xxvii
パフォーマンス向上	xx	V3R2 からの機能拡張	xxvii
使いやすさの向上	xx	パフォーマンスの改善	xxvii
V3R6 からの機能拡張	xxi	容易なマイグレーション	xxviii
DB2 V9 サポート	xxi	使いやすさの向上	xxix
デバッグの向上	xxii	V3R1 からの機能拡張	xxix
パフォーマンス向上	xxii	VisualAge PL/I からの機能拡張	xxx
使いやすさの向上	xxii		

本書について

本書は、PL/I プログラマーおよびシステム・プログラマーを対象とした資料です。PL/I プログラムをコンパイルするための Enterprise PL/I for z/OS の使用方法を理解するのに役立ちます。また本書は、プログラム・パフォーマンスを最適化し、エラーに対する処理を行うのに必要となるオペレーティング・システムの各種機能についても説明しています。

重要: 本書では、Enterprise PL/I for z/OS を Enterprise PL/I と呼びます。

Enterprise PL/I for z/OS のランタイム環境

Enterprise PL/I は Language Environment® をそのランタイム環境として使用します。これは 言語環境プログラム体系に適合し、ランタイム環境を他の言語環境プログラム適合言語と共用することができます。

言語環境プログラムはランタイム・オプションおよび呼び出し可能サービスの共通セットを提供します。また、各 ILC 呼び出しの言語固有の初期化および終了をなくすことによって、高水準言語 (HLL) とアセンブラーの間の言語間通信 (ILC) も改善しています。

資料の使用

Enterprise PL/I の資料は、PL/I のプログラミングを行う上で役立つように設計されています。言語環境プログラムの資料は、Enterprise PL/I で生成されたアプリケーションのランタイム環境を管理する上で役立つように設計されています。それぞれの資料がさまざまな作業に役立ちます。

下の表には、Enterprise PL/I および言語環境プログラムの資料の使用方法を示しています。使用するコンパイラーおよびランタイム環境についての情報を知ることができます。これらの資料および関連資料の正式名称および資料番号については、525 ページの『参考文献』を参照してください。

PL/I 情報

表 1. Enterprise PL/I 付属資料の使用方法

作業...	使用する資料...
Enterprise PL/I の評価	Fact Sheet
保証情報の理解	Licensed Programming Specifications
Enterprise PL/I の計画とインストール	Enterprise PL/I プログラム・ディレクトリー
コンパイラーおよびランタイム変更作業の理解と、プログラムの Enterprise PL/I および 言語環境プログラムへの適合	コンパイラーおよびランタイム 移行ガイド
プログラムの準備とテスト、およびコンパイラー・オプションについての詳細情報の入手	プログラミング・ガイド
PL/I の構文および言語エレメントの仕様についての詳細情報の入手	言語解説書
コンパイラーの問題診断および IBM への連絡	診断ガイド
コンパイル時メッセージについての詳細情報の入手	メッセージおよびコード

言語環境プログラム情報

表 2. z/OS 言語環境プログラム 付属資料の使用方法

作業...	使用する資料...
言語環境プログラムの評価	概念
言語環境プログラムの計画	概念 ランタイム・アプリケーション マイグレーション・ガイド
z/OS への言語環境プログラムのインストール	z/OS Program Directory
z/OS での言語環境プログラムのカスタマイズ	カスタマイズ
言語環境プログラムのプログラム・モデルおよび概念の理解	概念 プログラミング・ガイド
言語環境プログラムランタイム・オプションおよび呼び出し可能サービスの構文の検索	プログラミング・リファレンス
言語環境プログラムで実行されるアプリケーションの開発	プログラミング・ガイドおよび該当する言語のプログラミング・ガイド
言語環境プログラムで実行されるアプリケーションのデバッグ、ランタイム・メッセージに関する詳細情報の入手、言語環境プログラムの問題の診断	デバッグのガイドとランタイム・メッセージ
言語間通信 (ILC) アプリケーションの開発	ILC アプリケーションの作成

表 2. z/OS 言語環境プログラム 付属資料の使用方法 (続き)

作業...	使用する資料...
言語環境プログラムへのアプリケーションの移行	「ランタイム・アプリケーション マイグレーション・ガイド」、および言語環境プログラムで利用できる各言語の移行ガイド

本書で用いられる表記規則

本書では、『使用する規則』および xviii ページの『表記記号の読み方』に示す規則、構文図の書き方、および表記を用いて PL/I および非 PL/I のプログラミング構文を説明しています。

使用する規則

本書のプログラミング構文の一部では、各種のエレメントを活字フォントで区別しています。

- 大文字で (UPPERCASE のように) 示した項目は、そのとおりにタイプする必要がある重要な項目です。
- 小文字で (lowercase のように) 示した項目は、適切な名前または値に置き換えてタイプする必要があるユーザー提供変数です。変数は英字で始まり、ハイフン、数字、または下線文字 (_) を入れることができます。
- 数字 は、数字 (0 から 9 まで) に置き換えなければならないことを示します。
- *DO* グループ は、DO グループに置き換えなければならないことを示します。
- 下線付きの項目は、デフォルト・オプションです。
- 例は、上段専用文字で示されます。
- 特に指示がなければ、反復可能項目は 1 つ以上のブランクを使っておのおのを区切ります。

注: 示されている記号のうち、xviii ページの『表記記号の読み方』に説明のある純粹な表記記号以外の記号はすべて、プログラミング構文そのものの一部です。

これらの規則を用いたプログラミング構文の一例が、xviii ページの『表記例』にあります。

構文記法の読み方

本書の構文図には、次の規則が適用されます。

矢印記号

構文図は、左から右、上から下へと線をたどって読んでください。

- ▶— ステートメントはここから始まります。
- ステートメントの構文は次の行へ続きます。
- ▶— ステートメントは前の行から続いています。
- ▶ ステートメントはここで終わります。

完結したステートメント以外の構文単位の図は、▶— 記号で始まり、→▶ 記号で終わります。

規則

- キーワード、許容される同義語、および予約パラメーターは、MVS および OS/2® プラットフォームでは大文字で示され、UNIX® プラットフォームでは小文字で示されます。これらの項目は示されたとおりに入力する必要があります。
- 変数は、小文字のイタリック体で示します (例えば、*column-name*)。これらはユーザー定義のパラメーターまたはサブオプションを表します。
- コマンドの入力において、パラメーターおよびキーワードを区切る句読記号がない場合は、少なくとも 1 つのブランクで区切る必要があります。
- 句読記号 (スラッシュ、コンマ、ピリオド、括弧、引用符、等号) と数字は、示されたとおりに入力する必要があります。
- 脚注は、(1) のように番号を括弧に入れて示します。
- b 記号は、1 つのブランク位置を示します。

必須項目

必須項目は横線 (メインパス) に示します。

▶▶—REQUIRED_ITEM (必須項目)————▶▶

オプション項目

オプション項目は、メインパスの下に示します。

▶▶—REQUIRED_ITEM (必須項目)——┐————▶▶
 └optional_item (オプション項目)┘

メインパスより上にオプション項目を示すこともあります。これは読みやすくするためで、ステートメントの実行には影響を及ぼしません。

 ┐————▶▶
▶▶—REQUIRED_ITEM (必須項目)——└optional_item (オプション項目)┘

複数の必須項目またはオプション項目

複数の項目から選択する場合には、それらの項目が縦に重なって、スタックを形成しています。複数の項目からいずれか 1 つを選択しなければならない場合は、スタック上の項目のうち、1 つがメインパス上に置かれます。

▶▶—REQUIRED_ITEM (必須項目)——┐————▶▶
 └required_choice1 (必須項目 1)┘
 └required_choice2 (必須項目 2)┘

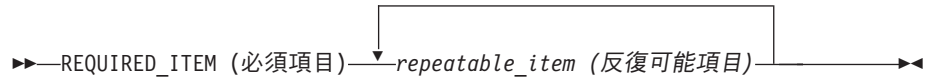
項目がオプションである場合、メインパスの下にある支線上に縦に並んだ項目として示されます。

▶▶—REQUIRED_ITEM (必須項目)————▶▶

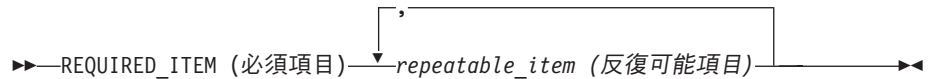
┐————▶▶
└optional_choice1 (オプション項目 1)┘
└optional_choice2 (オプション項目 2)┘

反復可能項目

メインパスの上方を通して左側へ戻る矢印は、項目が反復可能であることを示します。



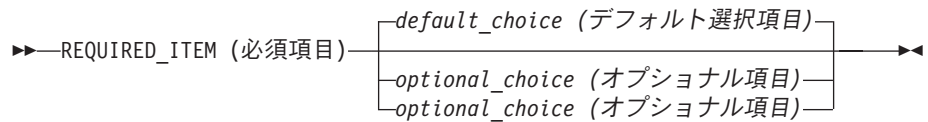
反復矢印の途中にコンマがある場合は、反復される項目をコンマで区切らなければなりません。



スタックの上の繰り返しを示す矢印は、スタックから複数の選択項目を指定できることを示しています。

デフォルト・キーワード

IBM 提供のデフォルト・キーワードはメインパスより上に示され、それ以外の選択項目はメインパスより下に示されます。構文図の下にあるパラメーター・リストでは、デフォルト選択項目に下線を付けてあります。

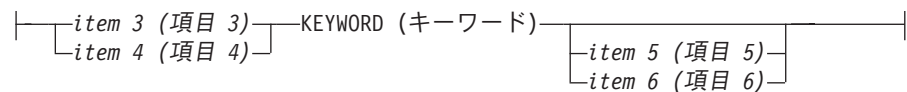


フラグメント

構文図は、フラグメント (部分) に分割する必要がある場合があります。フラグメントは文字またはフラグメント名を用いて | A | のように表します。フラグメントは主図のあとに置かれます。次の例は、フラグメントの使い方を示したものです。

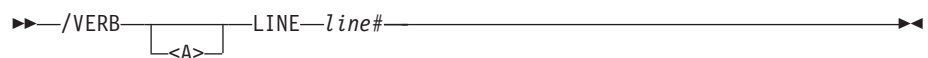


A:



置換ブロック

いくつかのパラメーターの集合を <A> のような置換ブロックで表すことができます。例えば、/VERB という仮のコマンドで、/VERB LINE 1、/VERB EITHER LINE 1、または /VERB OR LINE 1 と入力することができます。



ここで、<A> は次のようになります。



パラメーターの終わり

数値を持つパラメーターは記号 '#' で終わり、名前であるパラメーターは 'name' で終わり、汎用とすることができるパラメーターは '*' で終わります。



この例の MSNAME キーワードは名前の値をサポートし、SYSID キーワードは数値をサポートします。

表記記号の読み方

本書のプログラミング構文の一部は、表記記号を用いて表されています。同一構文の他の IBM 資料における記述法に合わせるため、またはテーブルまたは見出し内で 1 行に構文を入れられるようにするためです。

- **中括弧 { }** は、選択項目を示します。項目の 1 つに下線が付いている (デフォルトを示す) 場合、または項目がすべて大括弧で囲まれている場合を除き、少なくとも 1 つの項目を選択する必要があります。
- 単一の**縦線 |** で区切られた項目は、代替項目です。単一の縦線で区切られた項目 (または項目のグループ) では、その中の 1 つしか選択できません。(2 重縦線 || は、代替項目ではなく、連結演算を指定します。2 重縦線の詳細については、「PL/I 言語解説書」を参照してください。)
- **大括弧 []** に囲まれたものはすべてオプションです。項目が縦に積み重ねられて大括弧で囲まれていれば、1 つの項目しか指定することはできません。
- **省略符号 ...** は、直前の項目と同じタイプの項目を複数回指定できることを示します。

表記例

次の PL/I 構文は、『表記記号の読み方』の表記記号の実例です。

```
DCL file-reference FILE STREAM
      {INPUT | OUTPUT [PRINT]}
      ENVIRONMENT(option ...);
```

この例は、次のように解釈します。

- 最初の行は、*file-reference* を除き、このとおりのスペルで入力しなければなりません。*file-reference* は、参照するファイルの名前に置き換えます。
- 2 行目では INPUT または OUTPUT を指定することができますが、両方を指定することはできません。OUTPUT を指定した場合は、任意で PRINT も指定できます。どちらも選択しなかった場合は、デフォルトにより INPUT が採用されます。

- 1 つ以上のブランクでおのおのを区切り 1 つ以上のオプションに置き換えなければならない *option ...* を除き、最後の行は示されたとおりのスペル (括弧とセミicolonも含む) で入力しなければなりません。

本リリースにおける機能強化

本リリースでは、本書と他の IBM PL/I ブックに説明されている、次の機能強化を提供します。

パフォーマンス向上

- ARCH(8) および TUNE(8) オプションを指定すると、z/HE 命令が使用されます。
- HGPR オプションによって、32 ビット・コードでの 64 ビット・レジスターの使用がサポートされます。
- GOFF オプションによって、GOFF オブジェクトの生成がサポートされます。
- PFPO 命令は、異なる浮動フォーマットの変換で使用されます。
- SRSTU は、UTF-16 INDEX で生成されたコードで使用されます。
- MVCLE は、LEFT 組み込み関数で生成されたコードで使用されます。
- ヌルの内部プロシーチャーの呼び出しは、完全に除去されるようになりました。

使いやすさの向上

- PLISAXC によって、SAX インターフェースを使用して XML System Services パーサーにアクセスできます。
- INCDIR オプションは、バッチでサポートされます。
- LISTVIEW オプションによって、TEST の AFTERMACRO などのサブオプションで以前は提供されていたサポートが提供されるようになりました。
- RULES オプションの NOLAXENTRY サブオプションによって、非プロトタイプ ENTRY のフラグを立てることができます。
- DECIMAL オプションの (NO)FOFLONMULT サブオプションによって、FIXED DECIMAL の MULTIPLY で FOFL を発生させるかどうかを制御できます。
- USAGE オプションの HEX および SUBSTR サブオプションによって、ユーザーは、対応する組み込み関数の振る舞いをさらに制御できます。
- DDSQL コンパイラー・オプションによって、EXEC SQL INCLUDE で使用される代替 DD 名を指定できます。
- MACRO および SQL プリプロセッサによって提供される INCONLY サブオプションを使用して、当該プリプロセッサが INCLUDE のみを実行するように要求できます。
- LOB(DB2) SQL プリプロセッサ・オプションが選択されている場合に、統合された SQL プリプロセッサでは、既にサポート済みの BLOB、CLOB、および DBCLOB SQL 型以外にも、すべての *LOB_FILE、*LOCATOR、ROWID、BINARY、および VARBINARY SQL 型について、DB2 プリコンパイラー・スタイル宣言が生成されるようになりました。

V3R7 からの機能拡張

本リリースでは、本書と他の IBM PL/I ブックに説明されている、次の機能強化を提供します。

デバッグの向上

- TEST オプションが機能拡張され、ユーザーが指定したプリプロセッサが実行された後 (または、すべてのプリプロセッサが実行された後) にソースが表示されるのと同様に、ソースをリストおよび「デバッグ・ツール・ソース (Debug Tool source)」ウィンドウに表示することを選択できます。

パフォーマンス向上

- BASR 命令が、BALR 命令の代わりに使用されるようになりました。
- 仲介として FIXED BIN(63) を使用することにより、精度の高い FIXED DEC から FLOAT への変換が、インライン化され、高速化されました。
- CHAR 組み込み関数は、CHAR 式に適用される際には常にインライン化されるようになりました。
- FIXED BIN(p,q) からスケールのない FIXED DEC への変換用の生成コードが大幅に改良されました。
- ARCH(7) のもとで TRTR は、TRT が SEARCH および VERIFY に対して使用されたのと同じ状態で、SEARCHR および VERIFYR に対して使用されます。
- UNPKU は、一部の PICTURE を WIDECAR に変換する (ライブラリー呼び出しをするのではなく) ために使用されます。

使いやすさの向上

- IEEE 10 進浮動小数点 (DFP) がサポートされます。
- 新規の MEMCONVERT 組み込み関数を使用すると、任意のコード・ページ間で任意の長さのデータを変換できます。
- 新規の ONOFFSET 組み込み関数を使用すると、以前は実行時のエラー・メッセージまたはダンプ、つまりある条件が発生したユーザー・プロシーチャーのオフセットでのみ使用可能だった他の情報に簡単にアクセスすることができます。
- 新規の STACKADDR 組み込み関数は、現在の動的保存域 (z/OS 上のレジスター 13) のアドレスを戻し、ユーザー独自の診断コードの作成を容易にします。
- アセンブラー・リスト内の簡略記号フィールドの長さが拡大され、長い簡略記号を持つ新規 z/OS 命令のサポートが改善されました。
- 属性、相互参照、およびメッセージ・リストで、使用可能な右マージンが拡張されました。
- CODEPAGE オプションが、1026 (トルコ語コード・ページ) および 1155 (1026 コード・ページに加え、ユーロ記号) に対応できるようになりました。
- 新規の MAXNEST オプションを使用すると、BEGIN、DO、IF、および PROC ステートメントの過度なネストにフラグを立てることができます。

- RULES オプションの新規 (かつ、デフォルトでない) のサブオプション NOELSEIF を指定すると、直後に IF ステートメントが続く ELSE ステートメントにコンパイラーがフラグを立て、SELECT ステートメントとして書き直すように提案します。
- RULES オプションの新規 (かつ、デフォルトでない) のサブオプション NOLAXSTG を指定すると、コンパイラーは、変数 A が ADDR(B) および STG(A) > STG(B) の BASED として宣言されている場所にフラグを立てます。以前のように、B が定数の存在期間を指定した AUTOMATIC、BASED、または STATIC である場合のみでなく、B が定数の存在期間を指定して宣言されるパラメーターである場合にもフラグが立てられます。
- 新規の QUOTE オプションを使用すると、引用符 (") 記号に大体コード・ポイントを指定することができます。これは、この記号がコード・ページ・インバリエントではないためです。
- 新規の XML コンパイラー・オプションを使用すると、XMLCHAR 組み込み関数の出力内のタグを、すべて大文字にするか、大/小文字を宣言で使用したとおりに指定できます。
- メッセージを生成しないコンパイルであっても、コンパイラー・メッセージがリストされるはずの行に、「コンパイラー・メッセージはありません (no compiler messages)」というメッセージが表示されるようになりました。
- MACRO プリプロセッサは、%INCLUDE ステートメントのみを処理するのか、すべてのマクロ・ステートメントを処理する必要があるかを指定することのできる新規サブオプションをサポートします。
- LOB(DB2) SQL プリプロセッサ・オプションが選択されている場合に、統合された SQL プリプロセッサでは、既にサポート済みの BLOB、CLOB、および DBCLOB SQL 型以外にも、すべての *LOB_FILE、*LOCATOR、ROWID、BINARY、および VARBINARY SQL 型について、DB2 プリコンパイラー・スタイル宣言が生成されるようになりました。

V3R6 からの機能拡張

本リリースでは、本書と他の IBM PL/I ブックに説明されている、次の機能強化を提供します。

DB2 V9 サポート

- STDSQL(YES/NO) のサポート
- CREATE TRIGGER (複数 SQL ステートメント) のサポート
- FETCH CONTINUE のサポート
- SQL ステートメントに組み込まれた SQL スタイルのコメント ('--') のサポート
- 以下の追加 SQL TYPE のサポート
 - SQL TYPE IS BLOB_FILE
 - SQL TYPE IS CLOB_FILE
 - SQL TYPE IS DBCLOB_FILE
 - SQL TYPE IS XML AS
 - SQL TYPE IS BIGINT

- SQL TYPE IS BINARY
- SQL TYPE IS VARBINARY
- SQL プリプロセッサが DB2 コプロセッサ・オプションもリストするようになりました。

デバッグの向上

- TEST(NOSEpname) を指定すると、デバッグ・サイドのファイルの名前がオブジェクト・デックに保存されません。

パフォーマンス向上

- ARCH(7) での z/OS 拡張即値機能のサポート
- ARCH(6) での CLCLU、MVCLU、PKA、TP、および UNPKA 命令の活用
- ARCH(5) での CVBG および CVDG 命令の活用
- CLCLE および MVCLE の拡張使用
- DB2 日時パターンに関する変換をインライン化
- ALLOCATION 組み込み関数をインライン化
- 浮動 \$ を含んだ変換をインライン化
- PIC'(n)Z' への割り当てから条件付きコードを除去
- FIXED BIN からスケール因数を指定する PICTURE への変換をインライン化
- 次元を継承したが 8 で割り切れるストライドを持つ BIT 変数への割り当てをインライン化

使いやすさの向上

- ブロックが使用する、AUTOMATIC ストレージのストレージ・オフセット（ブロックごと）順のリストも、MAP 出力に含まれるようになりました。
- 規格合致検査が拡張されて構造体が含まられました。
- リストには、ファイル内の行番号用に 7 カラムが含まれるようになります。
- z/OS 環境では THREADID 組み込み関数がサポートされるようになりました。
- PICSPEC 組み込み関数がサポートされるようになりました。
- 新規 CEESTART オプションにより、オブジェクト・デックの始めまたは終わりに CEESTART csect を配置できるようになりました。
- 新規 PPCICS、PPMACRO、および PPSQL オプションにより、対応するプリプロセッサで使われるデフォルト・オプションを指定できるようになりました。
- ATTRIBUTES リストに ENVIRONMENT オプションが組み込まれるようになりました。
- DISPLAY オプションがサポートする新規サブオプションによって、REPLY を指定した DISPLAY か REPLY を指定しない DISPLAY が、さまざまな DESC コードで可能になりました。
- コメント内のセミコロンに対するフラグ・メッセージに、セミコロンのある行の行番号が組み込まれるようになりました。
- 使用されない INCLUDE ファイルに対してフラグが立ちます。
- REFER 項目が変更される可能性がある割り当てに対してフラグが立ちます。

- KEY/KEYFROM 文節が含まれない KEYED DIRECT ファイルの使用に対してフラグが立ちます。
- PICTURE をループ制御変数として使用することに対してフラグが立ちます。

V3R5 からの機能拡張

本リリースでは、本書と他の IBM PL/I ブックに説明されている、次の機能強化を提供します。

デバッグの向上

- TEST(SEPARATE) を指定すると、デバッグ情報の大部分が別個のデバッグ・ファイルに書き込まれます。
- AUTOMONITOR に割り当てのターゲットが含まれます。
- AUTOMATIC の初期化後に AT ENTRY フックが配置されるようになりました。それによって、変数を調べる前のブロックへのステップイントゥが必要なくなりました。

パフォーマンス向上

- 分岐相対命令の生成により、基底レジスターと制御権移動ベクトルの必要性が大幅に削減されます。
- ARCH(6) での z/OS 長変位機能のサポート。
- REFER を使用する単純構造体が、ライブラリー呼び出しを介さずにインラインでマップされます。
- REFER を使用する構造体のうち、これまでのようにライブラリー呼び出しを介してマップされるものについては、REFER が副構造体の配列の境界を指定していれば、生成されるコードが少なくなります。
- 重複 INCLUDE が高速で処理されるようになりました。
- 最終位置が I または R の PICTURE 変数への変換が、インライン化されました(最終文字が T の PICTURE 変数への変換は既にインライン化されていました)。
- B を含まないピクチャーがインライン化された場合、そのピクチャーに対応する、1 つ以上の B で終わる PICTURE 変数への変換がインライン化されるようになりました。
- CHARACTER 変数から X のみで構成される PICTURE 変数への変換が、インライン化されるようになりました。

使いやすさの向上

- リストなどのすべての部分で、ソース・ファイルがファイル 0、最初のインクルード・ファイルがファイル 1、2 番目の (固有の) インクルード・ファイルがファイル 2 のように数えられます。
- 規格合致検査が拡張されて配列が含められました。
- 呼び出されたプリプロセッサのビルドの日付がリストに組み込まれます。
- COBOL との ILC を簡単にするために、1 バイト FIXED BINARY 引数を抑止できます。
- SYSADATA、SYSXMLSD、および SYSDEBUG の代替 DD 名を指定できます。

- XNUMERIC を指定した場合、RULES(NOLAXMARGINS) はシーケンス番号を許容します。
- RULES(NOUNREF) は、参照されない AUTOMATIC 変数に対してフラグを立てます。
- 変数への割り当てがライブラリー呼び出しを介して行われる場合、ライブラリー呼び出しに対するフラグ・メッセージには、ターゲット変数の名前が含まれます。
- 一回限りの DO ループに対してフラグが立ちます。
- 引数として使用されるラベルに対してフラグが立ちます。
- PRV が使用される場合、FETCHABLE の PARAMETER CONTROLLED 以外の ALLOCATE と FREE に対してフラグが立ちます。
- DEFINED および BASED がそれぞれの基数より大きい場合、基数が後で宣言されるとしても、それぞれに対してフラグが立ちます。
- FIXED DEC から 8 バイト整数への暗黙的型変換に対してフラグが立ちます。

V3R4 からの機能拡張

本リリースでは、本書と他の IBM PL/I ブックに説明されている、次の機能強化を提供します。

マイグレーションの強化

- 旧コードとの CONTROLLED の共用のサポート
- デフォルト初期化の向上
- ADD、DIVIDE、および MULTIPLY での小数部指定の容易化
- GRAPHIC の STRING に対する旧セマンティクスのサポート
- DEFAULT ステートメントに対する旧セマンティクスのサポート
- ストレージ・オーバーレイのある宣言に対するフラグ付け
- BEGIN 内の RETURN での制限の撤廃
- コメント内のセミコロンに対するフラグ付け (オプション)
- アセンブラーで初期化される EXT STATIC のサポート
- 無効な紙送り制御文字に対するフラグ付け
- 言語の誤用 (特に RETURN) に対するフラグ付けの強化
- REPLACEBY2 組み込み関数のサポート
- SIZE を発生させる 10 進代入における FOFL の抑止 (オプション)
- 言語で旧コンパイラとは異なる処理をしたことに対するフラグ付けの強化

パフォーマンス向上

- INDEX および TRANSLATE でのコード生成の向上
- ピクチャーに対する代入のインライン化の向上
- ソースが FIXED DEC であった場合に、変換がインラインで行われるときの PICTURE に対する CHARACTER の変換で生成されるコードの向上
- パック 10 進数変換で生成されるコードの向上

- REFER の一部の使用法で生成されるコードの向上
- 長さが不明な文字ストリングの比較のインライン化
- 連結に使用されるスタック・ストレージの量の削減
- GET/PUT STRING EDIT ステートメントのインライン化の向上
- LE 条件処理の短縮の強化
- BIN FIXED の OR および AND のインライン化の向上
- ALIGNED BIT(8) に対する SIGNED FIXED BIN(8) のインライン化
- コンパイラーが実行時に構造にマップするためのライブラリー・ルーチン呼び出しを生成するステートメントに対するフラグ付け
- リスト生成に使用される I/O の量の削減

使いやすさの向上

- 16 進での AGGREGATE リストにおけるオフセットの提供 (オプション)
- LIMITS(FIXEDDEC(15,31)) オプションによる、必要な場合のみの DEC(31) のサポート
- オプション内でのコメントの許可
- 偶数精度での FIXED DEC 宣言に対するフラグ付け (オプション)
- SIZE を発生させる可能性のある DEC 代入に対する DEC のフラグ付け
- SIZE を発生させる可能性のある PIC 代入に対する DEC/PIC のフラグ付け
- NOINIT 属性による、INIT なしの LIKE のサポート
- z/OS UNIX での PDS からの組み込みの容易化
- MACRO プリプロセッサ内での LOWERCASE、MACNAME、TRIM および LOWERCASE 組み込み関数のサポート
- PTF による、オプションの紹介の容易化
- *PROCESS の使用の不許可 (オプション)
- MDECK 内での *PROCESS の保持 (オプション)
- 一回限りの INCLUDE のサポート
- マクロで決定する INCLUDE 名のサポート
- 実行時のストリング・パラメーター検査のサポート
- 未初期化変数である可能性があることに対するフラグ付けの強化
- コーディング・エラーの可能性があり、異常な比較に対するフラグ付け
- STORAGE オプションの出力のフォーマットが使いやすくなり、LIST オプションの出力には、コンパイル単位から各ブロックまでの 16 進オフセットが組み込まれるようになりました。

デバッグの向上

- オーバーレイ・フックに対するより良いサポート
- LE ダンプ内の CONTROLLED 変数の解決の容易性の強化
- リストへのユーザー指定のオプションの常時組み込み

V3R3 からの機能拡張

このリリースには、以下を含む Enterprise PL/I V3R3 で拡張された機能がすべて備わっています。

XML サポートの強化

XMLCHAR 組み込み関数が、参照される構造体のエレメントの名前と値で XML をバッファーに書き込み、書き込まれたバイト数を戻します。次にこの XML は、PL/I SAX パーサーを使用したコードとともに他のアプリケーションに渡され、実行されます。

パフォーマンスの改善

- OPT(2) でのコンパイル時間は、Enterprise PL/I V3R2 の場合よりも、特に大規模なプログラムで大幅に少なくなります。
- コンパイラーは、ED および EDMK 命令を使用して、PICTURE および CHARACTER へのインライン化された数値変換を行います。これにより、より早く、短いコード・シーケンスおよびコンパイルの高速化が実現しました。
- コンパイラーは、ストリング比較をさらに効率よく行うコードを生成するようになりました。このことも、より早く、短いコード・シーケンスという結果をもたらしています。
- コンパイラーは、より短く高速なコードを生成し、FIXED DECIMAL から、末尾に overpunch という文字のついた PICTURE への変換を行います。
- ARCH および TUNE コンパイラー・オプションは、有効なサブオプションとして 5 つを受け入れます。ARCH(5) のもとでは、コンパイラーが適切なときに、NILI、NILH、OILL、OILH、LLILL、および LLILH などの新規 z/Architecture 命令を生成します。

容易なマイグレーション

- 新規の BIFPREC コンパイラー・オプションは、さまざまな組み込み関数によって戻された FIXED BIN の結果の精度を制御し、これにより OS PL/I コンパイラーとのよりよい互換性を提供します。
- 新規の BACKREG コンパイラー・オプションは、コンパイラーが、逆チェーン・レジスターとしてどのレジスターを使用するかを制御し、これにより、古いオブジェクト・コードと新しいオブジェクト・コードの混合を容易にします。
- 新規の RESEXP コンパイラー・オプションは、コードの中の制限つき式の評価を制御し、OS PL/I コンパイラーとのよりよい互換性を提供します。
- 新規の BLKOFF コンパイラー・オプションは、コンパイラーの疑似アセンブラー・リストにおけるオフセットの計算方法を制御します。
- STORAGE コンパイラー・オプションは、それぞれのプロシージャーおよび開始ブロックによって使用されるストレージの要約をコンパイラーに作成させます。これは、リストの一部として作成され、OS PL/I コンパイラーで作成されたものと似ています。

使いやすさの向上

- RULES コンパイラー・オプションの新規の LAXDEF サブオプションにより、いわゆる無許可定義を使用できるようになり、この際、コンパイラーが E レベル・メッセージを生成することはありません。
- 新規の FLOATINMATH コンパイラー・オプションにより、数学関数の評価に関する精度の制御が容易になりました。
- 新規の MEMINDEX、MEMSEARCH(R) および MEMVERIFY(R) 組み込み関数により、32K より大きいストリングの検索が可能です。
- DISPLAY(WTO) コンパイラー・オプションの、新規の ROUTCDE および DESC サブオプションは、対応する WTO の要素の制御を提供します。
- コンパイラーは、それぞれのオブジェクトの中に短ストリングを保管し、それが関連するコードが実行されている間もストレージにあり、そのオブジェクトを作成するために使用されるオプションをすべて記録します。これにより、さまざまなツールによるより良い診断が可能になりました。
- コンパイラーは、ステートメントがマージまたは削除された場所をさらに識別するメッセージを発行します。
- PLIDUMP 出力は、静的ユーザーの 16 進ダンプをインクルードするようになりました。
- PLIDUMP 出力は、言語環境プログラムのトレースバックで、それぞれのプログラムをコンパイルするために使用するオプションをインクルードするようになりました。
- PLIDUMP 出力は、PL/I ファイルに関する情報をインクルードするようになりました。

デバッグ・サポートの向上

- REFER を使用した BASED 構造体は、DebugTool およびデータ指示 I/O ステートメントでサポートされるようになりました (制限は他のすべての BASED 変数上と同じです)。
- 他の構造体のスカラー・メンバーから (順番に BASED されるなどで) BASED された BASED 構造体は、DebugTool およびデータ指示 I/O ステートメントでサポートされるようになりました (制限は他のすべての BASED 変数上と同じです)。

V3R2 からの機能拡張

本リリースでは、次を含む Enterprise PL/I V3R2 での機能強化もすべて提供されます。

パフォーマンスの改善

- このコンパイラーでは、インライン・コードを生成してより多くの型変換を処理できるようになりました。これにより、型変換が以前に比べ、格段に早く行われるようになります。また、ライブラリー呼び出しによって行われるすべての型変換は、コンパイラーによってフラグが立てられるようになりました。
- コンパイラー生成コードが、さまざまな状況において使用するスタック・ストレージの容量が減少しました。

- コンパイラーが、TRANSLATE 組み込み関数を参照するために生成するコードが改善されました。
- SUBSCRIPTRANGE 検査用のコンパイラー生成コードは、既知の境界を持つ配列の場合は、処理速度が以前の倍になりました。
- ARCH と TUNE オプションは、サブオプション 4 をサポートするようになり、zSeries マシンで新たに命令の開発が行えるようになりました。
- ARCH(2)、FLOAT(AFP) および TUNE(3) がデフォルトになりました。

容易なマイグレーション

- マイグレーションを容易にし、互換性を持たせるために、コンパイラーのデフォルト値が変更されました。変更されたデフォルト値は、次のとおりです。
 - CSECT
 - CMPAT(V2)
 - LIMITS(EXTNAME(7))
 - NORENT
- コンパイラーは、OPTIONS(COBOL) を指定した PROC および ENTRY で、NOMAP、NOMAPIN および NOMAP 属性を指定できるようになりました。
- コンパイラーは、複数の ENTRY ステートメントを指定した PROC をサポートするようになりました。この ENTRY ステートメントは、前リリースのホスト・コンパイラーと同様、それぞれが異なる RETURNS 属性を保持することができます。
- コンパイラーは、OPTIONS(RETCODE) において、PROC と ENTRY には OPTIONS(COBOL) が指定されていることを想定しています。
- 未処理の場合、SIZE 条件は ERROR にプロモートされません。
- コンパイル時間とストレージ要件を減らすために、さまざまな変更が行われました。
- OFFSET オプションは、前リリースの PL/I コンパイラーで生成されたものとよく似たステートメント・オフセット・テーブルを生成します。
- FLAG オプションは前リリースのコンパイラーの場合と意味はまったく同じですが、新規の MAXMSG オプションは、指定された重大度において、メッセージが指定した回数発生した後にコンパイラーを終了すべきかどうかを、ユーザーが決定できるようになりました。例えば、FLAG(I) MAXMSG(E,10) を指定すると、I レベル・メッセージはすべて確認し、E レベル・メッセージは 10 回発生したらコンパイルを終了するように指定できます。
- AGGREGATE リストには、調節可能エクステンントを備えた構造を組み込めるようになりました。
- STMT オプションは、リストのいくつかのセクションをサポートするようになりました。
- LINESIZE で使用できる最大値は、F フォーマット・ファイルでは 32759、V フォーマット・ファイルでは 32751 に変更されました。

使いやすさの向上

- コンパイラー・オプションのデフォルトは、インストール時に変更できるようになりました。
- 内蔵 SQL プリプロセッサは、DB2 Unicode をサポートするようになりました。
- コンパイラーは、デバッグ・ツールが Auto Monitor をサポートできるようにする情報を生成するようになりました。それで、各ステートメントが実行される直前に、ステートメントで使用されるすべての変数のすべての値が表示されます。
- 新規の NOWRITABLE コンパイラー・オプションを使用すると、NORENT を指定した場合に、最適なパフォーマンスを犠牲にしても、コンパイラーが FILE と CONTROLLED を操作するコードを生成する際に、書き込み可能な静的値を使用しないように指定できます。
- 新規の USAGE コンパイラー・オプションを使用すると、RULES(IBMANS) オプションの他の影響を受けずに、ROUND および UNSPEC 組み込み関数の IBM または ANS 動作を完全に制御できます。
- 新規の STDSYS コンパイラー・オプションは、コンパイラーに SYSPRINT ファイルと C stdout ファイルを同一にするように指定します。
- 新規の COMPACT コンパイラー・オプションが使用されると、コードが大きくなることを制限する最適化を使用するようにコンパイラーに指示します。
- SYSPRINT の LRECL は 137 に変更され、C/C++ コンパイラーの LRECL と一致するようになりました。
- PUT LIST と PUT EDIT ステートメントで POINTER が使用できるようになりました。8 バイトの 16 進値が出力されます。
- ABNORMAL 属性を STATIC 変数で指定すると、STATIC 変数が使用されていなくてもこの変数は保存されます。

V3R1 からの機能拡張

本リリースでは、次を含む Enterprise PL/I V3R1 での機能強化もすべて提供されます。

- z/OS でのマルチスレッド化のサポート
- z/OS での IEEE 浮動小数点のサポート
- マクロ・プリプロセッサでの ANSWER ステートメントのサポート
- PLISAXA および PLISAXB 組み込みサブルーチンを介した SAX 形式 XML 構文解析
- 追加の組み込み関数
 - CS
 - CDS
 - ISMAIN
 - LOWERCASE
 - UPPERCASE

VisualAge PL/I からの機能拡張

本リリースでは、次を含む VisualAge PL/I V2R2 での機能強化もすべて提供します。

- WIDECHAR 属性を介した初期 UTF-16 サポート

以下については、まだサポートされていません。

- ソース・ファイル内の WIDECHAR 文字
- W スtring定数
- ストリーム入出力内の WIDECHAR 式の使用
- レコード入出力での WIDECHAR からの暗黙の型変換または WIDECHAR への暗黙の型変換
- レコード入出力での暗黙の endianness フラグ

WIDECHAR ファイルを作成する場合は、ファイルの最初の 2 バイトとして endianness フラグ ('fe_ff'wx) を書き込んでください。

- DEFAULT ステートメントでサポートされる DESCRIPTORS オプションと VALUE オプション
- PUT DATA 機能強化
 - POINTER、OFFSET およびサポートされているその他の非計算変数
 - タイプ 3 DO 仕様が使用可能
 - 添え字が使用可能
- DEFINE ステートメントの機能強化
 - 指定されていない構造体の定義
 - CAST および RESPEC タイプ関数
- 追加の組み込み関数
 - CHARVAL
 - ISIGNED
 - IUNSIGNED
 - ONWCHAR
 - ONWSOURCE
 - WCHAR
 - WCHARVAL
 - WHIGH
 - WIDECHAR
 - WLOW
- プリプロセッサ機能強化
 - プリプロセッサ・プロシージャでの配列のサポート
 - %DO ステートメントでの WHILE、UNTIL および LOOP キーワードのサポート
 - %ITERATE ステートメントのサポート
 - %LEAVE ステートメントのサポート
 - %REPLACE ステートメントのサポート
 - %SELECT ステートメントのサポート
 - 追加の組み込み関数

- COLLATE
- COMMENT
- COMPILEDATE
- COMPILETIME
- COPY
- COUNTER
- DIMENSION
- HBOUND
- INDEX
- LBOUND
- LENGTH
- MACCOL
- MACLMAR
- MACRMAR
- MAX
- MIN
- PARMSET
- QUOTE
- REPEAT
- SUBSTR
- SYSPARM
- SYSTEM
- SYSVERSION
- TRANSLATE
- VERIFY

第 1 部 プログラムのコンパイル

第 1 章 コンパイラ・オプションと機能の使用 . . . 5

コンパイル時オプションの説明 5

AGGREGATE	8
ARCH	9
ATTRIBUTES	10
BACKREG	10
BIFPREC	11
BLANK	12
BLKOFF	12
CEESTART	12
CHECK	13
CMPAT	14
CODEPAGE	15
COMMON	16
COMPACT	16
COMPILE	17
COPYRIGHT	18
CSECT	18
CSECTCUT	19
CURRENCY	19
DBCS	19
DD	20
DDSQL	20
DECIMAL	20
DEFAULT	22
DISPLAY	31
DLLINIT	32
EXIT	32
EXTRN	32
FLAG	33
FLOAT	33
FLOATINMATH	36
GOFF	36
GONUMBER	37
GRAPHIC	37
HGPR	38
INCAFTER	38
INCDIR	39
INCPDS	39
INITAUTO	40
INITBASED	40
INITCTL	41
INITSTATIC	41
INSOURCE	42
INTERRUPT	42
LANGLVL	43
LIMITS	44
LINECOUNT	45
LINEDIR	45
LIST	45
LISTVIEW	46

MACRO	47
MAP	47
MARGINI	47
MARGINS	48
MAXMEM	49
MAXMSG	50
MAXNEST	50
MAXSTMT	51
MAXTEMP	51
MDECK	51
NAME	52
NAMES	52
NATLANG	52
NEST	53
NOT	53
NUMBER	53
OBJECT	54
OFFSET	54
OPTIMIZE	55
OPTIONS	56
OR	56
PP	57
PPCICS	58
PPINCLUDE	58
PPMACRO	59
PPSQL	60
PPTRACE	60
PRECTYPE	60
PREFIX	61
PROCEED	61
PROCESS	62
QUOTE	63
REDUCE	63
RENT	64
RESEXP	65
RESPECT	65
RULES	66
SEMANTIC	72
SERVICE	73
SOURCE	73
SPILL	73
STATIC	74
STD SYS	74
STMT	74
STORAGE	75
STRINGOFGRAPHIC	75
SYNTAX	75
SYSARM	76
SYSTEM	77
TERMINAL	78
TEST	78

TUNE	82
USAGE	82
WIDECCHAR	83
WINDOW	84
WRITABLE	84
XINFO	85
XML	88
XREF	88
オプションの中のブランク、コメント、およびスト リング	89
デフォルト・オプションの変更	89
%PROCESS ステートメントまたは *PROCESS ステ ートメントでのオプションの指定	90
% ステートメントの使用	91
%INCLUDE ステートメントの使用	92
%OPTION ステートメントの使用	93
コンパイラー・リストの使用	94
見出し情報	94
コンパイルに使用するオプション	95
プリプロセッサ入力	95
SOURCE プログラム	95
ステートメントのネスト・レベル	95
ATTRIBUTE と相互参照テーブル	96
属性テーブル	96
相互参照テーブル	97
集合長さテーブル	97
ステートメント・オフセット・アドレス	97
ストレージ・オフセット・リスト	100
ファイル参照テーブル	101
メッセージと戻りコード	102
第 2 章 PL/I プリプロセッサ	105
インクルード・プリプロセッサ	106
マクロ・プリプロセッサ	107
マクロ・プリプロセッサのオプション	107
マクロ・プリプロセッサの例	109
SQL プリプロセッサ	110
プログラミングとコンパイルに関する考慮事項	111
SQL プリプロセッサ・オプション	112
PL/I アプリケーション内での SQL ステートメ ントのコーディング	119
SQL 連絡域の定義	120
SQL 記述子域の定義	120
SQL ステートメントの組み込み	121
ホスト変数の使用	122
SQL および PL/I の同等なデータ・タイプの 判別	125
ラージ・オブジェクト (LOB) サポートに関する 追加情報	128
LOB に関する一般情報	128
LOB サポートのための PL/I 変数宣言	130
SQL データ・タイプと PL/I データ・タイプの 互換性の判別	131
ホスト構造体の使用	131
標識変数の使用	132
ホスト構造体の例	133

コンパイラー・ユーザー出口 (IBMUEXIT) での SQL プリプロセッサの使用	133
DECLARE STATEMENT ステートメント	134
CICS プリプロセッサ	134
プログラミングとコンパイルに関する考慮事項	135
CICS プリプロセッサのオプション	136
PL/I アプリケーション内での CICS ステートメ ントのコーディング	136
CICS ステートメントの組み込み	136
PL/I を使用した CICS トランザクションの作成	136
エラー処理	137

第 3 章 PL/I カタログ式プロシージャーの用法	139
IBM 提供のカatalog式プロシージャー	139
コンパイルのみ (IBMZC)	140
コンパイルおよびバインド (IBMZCB)	142
コンパイル、バインド、および実行 (IBMZCBG)	144
コンパイル、プリリンク、およびリンク・エディ ット (IBMZCPL)	146
コンパイル、プリリンク、リンク・エディット、 および実行 (IBMZCPLG)	148
コンパイル、プリリンク、ロード、および実行 (IBMZCPG)	150
Catalog式プロシージャーの呼び出し	152
複数呼び出しの指定	153
PL/I カタログ式プロシージャーの変更	154
EXEC ステートメント	154
DD ステートメント	155

第 4 章 プログラムのコンパイル	157
z/OS UNIX の下でのコンパイラーの呼び出し	157
入力ファイル	157
z/OS UNIX の下でのコンパイル時オプションの 指定	158
-qoption_keyword	158
単一フラグおよび複数文字フラグ	159
JCL を使用した z/OS の下でのコンパイラーの呼び 出し	160
EXEC ステートメント	160
標準データ・セット用の DD ステートメント	160
入力 (SYSIN)	161
出力 (SYSLIN, SYSPUNCH)	162
一時作業ファイル (SYSUT1)	162
リスト (SYSPRINT)	162
ソース・ステートメント・ライブラリー (SYSLIB)	163
オプションの指定	163
EXEC ステートメントでのオプションの指定	164
オプション・ファイルを使用した EXEC ステ ートメントでのオプションの指定	164

第 5 章 リンク・エディットと実行	167
リンク・エディットに関する考慮事項	167
バインダーの使用	167
プリリンカーの使用	168
ENTRY カードの使用	168
実行時の考慮事項	168

PRINT ファイルのフォーマット設定に関する規則	168
PRINT ファイルでのフォーマットの変更 . . .	169
自動プロンプト	170
自動プロンプトの指定変更	170
長い入力行の句読法	171
行継続文字	171
GET LIST ステートメントと GET DATA ステートメントの句読法	171
GET EDIT での自動埋め込み	171
ENDFILE	172
SYSPRINT の考慮事項	172
ルーチン内での FETCH の使用	174
Enterprise PL/I ルーチンのフェッチ	174
z/OS C ルーチンのフェッチ	182
アセンブラー・ルーチンのフェッチ	182
z/OS UNIX を指定した場合の MAIN の呼び出し	182

第 1 章 コンパイラー・オプションと機能の使用

この章では、コンパイラーに使用できるオプションと、その省略形および IBM 提供のデフォルトについて説明します。アプリケーションをコンパイルするときに、PL/I は言語環境プログラム実行時へのアクセスを必要とするということを忘れないようにすることは重要です。大部分のデフォルトは、PL/I プログラムのコンパイル時に指定変更することができます。コンパイラーのインストール時には、デフォルトを指定変更することもできます。

コンパイル時オプションの説明

コンパイラー・オプションには 3 つのタイプがあります。ただし、大部分のコンパイラー・オプションには肯定形式と否定形式があります。否定形式は、肯定形式の初めに 'NO' を付け加えたものです (例えば TEST および NOTEST)。オプションによっては、肯定形式しかないものもあります (例えば SYSTEM)。コンパイラー・オプションのタイプは、次の 3 つです。

1. キーワードの単純な組み合わせ: 機能を要求する肯定形式、およびその機能を禁止する代替否定形式 (例えば、NEST および NONEST)。
2. オプションを修飾する値リストを提供するためのキーワード (例えば、FLAG(W))。
3. 上記の 1 と 2 を組み合わせたもの (例えば、NOCOMPILE(E))。

表 3 は、すべてのコンパイラー・オプションの省略形 (存在する場合) を、IBM 提供のデフォルト値と共にリストします。あるオプションに省略記述できるサブオプションがある場合は、これらの省略形をオプションのフルネームを示す列に記載します。

簡便のために、テーブル内のいくつかのオプションは簡単に説明しています (例えば、LANGLVL で必要となるサブオプションは 1 つだけです。同様に、TEST でサブオプションを 1 つ指定したら、他を指定する必要はありません)。その後のページで、完全かつ正確な構文を説明しています。

表 3 の後の項で、これらのオプションをアルファベット順に説明しています。コンパイラーが情報をリストすることを指定するオプションの場合、簡単な説明しか付けられていません。生成されるリストの説明は 94 ページの『コンパイラー・リストの使用』にあります。

表 3. コンパイル時オプション、省略形、および IBM 提供のデフォルト値

コンパイル時オプション	省略名	z/OS のデフォルト値
AGGREGATE[(DEC HEX)] NOAGGREGATE	AG NAG	NOAGGREGATE
ARCH(n)	-	ARCH(5)
ATTRIBUTES[(FULL SHORT)] NOATTRIBUTES	A NA	NA [(FULL)] ¹
BACKREG(5 11)	-	BACKREG(5)
BIFPREC(15 31)	-	BIFPREC(15)
BLANK('c')	-	BLANK('t') ²
BLKOFF NOBLKOFF	-	BLKOFF

表3. コンパイル時オプション、省略形、および IBM 提供のデフォルト値 (続き)

コンパイル時オプション	省略名	z/OS のデフォルト値
CEESTART(FIRST LAST)	-	CEESTART(FIRST)
CHECK(STORAGE NOSTORAGE, CONFORMANCE NOCONFORMANCE)	-	CHECK(NSTG, NOCONFORMANCE)
CMPAT(LE V1 V2 V3)	-	CMPAT(V2)
CODEPAGE(n)	CP	CODEPAGE(1140)
COMMON NOCOMMON	-	NOCOMMON
COMPACT NOCOMPACT	-	NOCOMPACT
COMPILE NOCOMPILE[(W E S)]	C NC	NOCOMPILE(S)
COPYRIGHT('string') NOCOPYRIGHT	-	NOCOPYRIGHT
CSECT NOCSECT	CSE NOCSE	CSECT
CSECTCUT(n)	-	CSECTCUT(4)
CURRENCY('c')	CURR	CURRENCY(\$)
DBCS NODBCS	-	NODBCS
DD(ddname-list)	-	DD(SYSPRINT,SYSIN, SYSLIB,SYPUNCH, SYSLIN,SYSADATA, SYSXMLSD,SYSDEBUG)
DDSQL(ddname)	-	DDSQL('')
DECIMAL(FOFLONASGN NOFOFLONASGN, FOFLONMULT NOFOFLONMULT, FORCEDSIGN, NOFORCEDSIGN)	DEC	DEC(FOFLONASGN, NOFOFLONMULT, NOFORCEDSIGN)
DEFAULT(attribute option)	DFT	31 ページを参照。
DISPLAY(STD WTO(ROUTCDE(x) DESC(y) REPLY(z)))	-	DISPLAY(WTO)
DLLINIT NODLLINIT	-	NODLLINIT
EXIT NOEXIT	-	NOEXIT
EXTRN(FULL SHORT)	-	EXTRN(FULL)
FLAG[(I W E S)]	F	FLAG(W)
FLOAT(AFP(NOVOLATILE VOLATILE) NOAFP, DFP NODFP)	-	FLOAT(AFP(NOVOLATILE) NODFP)
FLOATINMATH(ASIS LONG EXTENDED)	-	FLOATINMATH(ASIS)
GOFF NOGOFF	-	NOGOFF
GONUMBER NOGONUMBER	GN NGN	NOGONUMBER
GRAPHIC NOGRAPHIC	GR NGR	NOGRAPHIC
HGPR[(PRESERVE NOPRESERVE)] NOHGPR	-	NOHGPR
INCAFTER([PROCESS(filename)])	-	INCAFTER()
INCDIR('directory name') NOINCDIR	-	NOINCDIR
INCPDS('PDS name') NOINCPDS	-	NOINCPDS
INITAUTO NOINITAUTO	-	NOINITAUTO
INITBASED NOINITBASED	-	NOINITBASED
INITCTL NOINITCTL	-	NOINITCTL
INITSTATIC NOINITSTATIC	-	NOINITSTATIC
INSOURCE[(FULL SHORT)] NOINSOURCE	IS NIS	NOINSOURCE
INTERRUPT NOINTERRUPT	INT NINT	NOINTERRUPT
LANGLVL(SAA SAA2[,NOEXT OS])	-	LANGLVL(SAA2,OS)
LIMITS(options)	-	44 ページを参照。
LINECOUNT(n)	LC	LINECOUNT(60)
LINEDIR NOLINEDIR	-	NOLINEDIR
LIST NOLIST	-	NOLIST
LISTVIEW(SOURCE AFTERMACRO AFTERCICS AFTERSQL AFTERALL)	-	LISTVIEW(SOURCE)
MACRO NOMACRO	M NM	NOMACRO
MAP NOMAP	-	NOMAP

表 3. コンパイル時オプション、省略形、および IBM 提供のデフォルト値 (続き)

コンパイル時オプション	省略名	z/OS のデフォルト値
MARGINI('c') NOMARGINI	MI NMI	NOMARGINI
MARGINS(m,n[,c]) NOMARGINS	MAR(m,n)	MARGINS F-format: (2,72) V-format: (10,100)
MAXMEM(n)	MAXM	MAXMEM(1048576)
MAXMSG(I W E S,n)	-	MAXMSG(W,250)
MAXNEST(BLOCK(x) DO(y) IF(z))	-	MAXNEST(BLOCK(17) DO(17) IF(17))
MAXSTMT(n)	-	MAXSTMT(4096)
MAXTEMP(n)	-	MAXTEMP(50000)
MDECK NOMDECK	MD NMD	NOMDECK
NAME(['external name']) NONAME	N	NONAME
NAMES('lower'[,upper])	-	NAMES('#@\$','#@\$\$')
NATLANG(ENU UEN)	-	NATLANG(ENU)
NEST NONEST	-	NONEST
NOT	-	NOT('~')
NUMBER NONUMBER	NUM NNUM	NUMBER
OBJECT NOOBJECT	OBJ NOBJ	OBJECT
OFFSET NOOFFSET	OF NOF	NOOFFSET
OPTIMIZE(0 2 3) NOOPTIMIZE	OPT NOPT	OPT(0)
OPTIONS[(ALLIDOC)] NOOPTIONS	OP NOP	NOOPTIONS
OR('c')	-	OR(' ')
PP(pp-name) NOPP	-	NOPP
PPCICS('string') NOPPCICS	-	NOPPCICS
PPINCLUDE('string') NOPPINCLUDE	-	NOPPINCLUDE
PPMACRO('string') NOPPMACRO	-	NOPPMACRO
PPSQL('string') NOPPSQL	-	NOPPSQL
PPTRACE NOPTRACE	-	NOPTRACE
PREFIX(condition)	-	61 ページを参照。
PRECTYPE(ANS DECRESULT)	-	PRECTYPE(ANS)
PROCEED NOPROCEED[(W E S)]	PRO NPRO	NOPROCEED(S)
PROCESS[(KEEP DELETE)] NOPROCESS	-	PROCESS(DELETE)
QUOTE('')	-	QUOTE('')
REDUCE NOREDUCE	-	REDUCE
RENT NORENT	-	NORENT
RESEXP NORESEXP	-	RESEXP
RESPECT([DATE])	-	RESPECT()
RULES(options)	-	66 ページを参照。
SEMANTIC NOSEMANTIC[(W E S)]	SEM NSEM	NOSEMANTIC (S)
SERVICE('service string') NOSERVICE	SERV NOSERV	NOSERVICE
SOURCE NOSOURCE	S NS	NOSOURCE
SPILL(n)	SP	SPILL(512)
STATIC(FULL SHORT)	-	STATIC(SHORT)
STD SYS NOSTD SYS	-	NOSTD SYS
STMT NOSTMT	-	NOSTMT
STORAGE NOSTORAGE	STG NSTG	NOSTORAGE
STRINGOFGRAPHIC(CHAR GRAPHIC)	-	STRINGOFGRAPHIC (GRAPHIC)
SYNTAX NOSYNTAX[(W E S)]	SYN NSYN	NOSYNTAX(S)
SYSPARM('string')	-	SYSPARM('')
SYSTEM(MVS CICS IMS TSO OS)	-	SYSTEM(MVS)

表 3. コンパイル時オプション、省略形、および IBM 提供のデフォルト値 (続き)

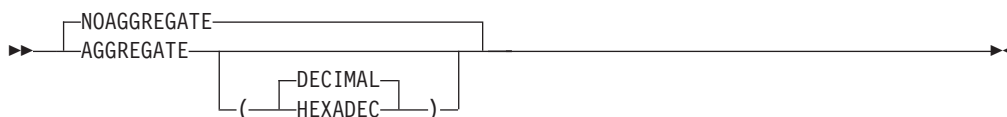
コンパイル時オプション	省略名	z/OS のデフォルト値
TERMINAL NOTERMINAL	TERM NTERM	
TEST(<i>options</i>) NOTEST	-	78 ページの『TEST』 ³ を参照
TUNE(<i>n</i>)	-	TUNE(5)
USAGE(<i>options</i>)	-	82 ページの『USAGE』 を参照
WIDECHAR(BIGENDIAN LITTLEENDIAN)	WCHAR	WIDECHAR(BIGENDIAN)
WINDOW(<i>w</i>)	-	WINDOW(1950)
WRITABLE NOWRITABLE[(FWS PRV)]	-	WRITABLE
XINFO(<i>options</i>)	-	XINFO(NODEF,NOMSG, NOSYMNOSYN,NOXML, NOXML)
XML(CASE(UPPER ASIS))	-	XML(CASE(UPPER))
XREF[(FULL SHORT)] NOXREF	X NX	NX [(FULL)] ¹

注:

1. FULL は、ATTRIBUTES または XREF の指定でサブオプションが省略された場合のデフォルト・サブオプションです。
2. BLANK 文字のデフォルト値は、'05'x 値のタブ文字です。
3. (ALL,SYM) は、TEST の指定でサブオプションを省略した場合のデフォルト・サブオプションです。

AGGREGATE

AGGREGATE オプションは、コンパイラーのリスト時にソース・プログラムの配列と大構造の長さを示す集合長さテーブルを作成します。



省略形: AG、NAG

AGGREGATE オプションの以下のサブオプションは、集合長さテーブル内でのサブ要素のオフセットの表示方法を決定します。

DECIMAL

すべてのオフセットが 10 進数で表示されます。

HEXADEC

すべてのオフセットが 16 進数で表示されます。

集合長さテーブルでは、次元設定のない大構造または小構造は、常にバイトで表現されますが、大構造または小構造に位置合わせされていないビット・要素が含まれていると、長さが不正確になる場合があります。

集合長さテーブルには、配列ではなく非固定エクステントを持つ構造が組み込まれています。しかし、この構造体は非固定エクステントを保持し、構造体内部の要素のサイズとオフセットは、不正確であるか、または * として指定されます。

ARCH

ARCH オプションは、実行可能プログラムの命令が生成されるアーキテクチャーを指定します。このオプションにより、最適化プログラムは特定のハードウェア命令セットの利点を利用できます。型式 (モデル) 番号が属するグループを指定するサブパラメーターがあります。



ARCH レベルで指定できる現行値は、次のとおりです。

- 5 モデル 2064-100 (z/900) の z/Architecture モードで使用可能な命令を使用するコードを生成します。

具体的には、これらの ARCH(5) マシンおよび後継品には、NILL、NILH、OILL、OILH、LLILL、および LLILH などの命令が組み込まれています。

- 6 2084-xxx (z990) および 2086-xxx (z890) モデルの z/Architecture モードで使用可能な命令を使用するコードを生成します。

特に、これらの ARCH(6) マシンと後継マシンについては、コンパイラーは長変位命令セットを活用する場合があります。長変位機能により、既存の 69 の命令と新規の 44 の命令で、20 ビット符号付き変位フィールドを使用できます (前者の場合は既存の命令で未使用のバイトを使用)。20 ビット符号付き変位を使用することにより、基底レジスターまたは基底と指標のレジスター・ペアによって指定されるロケーションを超えて最大 524,287 バイトまでの相対アドレッシングが可能になります。そのロケーションの前では、最大 524,288 バイトまでの相対アドレッシングが可能です。拡張される既存の命令は通常、64 ビット 2 進整数を扱う命令です。新規命令は概して、32 ビット 2 進整数用の新バージョンの命令です。新規命令には、以下の命令も含まれます。

- ストレージのバイトを符号拡張して 32 ビットまたは 64 ビットの結果を汎用レジスターに形成する LOAD BYTE 命令
- 新しい浮動小数点 LOAD および STORE 命令

長変位機能を使用すると、基底レジスターの必要性が低くなるのでレジスター制約が軽減される、使用される命令数が少なくなるのでコード・サイズが小さくなる、アドレス生成インターロックの可能性がなくなるのでパフォーマンスが向上する、という効果があります。

- 7 2094-xxx モデルの z/Architecture モードで使用可能な命令を使用するコードを生成します。

特に、これらの ARCH(7) マシンと後継マシンには、拡張即値機能や拡張変換機能などの機能がサポートする命令が追加されており、コンパイラーがこれらの機能を活用する場合があります。これらの機能の詳細については、「z/Architecture Principles of Operation」を参照してください。

ARCH(7) マシンは、DFP 命令もサポートします。

- 8 2097-xxx モデル (IBM System z10 EC) の z/Architecture モードで使用可能な命令を使用するコードを生成します。

特に、これらの ARCH(8) マシンと後継マシンには、コンパイラーで使用可能な、一般命令拡張機能によってサポートされる命令が追加されています。また、これらのマシンでは、10 進浮動小数点機能によってサポートされる命令が追加されています。これらの命令は、DFP コンパイラー・オプションが指定されていて、ソース・コードに 10 進浮動小数点のデータ・タイプが存在している場合に生成されます。これらの機能の詳細については、「z/Architecture Principles of Operation」を参照してください。

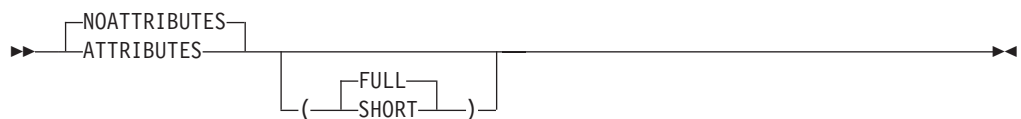
5 より小さい ARCH 値を指定すると、コンパイラーがその値を 5 に再設定します。

注: 上記のモデル番号内の "x" (9672-Rx4 など) は「ワイルドカード」で、そのタイプのマシン名 (9627-RA4 など) の任意の英数字を示します。

注: ARCH(n) でコンパイルされたコードは、 $m \geq n$ の場合にのみ ARCH(m) グループのマシンで実行します。

ATTRIBUTES

ATTRIBUTES オプションは、コンパイラー・リスト内にソース・プログラム ID とそれぞれの属性のテーブルをコンパイラーが組み込むことを指定します。



省略形: A、NA、F、S

FULL

すべての ID と属性がコンパイラー・リストに組み込まれます。FULL がデフォルトです。

SHORT

非参照 ID が省かれ、リストの取り扱いが簡単になります。

ATTRIBUTES と XREF (相互参照テーブルを作成する) を両方入れると、2 つのテーブルが組み合わせられます。ただし SHORT サブオプションと FULL サブオプションが対立する場合は、後から指定したオプションが使用されます。例えば、ATTRIBUTES (SHORT) XREF (FULL) と指定すると、組み合わせられたリストには FULL が使用されます。

BACKREG

BACKREG オプションは、逆チェーン・レジスターを制御し、このレジスターは、ネストされたルーチンが呼び出されたときに、親ルーチンの自動ストレージのアドレスを受け渡すのに使用されます。



PL/I for MVS & VM、OS PL/I V2R3 およびそれ以前のコンパイラーとの互換性のためには、BACKREG(5) を使用するのが最適です。

ENTRY VARIABLE を共用するルーチンはすべて同じ BACKREG オプションでコンパイルしなければなりません。また、アプリケーションの中のコードはすべて同じ BACKREG オプションでコンパイルすることを強くお勧めします。

VisualAge PL/I for OS/390 でコンパイルしたコードは、事実上 BACKREG(11) オプションを使用したものであることに注意してください。Enterprise PL/I V3R1 または V3R2 でコンパイルしたコードも、デフォルトで BACKREG(11) オプションを使用しています。

BIFPREC

BIFPREC オプションは、さまざまな組み込み関数によって戻された FIXED BIN の結果の精度を制御します。

▶▶ BIFPREC (($\overbrace{\hspace{1.5cm}}^{15}$
31)) ▶▶

PL/I for MVS & VM、OS PL/I V2R3 およびそれ以前のコンパイラーと互換性のためには、BIFPREC(15) を使用するのが最適です。

BIFPREC は次の組み込み関数に影響します。

- COUNT
- INDEX
- LENGTH
- LINENO
- ONCOUNT
- PAGENO
- SEARCH
- SEARCHR
- SIGN
- VERIFY
- VERIFYR

BIFPREC コンパイラー・オプションの影響が最も明らかに見えるのは、上記の組み込み関数の結果の 1 つが、パラメーター・リストなしに宣言された外部関数に受け渡されるときです。例えば、次のような部分コードがあるとします。

```
dc1 parm char(40) var;  
dc1 funky ext entry( pointer, fixed bin(15) );  
dc1 beans ext entry;  
call beans( addr(parm), verify(parm), ' ' );
```

関数 *beans* が実際にそのパラメーターを POINTER および FIXED BIN(15) として宣言すると、上記のコードがオプション BIFPREC(31) でコンパイルされたものであった場合、および z/OS のようなビッグ・エンディアン・システム上で実行されて

いた場合に、コンパイラーは 2 番目の引数として 4 バイトの整数を受け渡し、2 番目のパラメーターがゼロであるかのように見えます。

関数 *funky* は、すべてのシステム上でどちらのオプションでも機能することに注意してください。

BIFPREC オプションは、組み込み関数 DIM、HBOUND および LBOUND には影響しません。CMPAT オプションは、次の 3 つの関数から戻された FIXED BIN の結果の精度を判別します。CMPAT(V1) では、これらの配列処理関数は、FIXED BIN(15) の結果を返し、CMPAT(V2) および CMPAT(LE) では、FIXED BIN(31) の結果を返します。CMPAT(V3) では、FIXED BIN(63) の結果が返されます。

BLANK

BLANK オプションは、ブランク文字の代替記号を 10 個まで指定します。

▶▶ BLANK—(—' *char* '—)——▶▶

注: 引用符の間にブランクをコードしないでください。

BLANK 記号用の IBM 提供のデフォルト・コード・ポイントは、X'05' です。

char

単一の SBCS 文字。

英字、数字、および「PL/I 言語解説書」に定義されている特殊文字はどれも指定できません。

BLANK オプションを指定した場合でも、標準のブランク記号はブランクとして認識されます。

BLKOFF

BLKOFF オプションは、疑似アセンブラー・リスト (LIST オプションによって生成される) とステートメント・オフセット・リスト (OFFSET オプションによって生成される) に示されるオフセットを、現行モジュールの開始位置を基準とするか、現行プロシージャーの開始位置を基準とするかを制御します。

▶▶ BLKOFF
NOBLKOFF —▶▶

疑似アセンブラー・リストには、現行モジュールの開始位置からの各ブロックのオフセットも含まれています (それぞれのステートメントに表示されるオフセットを、ブロックまたはモジュールのオフセットに変換できるようにするため)。

CEESTART

CEESTART オプションは、コンパイラーが CEESTART csect を配置する場所を、他のすべての生成済みオブジェクト・コードの前にするか後にするかを指定します。



CEESTART(FIRST) オプションを使用すると、コンパイラーは CEESTART csect を、他のすべての生成済みオブジェクト・コードの前に配置します。
 CEESTART(LAST) オプションを使用すると、他のすべての生成済みオブジェクト・コードの後に配置します。

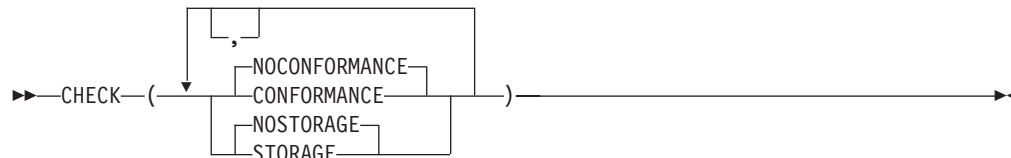
CEESTART(FIRST) を使用すると、バインド・ステップで ENTRY カードが指定されていなければ、バインダーはモジュールのエントリー・ポイントとして CEESTART を選択することになります。

リンカー CHANGE カードを使用するユーザーは、CEESTART(LAST) オプションを使用する必要があります。

なお、MAIN ルーチンは、ENTRY CEESTART リンケージ・エディター・カードとリンクする必要があります。ただし CEESTART(LAST) オプションを使用する場合は、MAIN ルーチンをリンクするときに ENTRY CEESTART カードをインクルードする必要があります。

CHECK

CHECK オプションは、コンパイラーが、さまざまなプログラミング・エラーを検出するための特殊なコードを生成するかどうかを指定します。



省略形: STG、NSTG

CHECK(CONFORMANCE) を指定すると、プロシージャーに渡された引数の属性が、宣言されたパラメーターの属性と一致する場合に、以下の状況で、実行時に検査するコードをコンパイラーが生成します。

- パラメーターが固定長で宣言されたストリング (またはストリングの配列) である場合、渡された引数が一致する長さでないと、STRINGSIZE 条件が発生します。
- パラメーターがストリング (またはストリングの配列) である場合、引数が同じ長さタイプ (VARYING、NONVARYING、または VARYINGZ) でないと、STRINGSIZE 条件が発生します。
- パラメーターが (スカラーまたは構造体の) 配列である場合、渡された引数の定数境界と一致しない定数境界があると、SUBSCRIPTRANGE 条件が発生します。すべてのエクステントが定数で、引数に含まれる配列エレメントのサイズとスペーシングがパラメーターのものと一致しない場合も、SUBSCRIPTRANGE 条件が発生します。構造体内部の配列はチェックされません。

- パラメーターが定数エクステントを持つ構造体または共用体である場合、最後の要素のオフセットが、渡された引数のオフセットと一致しないと、SUBSCRIPTRANGE 条件が発生します。
- プロシージャーに RETURNS BYADDR 属性があり、その属性がストリング・タイプを指定している場合、RETURNS 値に対して渡されたストリングが一致する長さでないと、STRINGSIZE 条件が発生します。

プロシージャーに NODESCRIPTOR オプションが適用される場合、またはブロックに ENTRY ステートメントが含まれている場合、あるいは CMPAT(LE) オプションが有効な場合には、この追加のコードは生成されません。

CHECK(STORAGE) を指定すると、コンパイラーは ALLOCATE ステートメントと FREE ステートメントについて多少異なるライブラリー・ルーチン呼び出します (これらのステートメントが AREA 内に出現する場合を除く)。「PL/I 言語解説書」で説明されている次の組み込み関数は、CHECK(STORAGE) を指定した場合だけ使用できます。

- ALLOCSIZE
- CHECKSTG
- UNALLOCATED

Enterprise PL/I アプリケーションでは、AMODE(24) は非推奨です。

CHECK(STORAGE) オプションを指定してコンパイルされたコードで、AMODE(24) を使用する必要がある場合は、HEAP(,BELOW) ランタイム・オプションも指定する必要があります。

CMPAT

CMPAT オプションは、ストリング、AREA、配列、および構造体 (あるいは、これらのいずれか) を共用するプログラムのために、OS PL/I バージョン 1、OS PL/I バージョン 2、PL/I for MVS and VM、VisualAge PL/I for OS/390 または Enterprise PL/I for z/OS とのオブジェクト互換性を維持するかどうかを指定します。



LE

CMPAT(LE) を指定すると、ユーザー・プログラムは、VisualAge PL/I for OS/390 または Enterprise PL/I for z/OS を使用してコンパイルし、そのコンパイル時に CMPAT(V1) および CMPAT(V2) オプションを使用しなかったプログラムに限り、ストリング、AREA、配列、および構造体を共用できます。

V1 CMPAT(V1) を指定すると、OS PL/I コンパイラーを使用してコンパイルされたプログラム、およびそれ以降の PL/I コンパイラーを使用してコンパイルされたプログラムとの間で、CMPAT(V1) オプションを使用した場合に限りストリング、AREA、配列、および構造体を共用できます。

V2 CMPAT(V2) を指定すると、OS PL/I コンパイラーを使用してコンパイルされたプログラム、およびそれ以降の PL/I コンパイラーを使用してコンパイルされた

プログラムとの間で、CMPAT(V2) オプションを使用した場合に限りストリング、AREA、配列、および構造体を共用できます。

V3 CMPAT(V3) では、CMPAT(V*) オプションのいずれかが使用されていれば、OS PL/I コンパイラでコンパイルされたプログラムおよび新しい PL/I コンパイラでコンパイルされたプログラムで、ストリングを共用できます。ただし、CMPAT(V3) を指定してコンパイルされていないコードでは、AREA、配列、または構造体は共用できません。

DB2 ストアード・プロシージャは、CMPAT(LE) を指定してコンパイルしてはいけません。

1 つのアプリケーション内のモジュールはすべて、同じ CMPAT オプションを指定してコンパイルする必要があります。

新旧のコードを混合する場合は、次の制限があります。これらの制限に関する情報については、「Enterprise PL/I for z/OS コンパイラおよびランタイム 移行ガイド」を参照してください。

DFT(DECLIST) オプションは、CMPAT(V*) オプションと競合するため、そのいずれかのオプションとともに指定された場合、メッセージが出されて、DFT(DESCLOCATOR) オプションとしてとられます。

CMPAT(V3) では、配列は、8 バイトの整数としてとれるすべての値で宣言できます。ただし、配列の合計サイズには今のところまだ、CMPAT(V2) で宣言された配列と同じ制限があります。

CMPAT(V3) では、以下の組み込み関数で、FIXED BIN(63) の結果が常に返されます。

- CURRENTSIZE/CSTG
- DIMENSION
- HBOUND
- LBOUND
- LOCATION
- SIZE/STG

これらの関数は 8 バイトの整数値を返すため、CMPAT(V3) では、LIMITS オプションの FIXEDBIN サブオプションにおける 2 番目のオプションは、63 にする必要があります。

ただし、CMPAT(V3) でも、ステートメントおよびフォーマット・ラベル定数は、4 バイト整数を使用して指定する必要があります。

CODEPAGE

CODEPAGE オプションは、次の目的に使用するコード・ページを指定します。

- CHARACTER と WIDECHAR の間の変換
- PLISAX 組み込みサブルーチンによって使用されるデフォルト・コード・ページ

▶▶CODEPAGE(—ccsid—)▶▶

サポートされる CCSID は、次のとおりです。

01047	01145	00273	00297
01140	01146	00277	00500
01141	01147	00278	00871
01142	01148	00280	00819
01143	01149	00284	00813
01144	00037	00285	00920
01025	01155		

デフォルト CCSID 1140 は、CCSID 37 (EBCDIC Latin 1、米国) と同等ですが、ユーロ記号を含んでいます。

COMMON

COMMON オプションは、EXTERNAL STATIC 変数用の CM リンケージ・レコードを生成するようコンパイラーに指示します。



COMMON オプションを指定すると、NORENT オプションが適用される場合に、RESERVED ではなく、INITIAL 値が含まれない、EXTERNAL STATIC 変数用の CM リンケージ・レコードが生成されます。これは、OS PL/I コンパイラーによって行われることと一致します。

NOCOMMON オプションを指定すると、Enterprise PL/I の初期のリリースではそうであったように、SD レコードが書き込まれます。

COMMON オプションは、RENT オプションと一緒に、または $n > 7$ の場合の LIMITS(EXTNAME(n)) と一緒に使用してはなりません。

COMPACT

コード生成中に実行される最適化において、処理は早い生成コードが大きくなる最適化と、処理は遅いが生成コードが小さくなる最適化のいずれかを選択しなければなりません。COMPACT オプションはこれらの選択項目に影響を与えます。COMPACT オプションを使用すると、コンパイラーは、コード・サイズの増大を制限する最適化を選択します。インライン化を含む、さまざまな最適化処理間の相互作用により、COMPACT オプションを使用してコンパイルしたコードが、常にコードとデータが少ないものになるとは限りません。



ユーザーのアプリケーションでの COMPACT オプションの使用について評価するには、次のようにします。

- COMPACT と NOCOMPACT で生成されたオブジェクトのサイズを比較します。
- COMPACT と NOCOMPACT で生成されたモジュールのサイズを比較します。

- COMPACT と NOCOMPACT を指定した代表的なワークロードの実行時間を比較します。

オブジェクトとモジュールが実行時の受け入れ可能な変更を伴い小さくなっていた場合は、COMPACT を使用することを考慮できます。

コンパイラーに新たな最適化が加えられると、COMPACT オプションの動作が変化する場合があります。コンパイラーの新規リリースが公開されるたびに、また、アプリケーション・コードを変更する際に、このオプションの使用を再評価するとよいでしょう。

COMPILE

COMPILE オプションは、プリプロセス中またはセマンティック検査中に指定されている重大度のメッセージが生成された場合に、ソース・プログラムのすべてのセマンティック検査のあとでコンパイラーを停止させます。コンパイラーが処理を続行するかどうかは、下記のリスト内の NOCOMPILE オプションで指定されたとおりの、検出したエラーの重大度で決まります。NOCOMPILE オプションを指定すると、セマンティック検査の後、処理は無条件に停止されます。



省略形: C、NC

COMPILE

重大エラーまたは回復不能エラーが検出されない限り、コードを生成します。
NOCOMPILE(S) と同等です。

NOCOMPILE

コンパイルはセマンティック検査後に停止されます。

NOCOMPILE(W)

警告、エラー、重大エラー、または回復不能エラーが検出された場合は、コードを生成しません。

NOCOMPILE(E)

エラー、重大エラー、または回復不能エラーが検出された場合は、コードを生成しません。

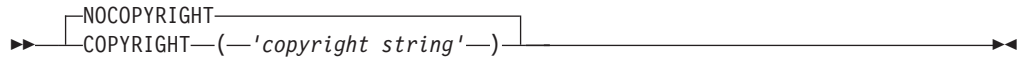
NOCOMPILE(S)

重大エラーまたは回復不能エラーが検出された場合は、コードを生成しません。

コンパイルが NOCOMPILE オプションによって終了された場合、相互参照リストおよび属性リストを作成することができます。ソース・プログラムに続く他のリストは作成されません。

COPYRIGHT

COPYRIGHT オプションは、オブジェクト・モジュール内にストリングを配置します。このストリングは、オブジェクトのリンク先であるロード・モジュールとともにメモリーにロードされます。



ストリングの長さは 1000 文字に制限されます。ただし、ストリングは、100 文字を超える場合、オプション・リストに表示されません。

ロケールが異なってもストリングを読み取ることができるように、インバリエント文字セットからの文字だけを使用する必要があります。

CSECT

CSECT オプションを使用すると、名前付き CSECT がオブジェクト・モジュール (生成された場合) に組み込まれます。製品のサービス、またはプログラムのデバッグの補助に SMP/E を使用する場合は、このオプションを使用してください。



省略形: CSE、NOCSE

NOCSECT オプションを使用すると、ユーザーのオブジェクト・モジュールのコードおよび静的セクションにはデフォルトの名前が指定されます。

CSECT オプションを使用すると、ユーザーのオブジェクト・モジュールのコードおよび静的セクションには、次のように定義された「パッケージ名」によって名前が指定されます。

- パッケージ・ステートメントが使用された場合は、「パッケージ名」はパッケージ・ステートメントの一番左のラベルになります。
- そうでない場合は、最初のプロシーチャー・ステートメントにある一番左のラベルが「パッケージ名」になります。

長さ 7 の「変更パッケージ名」は、次のように形成されます。

- パッケージ名が 7 文字より短い場合は、"*" が前に追加されて、7 文字の長さの変更パッケージ名が作成されます。
- パッケージ名が 7 文字より長い場合は、最初の n 文字および最後の $7 - n$ 文字を使用して変更パッケージ名が作成されますが、値 n は CSECTCUT オプションで設定されます。
- それ以外の場合は、パッケージ名が変更パッケージ名にコピーされます。

コード csect 名は、変更パッケージ名に '1' を追加して作成されます。

静的 csect 名は、変更パッケージ名に '2' を追加して作成されます。

したがって、“SAMPLE” という名前のパッケージの場合、コード csect 名は “*SAMPLE1” になり、静的 csect 名は “*SAMPLE2” になります。

CSECTCUT

CSECTCUT オプションは、CSECT オプションの処理のときに、コンパイラーが、どのようにロング・ネームを処理するかを制御します。

▶▶ CSECTCUT—(—⁴_n—)——▶▶

CSECTCUT オプションは、CSECT オプションを指定しない限り影響しません。CSECT オプションで使用する「パッケージ名」が 7 文字以下の場合も影響しません。

CSECTCUT オプションの中の値 n は、0 および 7 の間でなければいけません。

CSECT オプションで使用する「パッケージ名」が 7 文字より大きい場合は、コンパイラーが最初の n 文字と最後の $7 - n$ 文字を使ってこの名前を 7 文字に縮小します。

例えば、BEISPIEL という名前のプロシージャークラから成るコンパイルでは、

- CSECTCUT(3) を指定すると、コンパイラーが名前を BEIPIEL と縮小します。
- CSECTCUT(4) を指定すると、コンパイラーが名前を BEISIEL と縮小します。

CURRENCY

CURRENCY オプションを指定すると、ドル記号の代わりにピクチャー・ストリングで代替文字を指定できます。

▶▶ CURRENCY—(—'—^{\$}_x—')——▶▶

省略形: CURR

- x コンパイラーおよびランタイムがピクチャー・ストリング内でドル記号として認識し受け入れる必要がある文字。

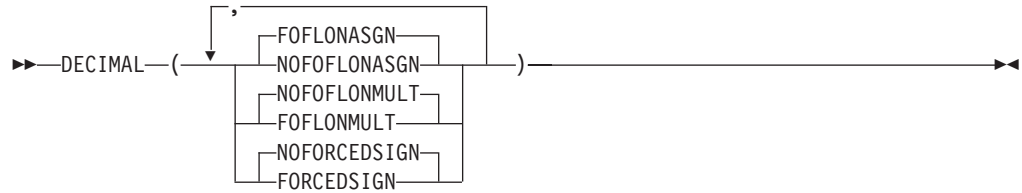
DBCS

DBCS オプションを指定すると、GRAPHIC オプションが指定されていなくても、リスト (生成された場合) は DBCS の存在を識別します。

▶▶ ^{NODBCS}_{DBCS}——▶▶

NODBCS オプションを指定すると、リスト (生成された場合) は DBCS シフト・コードをすべて “.” として示します。

GRAPHIC も指定されている場合は、NODBCS オプションを指定してはなりません。



FOFLONASGN

FOFLONASGN オプションでは、FIXEDOVERFLOW 条件が使用可能で SIZE 条件が使用不可である場合、コンパイラーは、FIXED DECIMAL 式が FIXED DECIMAL ターゲットに代入されて有効数字が失われたときに、常に FIXEDOVERFLOW 条件を発生するコードを生成する必要があります。

反対に、NOFOFLONASGN オプションを指定すると、コンパイラーは、このような代入で有効数字が失われたときに FIXEDOVERFLOW 条件を発生しないコードを生成します。

したがって、例えば FIXED DEC(5) と宣言された変数 A があるとした場合、代入 $A = A + 1$ は、FOFLONASGN オプションでは FOFL が発生しますが、NOFOFLONASGN オプションでは発生しません。

ただし、NOFOFLONASGN オプションを指定すると、LIMITS オプションの FIXEDDEC サブオプションで許可されるよりも桁数が多くなる結果を生成する演算によって、FIXEDOVERFLOW 条件が発生する可能性があることに注意してください。例えば、値 999_999_999_999_999 を持つ FIXED DEC(15) と宣言された変数 B があり、LIMITS の FIXEDDEC サブオプションで最大精度を 15 と指定しているものとした場合、代入 $B = B + 1$ は、FIXEDOVERFLOW 条件（もちろん、FOFL が使用可能な場合）を発生させます。これは、加算 $B + 1$ がこの条件を発生させるためです。

FOFLONMULT

FOFLONMULT オプションは、組み込み関数で指定された精度としては大きすぎる FIXED DEC の結果を生成する MULTIPLY 組み込み関数を使用されている場合に、FIXEDOVERFLOW 条件を発生させるコードを、コンパイラーが生成するように要求します。

逆に、NOFOFLONMULT オプションでは、コンパイラーは、そのような MULTIPLY 組み込み関数を使用されている場合に切り捨てた結果を生成するコードを生成します。

なお、FOFLONMULT オプションを使用すると、デフォルト言語のセマンティクスが変更されます（このため、SIZE 条件が有効にされていない限り、FIXED DEC に適用された MULTIPLY 組み込み関数の大きすぎる結果は切り捨てられます）。

FORCEDSIGN

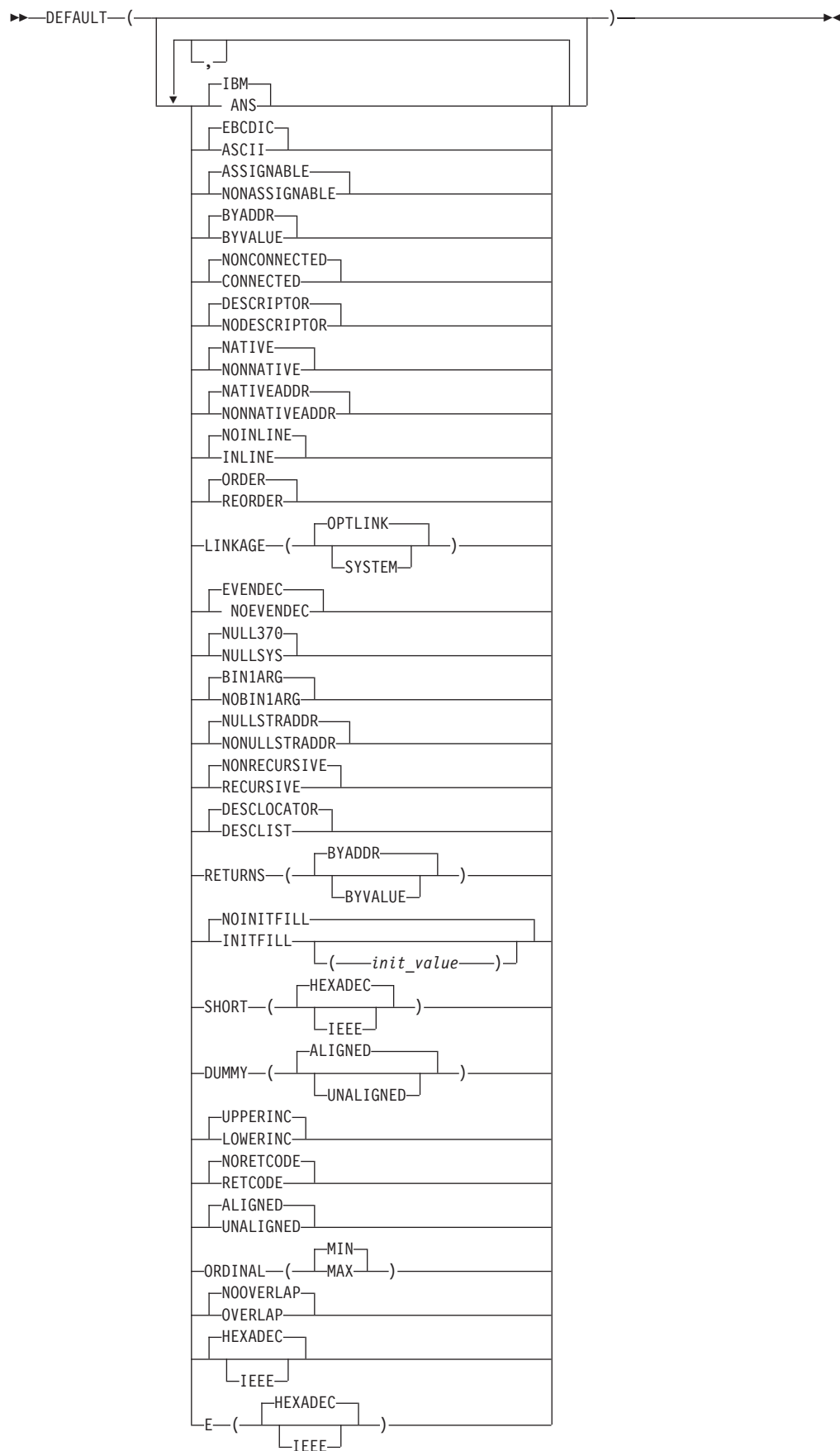
FORCEDSIGN オプションは、値ゼロの FIXED DECIMAL 結果が生成された場合に、結果の符号ニブルが常に値 'C'X を持つようにするために、追加のコードを生成するようコンパイラーに強制します。このオプションを指定すると、コンパイラーが生成するコードは、NOFORCEDSIGN サブオプションで生成されるコードよりもパフォーマンスが大幅に低下します。

また、このオプションが有効であると、プログラムを実行するときに、データ例外がより発生しやすくなります。例えば、1 つの `FIXED DEC(5)` 変数を別の `FIXED DEC(5)` 変数に代入する場合、通常、コンパイラーは移動を実行するために `MVC` 命令を生成します。しかし、このオプションが有効であると、結果に好みの符合が付くようにするために、コンパイラーは移動を実行するために `ZAP` 命令を生成します。ソースに無効なパック 10 進データが含まれていると、`MVC` 命令ではなく、`ZAP` 命令が 10 進データ例外を発生させます。

このオプションを使用すると、ある `PICTURE` 変数が別の `PICTURE` 変数に代入される場合も、データ例外が発生することがあります。この変換では通常、`FIXED DEC` への暗黙的変換が行われますが、このオプションが使用されていると、この変換で `ZAP` 命令が生成され、ソースに無効なデータが含まれていると、この命令がデータ例外を発生させるからです。

DEFAULT

DEFAULT オプションは、属性およびオプションのデフォルトを指定します。これらのデフォルトは、属性またはオプションがソースで明示的または暗黙に指定されていない場合だけ適用されます。



省略形: DFT、ASGN、NONASGN、NONCONN、CONN、INL、NOINL

IBM または ANS

IBM または ANS SYSTEM のデフォルトを使用します。IBM および ANS の場合の演算デフォルトは次のとおりです。

属性	DEFAULT(IBM)	DEFAULT(ANS)
FIXED DECIMAL	(5,0)	(10,0)
FIXED BINARY	(15,0)	(31,0)
FLOAT DECIMAL	(6)	(6)
FLOAT BINARY	(21)	(21)

IBM サブオプションのもとでは、名前が I から N までの文字で始まる変数のデフォルトは FIXED BINARY であり、それ以外の変数のデフォルトは FLOAT DECIMAL です。ANS サブオプションを選択した場合は、すべての変数のデフォルトは FIXED BINARY です。

IBM がデフォルトです。

ASCII | EBCDIC

このオプションは、問題プログラム文字データの内部表現に使用する文字セットのデフォルトを設定するために使用します。

ASCII は、ASCII 文字セット照合順序に依存するプログラムをコンパイルするときにだけ使用します。例えば、プログラムが数字のソート・シーケンスまたは小文字および大文字のアルファベット順を使用している場合に、このような依存関係が存在します。また、高位ビットの状態を変えて大文字の英字を作成するプログラムにも、このような依存関係が存在します。

注: コンパイラーは、A および E を文字ストリングの接尾部としてサポートします。A 接尾部は、EBCDIC コンパイラー・オプションが効力を持っている場合でも、ストリングが ASCII データを表現していることを示します。同様に、E 接尾部は、DEFAULT(ASCII) を選択した場合でもストリングが EBCDIC であることを示します。

'123'A は '313233'X と同じ
'123'E は 'F1F2F3'X と同じ

EBCDIC がデフォルトです。

ASSIGNABLE | NONASSIGNABLE

このオプションを指定すると、コンパイラーは、ASSIGNABLE 属性または NONASSIGNABLE 属性を持つものとして宣言されていないすべての静的変数に、指定の属性を適用します。コンパイラーは、NONASSIGNABLE 変数が割り当てのターゲットであるステートメントにフラグを付けます。

ASSIGNABLE がデフォルトです。

BYADDR | BYVALUE

引数またはパラメーターをアドレスで渡すか値で渡すかのデフォルトを設定します。BYVALUE は、いくつかの特定の引数およびパラメーターだけに適用されます。詳しくは、「PL/I 言語解説書」を参照してください。

BYADDR はデフォルトです。

CONNECTED | NONCONNECTED

パラメーターの接続または非接続に関するデフォルトを設定します。

CONNECTED を指定すると、パラメーターをレコード単位の入出力のターゲットまたはソースとして、あるいはストリング・オーバーレイ定義の基礎として使用できます。

NONCONNECTED がデフォルトです。

DESCRIPTOR | NODESCRIPTOR

PROCEDURE ステートメントで **DESCRIPTOR** を使用すると、記述子リストが渡されたことを示します。**ENTRY** ステートメントで **DESCRIPTOR** を使用すると、記述子リストを渡す必要があることを示します。**NODESCRIPTOR** を使用すると、より効率的なコードになりますが、次の制限があります。

- **PROCEDURE** ステートメントでは、次のものを含むパラメーターが 1 つでもある場合は、**NODESCRIPTOR** は無効です。
 - 配列の境界、ストリングの長さ、または配列のサイズ (**NONASSIGNABLE** 属性を持つ **VARYING** ストリングまたは **VARYINGZ** ストリングの場合を除く) に指定されたアスタリスク (*)
 - **NONCONNECTED** 属性
 - **UNALIGNED BIT** 属性
- **ENTRY** 宣言では、**ENTRY** 記述子リストの中で配列の境界、ストリングの長さ、またはエリアのサイズにアスタリスク (*) が指定されている場合は、**NODESCRIPTOR** は無効です。

DESCRIPTOR がデフォルトです。

NATIVE | NONNATIVE

このオプションは、固定 2 進数、序数、オフセット、エリア、および可変ストリング・データの内部表現だけに影響します。**NONNATIVE** サブオプションが有効な場合、**NONNATIVE** 属性は、**NATIVE** 属性を指定して宣言されていないこの種のすべての変数に適用されます。

NONNATIVE は、非ネイティブ・フォーマットに依存してこの種の変数を保持するプログラムをコンパイルする場合だけ指定してください。

固定 2 進変数がポインター変数またはオフセット変数を基礎とするプログラム (あるいは逆にポインター変数またはオフセット変数が固定 2 進変数を基礎とするプログラム) の場合は、次のどちらかを指定します。

- **NATIVE** サブオプションと **NATIVEADDR** サブオプションの両方
 - **NONNATIVE** サブオプションと **NONNATIVEADDR** サブオプションの両方
- それ以外の組み合わせを指定すると、結果は予測できません。

NATIVE がデフォルトです。

NATIVEADDR | NONNATIVEADDR

このオプションはポインターの内部表示だけに影響します。

NONNATIVEADDR サブオプションが有効な場合、**NONNATIVE** 属性は、**NATIVE** 属性を指定して宣言されていないすべてのポインター変数に適用されます。

固定 2 進変数がポインター変数またはオフセット変数を基礎とするプログラム (あるいは逆にポインター変数またはオフセット変数が固定 2 進変数を基礎とするプログラム) の場合は、次のどちらかを指定します。

- NATIVE サブオプションと NATIVEADDR サブオプションの両方
- NONNATIVE サブオプションと NONNATIVEADDR サブオプションの両方

それ以外の組み合わせを指定すると、結果は予測できません。

NATIVEADDR がデフォルトです。

INLINE | NOINLINE

このオプションは、インライン・プロシージャ・オプションのデフォルトを設定します。

INLINE を指定すると、コードの実行が高速になりますが、場合によっては実行可能ファイルも大きくなります。インライン化によるパフォーマンスの改善方法の詳細については、309 ページの『第 11 章 パフォーマンスの向上』を参照してください。

NOINLINE がデフォルトです。

ORDER | REORDER

ソース・コードの最適化に影響します。REORDER を指定すると、ソース・コードをさらに最適化できます。309 ページの『第 11 章 パフォーマンスの向上』を参照してください。

ORDER がデフォルトです。

LINKAGE

プロシージャ呼び出しのためのリンケージ規則は次のとおりです。

OPTLINK

Enterprise PL/I のデフォルトのリンケージ規則。このリンケージにより最良のパフォーマンスが得られます。

SYSTEM

システム API の標準リンケージ規則。

LINKAGE(OPTLINK) は、JAVA によって呼び出されるか、JAVA の呼び出しを行うルーチンすべてに対して使用する必要があります。また、C によって呼び出されるか、C の呼び出しを行うルーチンすべてに対しても使用する必要があります (C コードがデフォルト以外のリンケージを使用してコンパイルされた場合を除く)。

LINKAGE(SYSTEM) は、最後の (かつ最後だけの) パラメーターのアドレス内で高位ビットがオンになっていることを予期する、非 PL/I ルーチンすべてに対して使用する必要があります。

LINKAGE(OPTLINK) がデフォルトです。

EVENDEC | NOEVENDEC

このサブオプションは、偶数精度を指定して宣言された固定小数点変数に関するコンパイラの許容度を制御します。

NOEVENDEC のもとでは、固定小数点変数の精度は次の最大の奇数に切り上げられます。

EVENDEC を指定して FIXED DEC(2) 変数に 123 を割り当てると、SIZE 条件が発生します。NOEVENDEC を指定した場合は、SIZE 条件は発生しません。

EVENDEC がデフォルトです。

BIN1ARG | NOBIN1ARG

このサブオプションは、非プロトタイプ関数に渡される 1 バイト REAL FIXED BIN 引数をコンパイラーがどのように扱うかを制御します。

BIN1ARG の場合、コンパイラーはプロトタイプ化されていない関数に FIXED BIN 引数を現状のままで渡します。

しかし NOBIN1ARG の場合、コンパイラーは、プロトタイプ化されていない関数に渡された 1 バイトの REAL FIXED BIN 引数を 2 バイトの FIXED BIN に一時的に代入してから、代わりにそれを渡します。

次の例を考えてみてください。

```
dc1 f1 ext entry;  
dc1 f2 ext entry( fixed bin(15) );  
  
call f1( 1b );  
call f2( 1b );
```

DEFAULT(BIN1ARG) を指定した場合、コンパイラーは 1 バイトの FIXED BIN(1) 引数のアドレスをルーチン f1 に渡し、2 バイトの FIXED BIN(15) 引数のアドレスをルーチン f2 に渡します。しかし DEFAULT(NOBIN1ARG) を指定した場合には、コンパイラーはどちらのルーチンに対しても、2 バイトの FIXED BIN(15) 引数のアドレスを渡します。

ルーチン f1 が COBOL ルーチンの場合、そのルーチンに対して 1 バイトの整数引数を渡すと、COBOL は 1 バイトの整数をサポートしていないため問題が生じることに注意してください。この場合、DEFAULT(NOBIN1ARG) を使用するのも有用ですが、エンタリー宣言で引数属性を指定する方が良いでしょう。

BIN1ARG がデフォルトです。

NULLSTRADDR | NONULLSTRADDR

このサブオプションは、コンパイラーが引数として渡されたヌル・ストリングをどのように処理するのかを制御します。

NULLSTRADDR では、ヌル・ストリングが入り口呼び出しで引数として指定された場合、コンパイラーは自動ストレージの初期化部分のアドレスを渡します。これは、OS PL/I および PL/I for MVS コンパイラーの動作と互換性があります。

しかし、NONULLSTRADDR では、ヌル・ストリングが入り口呼び出しで引数として指定されると、コンパイラーは引数のアドレスとして NULL ポインターを渡します。これは、Enterprise PL/I コンパイラーの早期リリースの動作と互換性があります。

NULLSTRADDR がデフォルトです。

NULLSYS | NULL370

このサブオプションは、NULL 組み込み関数によって戻される値を決定します。NULLSYS を指定すると、binvalue(null()) は 0 に等しくなります。以前の PL/I のリリースの場合と同じく binvalue(null()) を 'ff_00_00_00'xn に等しくしたい場合は、NULL370 を指定します。

NULL370 がデフォルトです。

RECURSIVE | NONRECURSIVE

DEFAULT(RECURSIVE) を指定すると、コンパイラーはすべてのプロシージャに RECURSIVE 属性を適用します。DEFAULT(NONRECURSIVE) を指定すると、RECURSIVE 属性を持つプロシージャ以外のすべてのプロシージャが非再帰的になります。

NONRECURSIVE がデフォルトです。

DECLIST | DESCLOCATOR

DEFAULT(DECLIST) を指定すると、コンパイラーはリストの中のすべての記述子を「隠れた」最後のパラメーターとして渡します。

DEFAULT(DESCLOCATOR) を指定すると、以前のリリースの PL/I の場合と同様に、記述子を必要とするパラメーターが記述子またはロケーターを使用して渡されます。したがって、古いコードで、1 つのルーチンからポインターを受け取ることを予想しているルーチンへ構造体を渡していた場合も、その古いルーチンは引き続き有効です。

DFT(DECLIST) オプションは、CMPAT(V*) オプションと対立するため、そのいずれかのオプションとともに指定された場合、メッセージが出されて、DFT(DESCLOCATOR) オプションとしてとられます。

DESCLOCATOR がデフォルトです。

RETURNS (BYVALUE | BYADDR)

関数が値を戻す方法のデフォルトを設定します。詳しくは、「PL/I 言語解説書」を参照してください。

アプリケーションに ENTRY ステートメントが含まれていて、その ENTRY ステートメントまたはそのステートメントを含むプロシージャ・ステートメントに RETURNS オプションが指定されている場合は、RETURNS(BYADDR) を指定してください。また、それらのエントリーのエントリー宣言でも、RETURNS(BYADDR) を指定する必要があります。

RETURNS(BYADDR) がデフォルトです。

INITFILL | NOINITFILL

このサブオプションは、自動変数のデフォルト初期化を制御します。

INITFILL に 16 進値 (nn) を指定した場合は、ブロックに入るたびに、ブロック内のすべての自動変数によって使用されるストレージが、その値を使用して初期化されます。16 進値を入力しない場合、デフォルトは '00' です。

16 進値は引用符を付けても付けなくても指定できますが、引用符を付けて指定する場合は、ストリングに X 接尾部を付けてはなりません。

NOINITFILL を指定した場合は、変数が明示的に初期化されない限り、自動変数によって使用されるストレージには任意のビット・パターンを保持することができます。

INITFILL を指定するとプログラムの実行速度が大幅に低下する可能性があるので、実動プログラム内では指定しないでください。ただし、INITFILL オプションの生成するコードは、LE STORAGE オプションよりも高速に実行されます。またこのオプションは、プログラム開発時に、未初期化自動変数を検出するため

に役立ちます。DFT(INITFILL('00')) と DFT(INITFILL('ff')) を指定してプログラムが正しく実行されれば、未初期化自動変数はおそらく存在しません。

NOINITFILL がデフォルトです。

SHORT (HEXADEC | IEEE)

このサブオプションは、IBM 以外の UNIX コンパイラーとの互換性を向上させます。SHORT (HEXADEC) は、 $p \leq 21$ の場合に FLOAT BIN (p) を短い (4 バイトの) 浮動小数点数にマップします。SHORT(IEEE) は、 $p \leq 24$ の場合に FLOAT BIN (p) を短い (4 バイトの) 浮動小数点数にマップします。

SHORT (HEXADEC) がデフォルトです。

DUMMY (ALIGNED | UNALIGNED)

このサブオプションは、仮引数が作成される状態の数を減らします。

DUMMY(ALIGNED) は、引数が位置合わせにおいてのみパラメーターと相違している場合にも、仮引数を作成するべきであることを示します。

DUMMY(UNALIGNED) は、スカラー (不変ビットを除く) またはスカラーの配列が位置合わせにおいてのみパラメーターと相違している場合に、そのスカラーまたはスカラーの配列には仮引数を作成してはならないことを示します。

次の例を考えてみてください。

```
dc1
1 a1 unaligned,
2 b1 fixed bin(31),
2 b2 fixed bin(15),
2 b3 fixed bin(31),
2 b4 fixed bin(15);

dc1 x entry( fixed bin(31) );

call x( b3 );
```

DEFAULT(DUMMY(ALIGNED)) を指定すると仮引数が作成されますが、DEFAULT(DUMMY(UNALIGNED)) を指定すると仮引数は作成されません。

DUMMY(ALIGNED) がデフォルトです。

LOWERINC | UPPERINC

LOWERINC を指定した場合は、コンパイラーは INCLUDE ファイルの実際のファイル名が小文字であることを要求します。UPPERINC を指定した場合は、コンパイラーはこの名前が大文字であることを要求します。

注: このサブオプションは、z/OS UNIX 環境でのコンパイルにだけ適用されます。

z/OS UNIX 環境では、インクルード名は拡張子「.inc」を使用して作成されます。したがって、例えば DFT(LOWERINC) オプションを指定して、%INCLUDE STANDARD; ステートメントを使用すると、コンパイラーは *standard.inc* の組み込みを試みます。一方 DFT(UPPERINC) オプションを指定して %INCLUDE STANDARD; ステートメントを使用すると、コンパイラーは *STANDARD.INC* の組み込みを試みます。

UPPERINC がデフォルトです。

RETCODE | NORETCODE

RETCODE を指定すると、コンパイラーは、RETURNS 属性を持たない外部プ

ロシージャーに対して余分のコードを追加して、そのプロシージャーが、プロシージャーからの戻りの直前に PLIRETV 組み込み関数を呼び出して取得した整数値を戻すようにします。

NORETCODE を指定すると、RETURNS 属性を持たないプロシージャーに対して特別なコードは生成されません。

NORETCODE がデフォルトです。

ALIGNED | UNALIGNED

このサブオプションにより、すべてのユーザー変数でバイト位置合わせを強制できます。

ALIGNED を指定すると、明示的に (おそらく親構造体で) または DEFAULT ステートメントにより暗黙に UNALIGNED 属性が指定されていない限り、文字、ビット、グラフィック、およびピクチャーを除くすべての変数に ALIGNED 属性が与えられます。

UNALIGNED を指定すると、明示的に (おそらく親構造体で) または DEFAULT ステートメントにより暗黙に ALIGNED 属性が指定されていない限り、すべての変数に UNALIGNED 属性が与えられます。

ALIGNED がデフォルトです。

ORDINAL(MIN | MAX)

ORDINAL(MAX) を指定すると、定義に PRECISION 属性が含まれていないすべての順序数に、属性 PREC(31) が与えられます。これを指定しない場合は、これらの順序数には、その値の範囲をカバーする最小の精度が与えられます。

ORDINAL(MIN) がデフォルトです。

OVERLAP | NOOVERLAP

OVERLAP を指定すると、コンパイラーは、割り当てでソースとターゲットのオーバーラップが起こることがあると想定し、割り当ての結果が正しくなるように、必要に応じて追加のコードを生成します。

NOOVERLAP を使用すると、パフォーマンスの優れたコードが生成されます。しかし、NOOVERLAP を使用する場合は、ソースとターゲットが決してオーバーラップしないようにしなければなりません。

NOOVERLAP がデフォルトです。

HEXADEC | IEEE

このサブオプションを使用すると、FLOAT 変数すべて、および浮動小数点の中間結果すべての保管に使用するデフォルト表現を指定できます。また、このサブオプションは、コンパイラーが浮動小数点の式を評価するときに、16 進の命令と数学ルーチンを使用するか、IEEE 浮動小数点の命令と数学ルーチンを使用するかを決定します。

JAVA とやり取りするプログラムは IEEE オプションを使用する 경우가多く、浮動小数点データのデフォルト表現として IEEE を使用するプラットフォームとの間でデータの受け渡しをするプログラムも、IEEE オプションを使用する場合があります。

HEXADEC がデフォルトです。

E (HEXADEC | IEEE)

E サブオプションは、E フォーマット項目の指数として使用する数字の桁数を指定します。

E(IEEE) を指定すると、E フォーマット項目の指数として 4 桁の数字が使用されます。

E(HEXADEC) を指定すると、E フォーマット項目の指数として 2 桁の数字が使用されます。

DFT(E(HEXADEC)) を指定した場合は、99 より大きい絶対値を持つ指数がある式の使用を試みると、SIZE 条件が発生します。

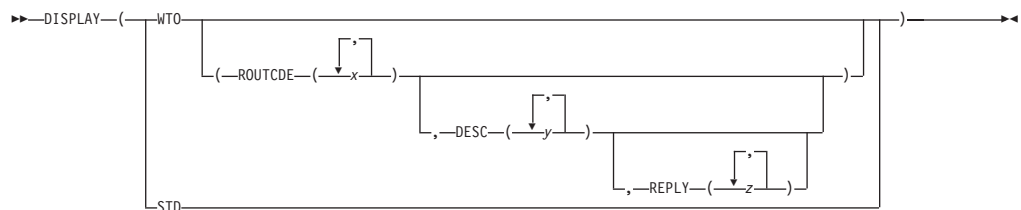
コンパイラー・オプション DFT(IEEE) が有効になっている場合、通常はオプション DFT(E(IEEE)) も使用する必要があります。ただしこのオプションを指定すると、DFT(E(HEXADEC)) を指定した場合には有効になる E フォーマット項目のいくつかが無効になります。例えば、DFT(E(IEEE)) を指定すると、E フォーマットが無効であるためにステートメント "put skip edit(x) (e(15,8));" にフラグが立てられます。

E(HEXADEC) がデフォルトです。

デフォルト: DEFAULT(IBM EBCDIC ASSIGNABLE BYADDR NONCONNECTED
DESCRIPTOR NATIVE NATIVEADDR NOINLINE ORDER LINKAGE(OPTLINK)
EVENDEC NOINITFILL UPPERINC NULL370 BIN1ARG NULLSTRADDR
NONRECURSIVE DESCLOCATOR RETURNS(BYADDR) SHORT(HEXADEC)
DUMMY(ALIGNED) NORETCODE ALIGNED ORDINAL(MIN) NOOVERLAP
HEXADEC E(HEXADEC))

DISPLAY

DISPLAY オプションは、DISPLAY ステートメントが入出力を実行する方法を決定します。



STD

DISPLAY ステートメントは、すべてテキストを stdout に書き出し、REPLY テキストを stdin から読み込んで完了します。

WTO

REPLY が指定されない DISPLAY ステートメントはすべて WTO を使用して完了し、REPLY が指定された DISPLAY ステートメントはすべて WTOR を使用して完了します。これはデフォルトです。

次のサブオプションがサポートされています。

ROUTCDE

WTO で ROUTCDE として使用される 1 つ以上の値を指定します。デフォルトの ROUTCDE は 2 です。

DESC

WTO で DESC として使用される 1 つ以上の値を指定します。デフォルトの DESC は 3 です。

REPLY

WTOR で DESC として使用される 1 つ以上の値を指定します。省略した場合は、DESC オプション (またはデフォルト) の値が使用されます。

ROUTCDE、DESC、および REPLY に指定される値はすべて 1 から 16 までの間の値でなければなりません。

DLLINIT

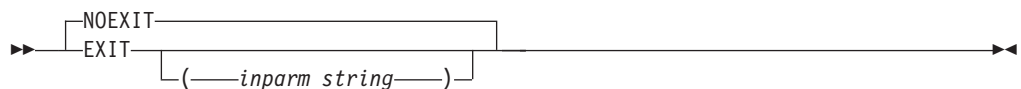
DLLINIT オプションは、MAIN でないすべての外部プロシージャに OPTIONS(FETCHABLE) を適用します。このオプションは、1 つの外部プロシージャを含むコンパイル単位でだけ使用してください。その場合、そのプロシージャを DLL としてリンクする必要があります。



NODLLINIT はユーザーのプログラムには影響しません。

EXIT

EXIT オプションにより、コンパイラー・ユーザー出口を呼び出すことができます。



inparm_string

初期化中にコンパイラー・ユーザー出口ルーチンに渡されるストリング。ストリングの長さは最大 31 文字です。

このオプションの活用方法の詳細については、459 ページの『第 20 章 ユーザー出口の用法』を参照してください。

EXTRN

EXTRN オプションは、外部入り口定数の EXTRN を発行する時期を制御します。



FULL

宣言された外部入り口定数すべてに対して EXTRN を発行します。これはデフォルトです。

SHORT

EXTRN は、参照された定数に対してだけ発行されます。

FLAG

FLAG オプションは、コンパイラー・リストにメッセージをリストすることが必要になるエラーの最小重大度を指定します。



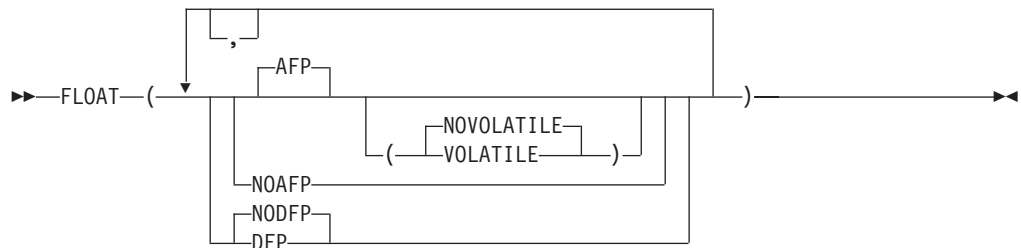
省略形: F

- I** すべてのメッセージをリストします。
- W** 通知メッセージを除くすべてのメッセージをリストします。
- E** 警告メッセージと通知メッセージを除くすべてのメッセージをリストします。
- S** 重大エラー・メッセージおよび回復不能エラー・メッセージだけをリストします。

指定した重大度を下回っているメッセージ、またはコンパイラー出力ルーチンによりフィルターに掛けられて取り除かれたメッセージは、リストされません。

FLOAT

FLOAT オプションは、追加の浮動小数点レジスターの使用、および 10 進浮動小数点をサポートするかどうかを制御します。



NOAFP

コンパイラー生成コードは従来式の 4 つの浮動小数点レジスターを使用します。

AFP

コンパイラー生成コードは 16 の浮動小数点レジスターすべてを使用できます。

VOLATILE

コンパイラーは、呼び出されたプログラムが FPR8 から FPR15 までのレジスターを保持することは想定しません。そのためコンパイラーは、これらのレジスターを保護するための追加コードを生成します。

このオプションを指定すると、浮動小数点レジスターが使用されなくても追加コードが生成されるため、パフォーマンスに悪影響を与えることになります。アプリケーションが浮動小数点をほとんど、またはまったく使用しない場合は、FLOAT(AFP(VOLATILE)) よりも FLOAT(NOAFP) を指定してコンパイルした方が高いパフォーマンスが得られます。

NOVOLATILE

コンパイラーは、(z/OS で推奨されているように) 呼び出されたプログラムが FPR8 から FPR15 までのレジスターを保持することを想定します。このオプションを使用すると最適なコードが生成されます。CICS コードを使用しない場合は、このオプションを強くお勧めします。ただし、CICS アプリケーションに組み込まれるコードの場合は、このオプションの使用には注意が必要です。CICS アプリケーションの場合は、次のどちらかの方法でコンパイルします。

- EXEC CICS ステートメントを含むすべてのコードは、FLOAT(AFP(VOLATILE)) を指定してコンパイルします。
- 浮動小数点を使用するすべてのコードは、FLOAT(NOAFP) または FLOAT(AFP(VOLATILE)) を指定してコンパイルします。

CICS アプリケーションの一部として実行されるコードの場合は、FLOAT(AFP(NOVOLATILE)) オプションを使用しないことが、安全でおそらく最も簡単な方法です。

DFP

DFP 機能が活用されます。すべての DECIMAL FLOAT データが「z/OS Principles of Operations」マニュアルに記載されている DFP フォーマットで保持され、DECIMAL FLOAT を使用する操作は、同マニュアルに記載されている DFP ハードウェア命令を使用して実行されます。

ARCH オプションは 7 (またはそれ以上) でなければなりません。そうでない場合、このオプションは拒否されます。

NODFP

DFP 機能は活用されません。

FLOAT(DFP) を指定すると、

- 拡張 DECIMAL FLOAT の最大精度は 34 (16 進浮動小数点と同様に 33 ではない) になります。
- 短精度 DECIMAL FLOAT の最大精度は 7 (16 進浮動小数点と同様に 6 ではない) になります。
- 次の組み込み関数での DECIMAL FLOAT の値は、すべて適切に変更されます。
 - EPSILON
 - HUGE
 - MAXEXP
 - MINEXP
 - PLACES
 - RADIX
 - TINY

- 次の組み込み関数はすべて DECIMAL FLOAT の適切な値を返します (また、人間がはるかに容易に理解できる値を返します。例えば、SUCC(1D0) は 1.000_000_000_000_001 になり、ROUND 関数は小数点以下の桁で丸めます)。
 - EXPONENT
 - PRED
 - ROUND
 - SCALE
 - SUCC
- 10 進浮動小数点リテラルは、ゼロ以外の数字が小数点の後に続かないように、必要に応じて「右側ユニット表示」に変換された場合、つまり、指数が調整された場合 (例えば、3.1415E0 を 31415E-4 として表示する場合に行われるように)、当該リテラルの精度に対応した、正常な値の範囲にある指数を持つ必要があります。この範囲は、MINEXP-1 および MAXEXP-1 の値によって定められます。特に、以下を適用する必要があります。
 - 短精度の浮動小数点の場合、 $-95 \leq \text{指数} \leq 90$
 - 長精度の浮動小数点の場合、 $-383 \leq \text{指数} \leq 369$
 - 拡張精度の浮動小数点の場合、 $-6143 \leq \text{指数} \leq 6111$
- DECIMAL FLOAT が CHARACTER に変換される場合、ストリングは、指数に 4 桁を保持します (対して 16 進浮動小数点の場合は 2 桁使用されます)。
- IEEE および HEXADEC 属性は、FLOAT BIN に適用される場合のみ受け入れられ、DEFAULT(IEEE/HEXADEC) オプションは FLOAT BIN にのみ適用されます。
- 数学的な組み込み関数 (ACOS、COS、SQRT など) は、DECIMAL FLOAT の引数に対応しますが、これらの関数に対する言語環境サポートが不完全であるため、引数を IEEE BINARY FLOAT に変換し、対応する IEEE BINARY FLOAT 数学ルーチンを呼び出してから、この結果をまた DECIMAL FLOAT に変換します。同じ理由で、DECIMAL FLOAT 指数は同様の方法で処理されます。
- DFP を使用する場合は、一方のオペランドが FLOAT DECIMAL で他方がバイナリー (つまり、FIXED BINARY、FLOAT BINARY、または BIT) である場合の演算で生じる変換に留意する必要があります。そうした演算では、PL/I 言語の規則により、FLOAT DECIMAL オペランドは FLOAT BINARY に変換され、その変換にはライブラリー呼び出しが必要となります。したがって、例えば、 $A = A + B$; の形の代入で、 A が FLOAT DECIMAL で B が FIXED BINARY の場合は、3 つの変換が行われ、そのうち 2 つはライブラリー呼び出しになります。
 1. A は、ライブラリー呼び出しにより、FLOAT DECIMAL から FLOAT BINARY に変換されます。
 2. B は、インライン・コードにより、FIXED BINARY から FLOAT BINARY に変換されます。
 3. $A + B$ の和は、ライブラリー呼び出しにより、FLOAT BINARY から FLOAT DECIMAL に変換されます。

DECIMAL 組み込み関数は、次の場合に役立ちます。ステートメントが $A = A + DEC(B)$; に変更された場合、ライブラリー呼び出しが除去されます。ライブラリー呼び出しは、前もって B を FLOAT DECIMAL 一時変数に割り当て、それを A に加算することにより除去できます。

- 組み込み関数 SQRTF では、DECIMAL FLOAT の引数のサポートがありません (マップ先にするのできるハードウェア命令がないため)。
- DFP は、CAST タイプ付き関数でサポートされません。

FLOATINMATH

FLOATINMATH オプションは、数学組み込み関数を呼び出すときに、コンパイラーが使用する精度を指定します。



ASIS

数学組み込み関数に対する引数が、long 型または拡張型浮動小数点精度を持つように強制されることはありません。

LONG

short 型浮動小数点精度の数学組み込み関数に対する引数が、最大 long 型浮動小数点精度に変換され、同じ最大 long 型浮動小数点精度の結果を出します。

EXTENDED

short または、long 型浮動小数点精度の数学組み込み関数に対する引数が、最大拡張浮動小数点精度に変換され、同じ最大拡張浮動小数点精度の結果を出します。

精度 p のついた FLOAT DEC 式では、 $p \leq 6$ の場合は short 型浮動小数点精度、 $6 < p \leq 16$ の場合は long 型浮動小数点精度、 $p > 16$ の場合は、拡張型浮動小数点精度です。

精度 p のついた FLOAT BIN 式では、 $p \leq 21$ の場合は short 型浮動小数点精度、 $21 < p \leq 53$ の場合は long 型浮動小数点精度、 $p > 53$ の場合は拡張型浮動小数点精度です。

最大拡張型浮動小数点精度は、プラットフォームによって決まります。

GOFF

GOFF オプションは、一般オブジェクト・ファイル・フォーマット (GOFF) でオブジェクト・ファイルを生成するようにコンパイラーに指示します。



GOFF および OBJECT オプションが有効になっている場合、コンパイラーは、オブジェクト・ファイルを GOFF フォーマットで生成します。

NOGOFF および OBJECT オプションが有効になっている場合、コンパイラーは、オブジェクト・ファイルを XOBJ フォーマットで生成します。

GOFF フォーマットは、S/370 オブジェクト・モジュール・フォーマットおよび拡張オブジェクト・モジュール・フォーマットを置き換えるものです。このフォーマ

ットによって、以前のフォーマットのさまざまな制限 (例: 16 MB のセクション・サイズ) が除去され、ロング・ネームおよび長い属性のネイティブ z/OS サポートなど、多くの便利な拡張機能が提供されるようになりました。GOFF は、XCOFF および ELF などの業界標準の一部の側面を取り込んでいます。

GOFF オプションを指定した場合、出力オブジェクトをバインドするバインダーを使用する必要があります。プリリンカーを使用して GOFF オブジェクトを処理することはできません。

GOFF オプションでは、以下のオプションはサポートされていません。

- COMMON
- NOWRITABLE(PRV)

注: GOFF およびファイル名が重複したソース・ファイルを使用した場合、リンカーは、エラーを出してコード・セクションの 1 つを破棄する可能性があります。このような場合には、NOCSECT を指定して CSECT オプションをオフにしてください。

GONUMBER

GONUMBER オプションを指定すると、コンパイラーは、ソース・プログラムの行番号をランタイム・メッセージに含めるための追加情報を生成します。



省略形: GN、NGN

あるいは、オフセット・アドレスを使用して行番号を導き出すこともできます。オフセット・アドレスはランタイム・メッセージにも、OFFSET オプションで生成されるテーブルか LIST オプションで生成されるアセンブラー・リストにも、常に含まれています。

GONUMBER の使用は TEST オプションの ALL および STMT サブオプションにより強制されます。

GOSTMT オプションがないことに注意してください。実行時に情報を生成し、エラーの発生個所を識別する唯一のオプションは、GONUMBER オプションです。

GONUMBER オプションが使用されているとき、ランタイム・エラー・メッセージの中の「ステートメント (statement)」という用語は、NUMBER コンパイラー・オプションによって使用される行番号を参照しますが、これは、STMT オプションの実行下であっても同じであることに注意してください。

GRAPHIC

GRAPHIC オプションを指定すると、ソース・プログラムに 2 バイト文字を入れることができます。16 進コードの '0E' および '0F' は、ソース・プログラム内のどこにあっても (コメントおよびストリング定数の中での出現も含めて) それぞれシフトアウトおよびシフトインの制御コードとして扱われます。



省略形: GR、NGR

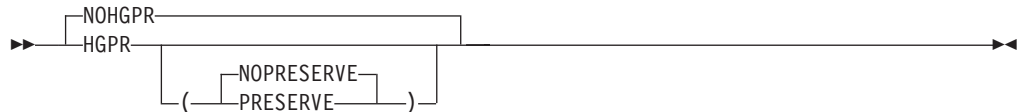
GRAPHIC オプションを指定すると、コンパイル中に使用されるどの STREAM ファイルにも GRAPHIC ENVIRONMENT オプションが適用されます。

ソース・プログラムで次のいずれかを使用する場合は、GRAPHIC オプションを指定しなければなりません。

- DBCS ID
- 漢字ストリング定数
- 混合ストリング定数
- ソース内のどこか別の場所にあるシフト・コード

HGPR

HGPR オプションは、z/Architecture ハードウェアをターゲットにした 32 ビット・プログラムで 64 ビット汎用レジスター (GPR) をコンパイラーが活用できることを指定します。



PRESERVE

PRESERVE は、関数のプロローグで保存して、エピローグで復元することによって、関数が使用する 64 ビット GPR の高位半分を保存するようにコンパイラーに指示します。PRESERVE サブオプションは、呼び出し元が Enterprise PL/I コンパイラーと z/OS XL C/C++ コンパイラーまたはそのいずれかによって生成されたコードで認識されていない場合にのみ必要です。

NOPRESERVE

NOPRESERVE によって、コンパイラーは、関数が使用する 64 ビット GPR の高位半分の保存を省略できます。パフォーマンスの考慮のため、HGPR のデフォルト・サブオプションは、NOPRESERVE です。

HGPR は、「High-half of 64-bit GPR (64 ビット GPR の高位半分)」のことで、ネイティブ 64 ビット命令を使用することを指します。特に、アプリケーションで 8 バイトの整数を使用する場合は、ネイティブ 64 ビット命令を活用することをお勧めします。

注: NOHGPR オプションは、すべての CICS および SQL アプリケーションで使用する必要があります。

INCAFTER

INCAFTER オプションでは、ソース・プログラムの中で特定のステートメントの後に組み込むファイルを指定します。

```

  ┌──INCAFTER──(──┐
  │               │
  │               └──PROCESS──(──filename──)──┐
  └──────────────────────────────────────────┘

```

filename

最後の PROCESS ステートメントの後に組み込むファイルの名前。

現在、PROCESS は唯一のサブオプションであり、最後の PROCESS ステートメントの後に組み込むファイルの名前を指定します。

次の例を考えてみてください。

```
INCAFTER(PROCESS(DFTS))
```

この例は、ソースの最後の PROCESS ステートメントの後にステートメント `%INCLUDE DFTS;` をコーディングするのと同等です。

INCDIR

INCDIR コンパイラー・オプションでは、インクルード・ファイルを位置指定するために使用される検索パスに追加されるディレクトリーを指定します。

```

  ┌──NOINCDIR──┐
  │            │
  └──INCDIR──(──'directory name'──)──┐
  └──────────────────────────────────┘

```

directory name

組み込みファイルを検索するディレクトリーの名前。INCDIR オプションを複数回指定できます。その場合、ディレクトリーは指定された順序で検索されます。

パッチの場合を除いて、コンパイラーは次の順序で INCLUDE ファイルを探します。

1. 現行ディレクトリー
2. -I フラグまたは INCDIR コンパイラー・オプションで指定されたディレクトリー
3. /usr/include ディレクトリー
4. INCPDS コンパイラー・オプションで指定された PDS

パッチの場合は、このオプションは恐らく、DFT(LOWERINC) オプションとともに最も使用され、形式「`%include x;`」のインクルードのみに影響を与えます。これらのインクルードでは、`x.inc` という名前の HFS ファイルはまず、このオプションで指定されたディレクトリーで検索されます。見つからなければ、`x` は、SYSLIB DD で指定された PDS(E) のメンバーである必要があります。形式「`%include dd(x);`」のインクルードでは、HFS ファイルはインクルードされません。メンバー「`x`」は常に、指定された DD で指定された PDS のメンバーである必要があります。

INCPDS

INCPDS オプションは、z/OS UNIX 環境でプログラムをコンパイルするときに、コンパイラーが組み込むファイルがある PDS を指定します。

注: このオプションは、z/OS UNIX 環境でのコンパイルにだけ適用されます。



PDS name

PDS の名前で、そこからファイルが組み込まれます。

例えば、SOURCE.PLI という名前の PDS からプログラム TEST をコンパイルしたい場合で、PDS SOURCE.INC からの INCLUDE ファイルを使用したい場合には、以下のコマンドを指定することができます。

```
pli -c -qincpds="SOURCE.INC" "'/'SOURCE.PLI(TEST)'"
```

コンパイラは次の順序で INCLUDE ファイルを探します。

1. 現行ディレクトリー
2. -I フラグまたは INCDIR コンパイラ・オプションで指定されたディレクトリー
3. /usr/include ディレクトリー
4. INCPDS コンパイラ・オプションで指定された PDS

INITAUTO

INITAUTO オプションは、INITIAL 属性を指定せずに宣言されたすべての AUTOMATIC 変数に、INITIAL 属性を追加するようコンパイラに指示します。



INITAUTO を指定すると、コンパイラは、INITIAL 属性を持たない AUTOMATIC 変数に対して、そのデータ属性に従って、以下の INITIAL 属性を追加します。

- INIT((*) 0) - データ属性が FIXED または FLOAT の場合
- INIT((*) '') - データ属性が PICTURE、CHAR、BIT、GRAPHIC、または WIDECHAR の場合
- INIT((*) SYSNULL()) - データ属性が POINTER または OFFSET の場合

コンパイラは、その他の属性を持つ変数には INITIAL 属性を追加しません。

完全に初期化されていない AUTOMATIC 変数 (ただし、DFT(INITFILL) オプションと違って、これらの変数は意味のある初期値を持つようになりました) が入っている各ブロックごとに、プロログの中により多くのコードが INITAUTO により生成されるようになり、パフォーマンスに悪い影響を与えることになります。

INITAUTO オプションは、NOINIT 属性を指定して宣言された変数に対しては、INITIAL 属性を適用しません。

INITBASED

INITBASED オプションは、INITIAL 属性を指定せずに宣言されたすべての BASED 変数に、INITIAL 属性を追加するようコンパイラに指示します。



このオプションは、**BASED** 変数に対してであることを除き、**INITAUTO** と同じ機能を実行します。

INITBASED オプションにより、完全に初期化されていない **BASED** 変数の **ALLOCATE** に対して、より多くのコードが生成されるようになり、パフォーマンスに悪い影響を与えることになります。

INITBASED オプションは、**NOINIT** 属性を指定して宣言された変数に対しては、**INITIAL** 属性を適用しません。

INITCTL

INITCTL オプションは、**INITIAL** 属性を指定せずに宣言されたすべての **CONTROLLED** 変数に、**INITIAL** 属性を追加するようコンパイラーに指示します。



このオプションは、**CONTROLLED** 変数に対してであることを除き、**INITAUTO** と同じ機能を実行します。

INITCTL オプションにより、完全に初期化されていない **CONTROLLED** 変数の **ALLOCATE** に対して、より多くのコードが生成されるようになり、パフォーマンスに悪い影響を与えることになります。

INITCTL オプションは、**NOINIT** 属性を指定して宣言された変数に対しては、**INITIAL** 属性を適用しません。

INITSTATIC

INITSTATIC オプションは、**INITIAL** 属性を指定せずに宣言されたすべての **STATIC** 変数に、**INITIAL** 属性を追加するようコンパイラーに指示します。



このオプションは、**STATIC** 変数に対してであることを除き、**INITAUTO** と同じ機能を実行します。

INITSTATIC オプションでは、一部に大きいオブジェクトを作成したり、長いコンパイルを作成する可能性はありますが、それ以外はパフォーマンスに影響を与えることはありません。

INITSTATIC オプションは、**NOINIT** 属性を指定して宣言された変数に対しては、**INITIAL** 属性を適用しません。

INSOURCE

INSOURCE オプションは、PL/I マクロ・プリプロセッサが変換できるよう、ソース・プログラムのリストをコンパイラーが組み込むことを指定します。



省略形: IS、NIS

FULL

INSOURCE リストは %NOPRINT ステートメントを無視し、プリプロセッサがソースを変換する前にすべてのソースがリストに組み込まれます。

FULL がデフォルトです。

SHORT

INSOURCE リストは %PRINT ステートメントと %NOPRINT ステートメントを区別します。

MACRO オプションが有効になっていない場合、INSOURCE リストの効果はありません。

INSOURCE オプションを指定すると、プログラムのロジックとは関係なく、各ファイルの読み取り順にテキストがリストに入れます。例えば、PROC と END の両ステートメント間に %INCLUDE ステートメントがある、次の単純なプログラムを考えてみましょう。

```
insource: proc options(main);  
    %include member;  
end;
```

INSOURCE リストには、ファイル "member" からインクルードされるテキストの前に、メインプログラム全体が入ります (また、ファイルによってインクルードされるテキストの前にそのファイル全体が入り、以下も同様)。

INSOURCE(SHORT) オプションを指定した場合、%INCLUDE ステートメントによってインクルードされるテキストは、%INCLUDE ステートメントの実行時に有効だった print/noprint 状況を継承しますが、その print/noprint 状況はインクルードされるテキストの終わりで復元されます (ただし SOURCE リスト内では、インクルードされるテキストの終わりで print/noprint 状況は復元されません)。

INTERRUPT

INTERRUPT オプションを指定すると、コンパイル済みプログラムがアテンション要求 (割り込み) に応答します。



省略形: INT、NINT

このオプションにより、コンパイル済み PL/I プログラムが対話式システムの下で実行されるときのアテンション割り込みの効果が決まります。このオプションは、TSO の下で実行されるプログラムに対してだけ効果があります。ATTENTION 条件の発生に依存するプログラムを作成した場合、そのプログラムは INTERRUPT オプションを使ってコンパイルしなければなりません。このオプションを使用すると、アテンション割り込みをプログラミングの不可欠な部分にすることができます。この方法により、プログラムを対話式で大幅に制御できます。

INTERRUPT オプションを指定すると、アテンション割り込みが生じた場合、確立されている ATTENTION ON ユニットが制御を得ます。ATTENTION ON ユニットの実行が完了すると、GOTO ステートメントにより別の場所が指定されていない限り、制御は割り込み点に戻ります。ATTENTION ON ユニットを確立していないと、アテンション割り込みは無視されます。

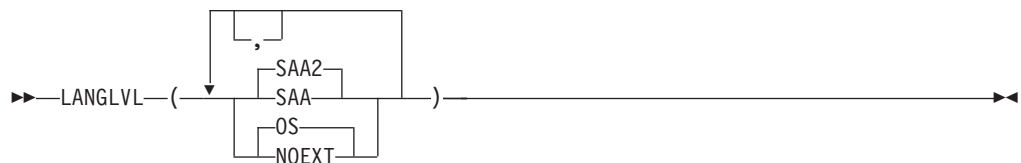
NOINTERRUPT を指定すると、プログラムの実行時のアテンション割り込みで、ATTENTION ON ユニットに制御が渡ることはありません。

テストの目的だけにアテンション割り込み機能が必要な場合は、INTERRUPT オプションの代わりに TEST オプションを使用してください。詳しくは 78 ページの『TEST』を参照してください。

プログラムでの割り込みの使用法の詳細については、453 ページの『第 18 章 割り込みとアテンションの処理』を参照してください。

LANGLVL

LANGLVL オプションでは、コンパイラーに受け入れさせたい PL/I 言語定義のレベルを指定します。



SAA

コンパイラーは、OS PL/I バージョン 2 リリース 3 でサポートされていないキーワードとその他の言語構造体にフラグを付け、OS PL/I バージョン 2 リリース 3 でサポートされていない組み込み関数を認識しません。

SAA2

コンパイラーは、「PL/I 言語解説書」に記載されている PL/I 言語定義を受け入れます。

NOEXT

受け入れられる ENVIRONMENT オプションは次のとおりです。

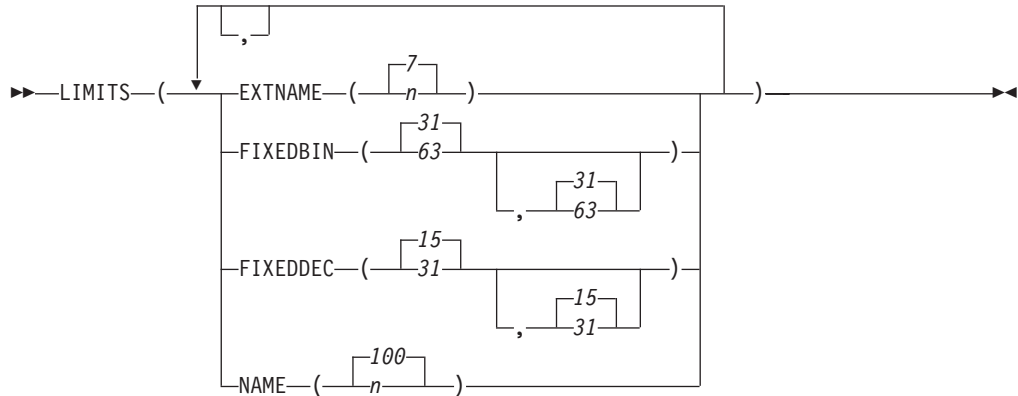
Bkwd	Genkey	Keyloc	Relative
Consecutive	Graphic	Organization	Scalarvarying
Ctlasa	Indexed	Recsize	Vsam
Deblock	Keylength	Regional	

OS

すべての ENVIRONMENT オプションが使用できます。すべての ENVIRONMENT オプションが 209 ページの表 13 にリストされています。

LIMITS

LIMITS オプションでは、各種のインプリメンテーションの制限を指定します。



EXTNAME

EXTERNAL 名の最大長を指定します。n の最大値は 100、最小値は 7 です。

FIXEDDEC

FIXED DECIMAL の最大精度として 15 または 31 のいずれかを指定します。デフォルトは 15 です。

FIXEDDEC(15,31) を指定した場合は、15 よりも大きい精度を指定した FIXED DECIMAL 変数を宣言できますが、式に 15 よりも大きい精度のオペランドが含まれていないかぎり、すべての算術演算は、最大精度として 15 を使用して行われます。

FIXEDDEC(15,31) は、FIXEDDEC(31) よりもかなり良いパフォーマンスを提供します。

FIXEDDEC(15) と FIXEDDEC(15,15) は等価であり、同様に FIXEDDEC(31) と FIXEDDEC(31,31) も等価です。

FIXEDDEC(31,15) は指定できません。

FIXEDBIN

SIGNED FIXED BINARY の最大精度として 31 または 63 を指定します。デフォルトは 31 です。

FIXEDBIN(31,63) を指定した場合は、8 バイト整数を宣言できますが、式に 8 バイト整数が含まれていない場合、整数算術演算はすべて 4 バイト整数を使用して行われます。

ただし、FIXEDBIN(31,63) または FIXEDBIN(63) オプションを指定すると、コンパイラーはデータ・タイプが混在する式に、8 バイト整数算術計算を使用する場合があります。例えば FIXED BIN(31) の値が FIXED DEC(13) の値に加算される場合、コンパイラーは FIXED BIN の結果を生成し、LIMITS(FIXEDBIN(31,63)) が指定されていると、その結果の精度は 31 より大

きくなります (FIXED DEC の精度が 9 より大きいため)。この状況が発生すると、コンパイラーは通知メッセージ IBM2809 を発行します。

FIXEDBIN(31,63) は、FIXEDBIN(63) よりもかなり良いパフォーマンスを提供します。

FIXEDBIN(31) と FIXEDBIN(31,31) は等価であり、同様に FIXEDBIN(63) と FIXEDBIN(63,63) も等価です。

FIXEDBIN(63,31) は指定できません。

UNSIGNED FIXED BINARY の最大精度は、1 を加えた数、つまり 32 または 64 です。

NAME

プログラムの中の変数名の最大長を指定します。 n の最大値は 100、最小値は 31 です。

LINECOUNT

LINECOUNT オプションは、コンパイラー・リストのページ当たりの行数 (ブランク行と見出し行を含む) を指定します。

▶▶—LINECOUNT—(\overbrace{n}^{60})—▶▶

省略形: LC

n リストの 1 ページの行数。値の範囲は 10 から 32,767 までです。

LINEDIR

このオプションは、%LINE ディレクティブをコンパイラーが受け入れるようにするかどうかを指定します。

▶▶— $\overbrace{\text{NLINEDIR}}^{\text{NLINEDIR}}$
—LINEDIR—▶▶

LINEDIR オプションが指定されると、コンパイラーはすべての %INCLUDE ステートメントを拒否します。LINEDIR オプションが指定されると、コンパイラーは TEST オプションの SEPARATE サブオプションの使用も拒否します。

LIST

LIST オプションは、コンパイラーが疑似アセンブラー・リストを生成するように指定します。

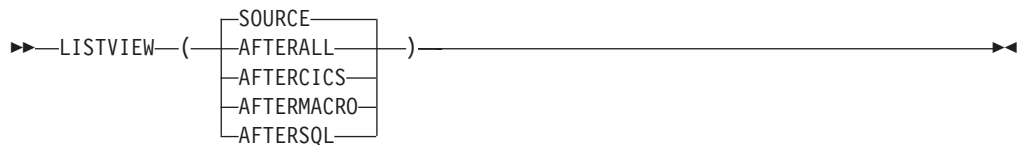
▶▶— $\overbrace{\text{NOLIST}}^{\text{NOLIST}}$
—LIST—▶▶

LIST オプションを指定すると、コンパイル時に必要な時間と領域が増加します。OFFSET と MAP オプションは、ごくわずかのコストで必要な情報を提供します。

各ブロックごとに、疑似アセンブラー・リストも、リストの終わりに、全コンパイル単位の開始位置からそのブロック内の最初の命令までのオフセットで組み込まれます。

LISTVIEW

LISTVIEW オプションは、コンパイラーがソース・リストでソースを表示するかどうか、または 1 つ以上のプリプロセッサで処理された後にソースを表示するかどうかを指定します。



SOURCE

ソース・リストで生のソースを表示します。また、さらに重要なこととして、デバッグ・ツールでこれをソース・ビューとして表示します。

AFTERALL

最後のプリプロセッサの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツールでこれをソース・ビューとして立ち上げます。

AALL は、AFTERALL の省略形として使用されることがあります。

AFTERCICS

CICS プリプロセッサの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツールでこれをソース・ビューとして立ち上げます。

ACICS は、AFTERCICS の省略形として使用されることがあります。

AFTERMACRO

MACRO プリプロセッサの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツールでこれをソース・ビューとして立ち上げます。

AMACRO は、AFTERMACRO の省略形として使用されることがあります。

AFTERSQL

SQL プリプロセッサの最後の呼び出し (存在する場合) の MDECK 由来のソースとしてソース・リストで表示します。また、さらに重要なこととして、TEST コンパイラー・オプションの SEPARATE サブオプションも指定されている場合は、デバッグ・ツールでこれをソース・ビューとして立ち上げます。

ASQL は、AFTERSQL の省略形として使用されることがあります。

TEST オプションを指定して、LISTVIEW に SOURCE 以外のサブオプションを指定した場合には、TEST オプションに SEPARATE サブオプションも指定する必要があります。

AFTERMACRO、AFTERSQL、および AFTERALL サブオプションの異なる影響の例として、PP オプションが PP(MACRO('INCONLY'), SQL, MACRO) であったと仮定してください。そうすると、

- LISTVIEW(AFTERMACRO) では、TEST(SEP) が指定されている場合、リストおよび「デバッグ・ツール・ソース (Debug Tool source)」ウィンドウの「ソース」が、MACRO プリプロセッサの 2 番目の呼び出しが生成した MDECK からきたかのように表示されます。
- LISTVIEW(AFTERSQL) では、TEST(SEP) が指定されている場合、リストおよび「デバッグ・ツール・ソース (Debug Tool source)」ウィンドウの「ソース」は、SQL プリプロセッサの呼び出しが生成した MDECK からきたかのように表示されます (したがって、%DCL およびその他のマクロ・ステートメントはまだ表示されます)。
- LISTVIEW(AFTERALL) では、MACRO プリプロセッサが PP オプションの最後であるため、「ソース」は LISTVIEW(AFTERMACRO) オプションの下にきます。

MACRO

MACRO オプションは MACRO プリプロセッサを呼び出します。



省略形: M、NM

PP(MACRO) オプションを介して MACRO プリプロセッサを呼び出すこともできます。PP オプションについて、詳しくは 57 ページの『PP』を参照してください。

MACRO プリプロセッサについて詳しくは、107 ページの『マクロ・プリプロセッサ』を参照してください。

MAP

MAP オプションを指定すると、ダンプ内の静的変数と自動変数を見つけるために使用できる追加情報が、コンパイラによって生成されます。



MARGINI

MARGINI オプションでは、INSOURCE オプションおよび SOURCE オプションで生成されたリストの左側のマージンの前の桁と、右側のマージンの後の桁に、コンパイラが置く文字を指定します。



省略形: MI、NMI

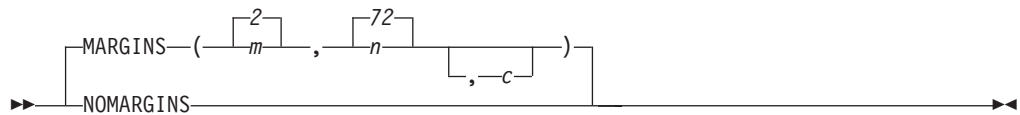
c マージン標識として印刷される文字。

注: NOMARGINI は MARGINI(' ') と同等です。

MARGINS

MARGINS オプションは、各コンパイラー入力レコードのどの部分に PL/I ステートメントを入れるかを指定します。また、SOURCE オプションまたは INSOURCE オプションが適用される場合は、リストをフォーマット設定する ANS 制御文字の位置も指定します。コンパイラーは、これらの限界外にあるデータは処理ませんが、ソース・リストには入れます。

PL/I ソースがソース入力レコードから取り出される際、レコードの最初のデータ・バイトが、その直前のレコードの最後のデータ・バイトのすぐ後にくるように取り出されます。変数レコードの場合、ブランクが必要であれば、必ず各レコードのマージン間に明示的にブランクを挿入するようにしなければなりません。



省略形: MAR

m コンパイラーによって処理される左端の文字 (最初のデータ・バイト) の桁番号。これは、100 を超えてはなりません。

n コンパイラーによって処理される右端の文字 (最後のデータ・バイト) の桁番号。これは *m* より大きくなければならず、200 を超えてはなりません。ただし MVS パッチ環境では、100 を超えてはなりません。

可変長レコードは、最大レコード長になるように効果的にブランクが埋め込まれます。

c ANS プリンター制御文字の桁番号。これは 200 を超えてはならず (ただし、MVS パッチ環境では 100 を超えてはならない)、かつ *m* と *n* に指定した値の範囲の外にあることが必要です。*c* の値として 0 を指定すると、ANS 制御文字がないことが示されます。次の制御文字だけを使用できます。

(ブランク)

1 行スキップしてから印刷する。

0 2 行スキップしてから印刷する。

- 3 行スキップしてから印刷する。

+ スキップしないで印刷する。

1 改ページする

これ以外の文字を使用するとエラーになり、ブランクに置き換えられます。

ソース・レコードの最大長より大きい値の *c* は使用しないでください。使用すると、リストのフォーマットが予測できないものになります。この問題を避けるには、可変長レコードのソース・マージンの左側に紙送り制御文字を置きます。

%PAGE や %SKIP ステートメントを使用する代わりの方法として、MARGINS(,c) を指定することもできます (「PL/I 言語解説書」で説明しています)。

固定長レコードの IBM 提供のデフォルトは MARGINS(2,72) です。可変長レコードと不定長レコードの IBM 提供のデフォルトは MARGINS(10,100) です。このデフォルトは、プリンター制御文字がないことを指定します。

プログラム内の 1 次入力のデフォルトを指定変更するには、MARGINS オプションを使用します。2 次入力のマージンは 1 次入力の場合と同じでなければなりません。

NOMARGINS オプションは、前に出現した MARGINS オプションのインスタンスを抑制します。このオプションの目的は、ご使用のシステムのコンパイル時間オプションをデフォルト設定にできるようにすることで、このコンパイル時間オプションは、変数ソース・フォーマット・ファイルが使用可能になっている間、固定フォーマット・ソース設定用に調整された MARGINS オプションを使用します。

コンパイラーに渡されたパラメーター・ストリングの一部として使用する場合は、通常、NOMARGINS オプションを指定します。このコンパイラーは、%PROCESS の中にオプションを見付けると、NOMARGINS を無視します。

MAXMEM

OPTIMIZE を指定してコンパイルする場合、MAXMEM オプションは、メモリーを多く消費する特定の最適化のローカル・テーブル用に使用されるメモリーの量を、指定したキロバイト数までに制限します。指定できる最小キロバイト数は 1 です。指定できる最大キロバイト数は 2097152 で、デフォルトは 1048576 です。

最大値の 2097152 を指定した場合、コンパイラーは無制限のメモリーが使用可能であることを想定します。最大値より小さい値を MAXMEM に指定した場合 (特に OPT(2) オプションが有効の場合) は、コンパイラーがメッセージを出すことがあります。このメッセージは、最適化が禁止されていることを示し、MAXMEM により大きな値を使用するようにユーザーに促します。

MAXMEM オプションは、使用可能なメモリーの量がデフォルト値によって暗黙指定される量より少ない (または多い) ことが分かっている場合に使用してください。

MAXMEM オプションに指定したメモリーが最適化に十分でない場合は、最適化の品質が低下した状態でコンパイルが完了し、警告メッセージが出されます。

▶▶—MAXMEM—(size)—▶▶

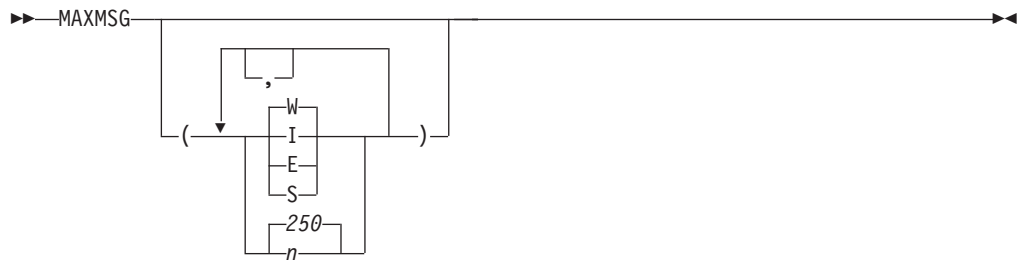
省略形: MAXM

MAXMEM に大きなサイズを指定した場合は、コンパイルされるソース・ファイル、ソース内のサブプログラムのサイズ、およびコンパイル用に使用できる仮想記憶域のサイズによっては、仮想記憶域の不足が原因でコンパイルが停止する場合があります。

MAXMEM オプションを使用する利点は、大規模で複雑なアプリケーションをコンパイルする場合に、コンパイラーが「仮想記憶域の不足」を示すエラー・メッセージを出してコンパイルを終了するのでなく、最適化品質の少し低下したオブジェクト・モジュールを作成して、警告メッセージを生成することです。

MAXMSG

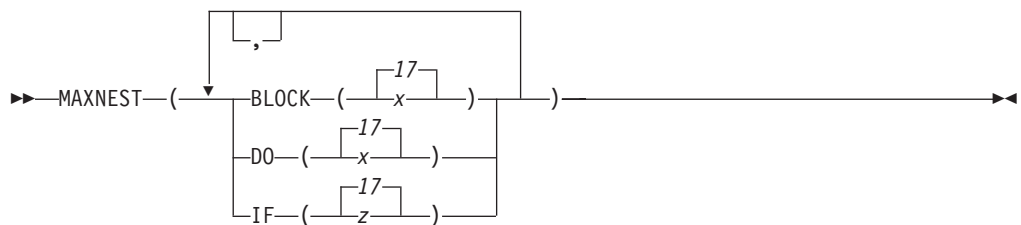
MAXMSG オプションは、コンパイル時に生成されるはずの指定された重大度（またはそれ以上）を持つメッセージの最大数を指定します。



- I** すべてのメッセージを数えます。
- W** 情報メッセージを除くすべてのメッセージを数えます。
- E** 警告メッセージと通知メッセージを除くすべてのメッセージを数えます。
- S** 重大エラー・メッセージおよび回復不能エラー・メッセージだけを数えます。
- n** メッセージの数がこの値を超えた場合、コンパイルを終了します。指定の重大度より低いメッセージ、またはコンパイラー出力ルーチンによりフィルターに掛けられて取り除かれたメッセージは、カウントされません。n の値の範囲は 0 から 32767 までです。0 を指定した場合、指定された重大度の最初のエラーが検出されるとコンパイルは終了します。

MAXNEST

MAXNEST オプションは、様々な種類のステートメントの最大ネストを指定し、これを超えると、コンパイラーはご使用のプログラムに対し、複雑すぎるということを示すフラグを立てます。



BLOCK

BEGIN および PROCEDURE ステートメントの最大ネストを指定します。

DO

DO ステートメントの最大ネストを指定します。

IF IF ステートメントの最大ネストを指定します。

すべてのネスト制限の値は、1 から 50 (その数を含む) です。

デフォルトは、MAXNEST(BLOCK(17) DO(17) IF(17)) です。

MAXSTMT

MAXSTMT オプションを指定すると、指定した数を超えるステートメントがあるブロックの最適化がオフになります。MAXSTMT オプションは、プログラムに対して生成されるコードを最適化する場合に、そのプログラム内で適度なサイズのブロックだけを最適化するようにコンパイラーに指示するために、適度なステートメント数の制限を指定して使用します。

▶▶—MAXSTMT—(*size*)—————▶▶

MAXSTMT に大きなサイズを指定した場合に、多数のステートメントを含むブロックがあると、十分な仮想記憶域が使用できなければコンパイルが打ち切られる場合があります。

MAXSTMT のデフォルトは 4096 です。

MAXTEMP

MAXTEMP オプションは、コンパイラー生成一時ステートメント用のストレージの量を非常に多く使用しているステートメントについて、コンパイラーがいつフラグを立てるかを判断します。

▶▶—MAXTEMP—(—*max*—)—————▶▶

max

コンパイラー生成一時ステートメントに使用できるバイト数の限度。 *max* で指定されたバイト数より多くのバイトを使用するステートメントは、コンパイラーによりフラグが付けられます。 *max* のデフォルト値は 50000 です。

このオプションでフラグのついたステートメントを調べてください。それらを別の方法でコード化すると、コードが必要とするスタック・ストレージの量が削減できる場合があります。

MDECK

MDECK オプションを指定すると、プリプロセッサは、z/OS の場合は SYSPUNCH DD ステートメントで定義されたファイルに、z/OS UNIX の場合は .dek ファイルに、プリプロセッサの出力のコピーを作成します。

▶▶—

NOMDECK
MDECK

—————▶▶

省略形: MD、NMD

MDECK オプションを使用すると、プリプロセッサの出力を 80 桁のレコードのファイルとして保持できます。このオプションは、MACRO オプションの使用中にだけ使用することができます。

NAME

NAME オプションは、コンパイラによって作成される TEXT ファイルに NAME レコードを入れるように指定します。



省略形: N

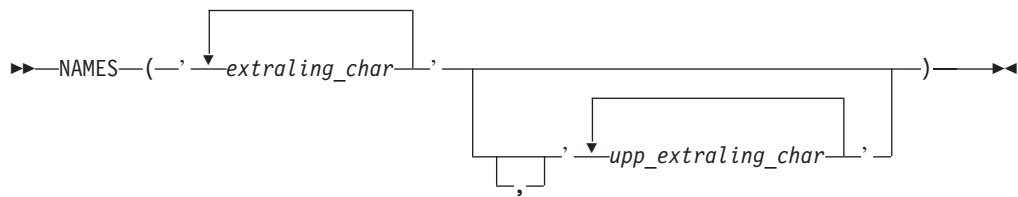
NAME オプションのサブオプションとして 'name' を指定しない場合、使用される 'name' は次のように決定されます。

- PACKAGE ステートメントがある場合は、そのステートメントの左端にある名前が使用される。
- そうでない場合には、最初の PROCEDURE ステートメントの左端にある名前が使用される。

$n \leq 8$ を指定した LIMITS(EXTNAME(n)) オプションを使用している場合、'name' の長さは 8 文字を超えてはなりません。

NAMES

NAMES オプションでは、ID に使用できる特別言語文字 を指定します。特別言語文字とは、「PL/I 言語解説書」で定義されている特殊文字、26 個の英字、および 10 個の数字以外の文字です。



extralingual_char

特別言語文字。

upp_extralingual_char

最初のサブオプションで指定した文字に対応する大文字として解釈させる特別言語文字。

2 番目のサブオプションを省略すると、PL/I は最初のサブオプションで指定された文字を小文字と大文字の両方として使用します。2 番目のサブオプションを指定する場合は、最初のサブオプションで指定したのと同じ数の文字を指定しなければなりません。

デフォルトは NAMES('#@\$' '#@\$') です。

NATLANG

NATLANG オプションは、コンパイラ・メッセージ、ヘッダーなどの「言語」を指定します。



ENU

コンパイラー・メッセージ、ヘッダーなどはすべて大文字小文字混合の英語である。

UEN

コンパイラー・メッセージ、ヘッダーなどはすべて大文字の英語である。

NEST

NEST オプションを指定すると、SOURCE オプションの実行結果のリストに、各ステートメントごとのブロック・レベルと do グループ・レベルが示されます。



NOT

NOT オプションでは、論理否定演算子として使用できる代替記号を最大 7 つ指定します。



char

単一の SBCS 文字。

標準の論理否定記号 (¬) の場合を除き、英字、数字、および「PL/I 言語解説書」に定義されている特殊文字はどれも指定できません。少なくとも有効な文字を 1 文字指定する必要があります。

NOT オプションを指定すると、標準 NOT 記号は、文字ストリング内の 1 つの文字として指定しない限り認識されなくなります。

例えば、NOT('~) と指定すると、波形記号 X'A1' が論理否定演算子として認識され、標準 NOT 記号 (¬) X'5F' は認識されません。同様に、NOT (~¬) は波形記号または標準 NOT 記号のどちらかが論理否定演算子として認識されることを意味します。

NOT 記号用の IBM 提供のデフォルト・コード・ポイントは、X'5F' です。論理否定記号は、キーボード上では論理否定記号 (¬) または脱字記号 (^) として表記されていることがあります。

NUMBER

番号オプションは、ソース・プログラム内のステートメントが、ステートメントの派生元のファイルの行番号とファイル番号によって識別されることを指定し、この番号のペアを使用して、AGGREGATE、ATTRIBUTES、LIST、MAP、OFFSET、SOURCE および XREF オプションから作成されるコンパイラー・リスト内のステ

ートメントが識別されることを指定します。リストの終わりのファイル参照テーブルでは、コンパイル時に読み取られるそれぞれの入力ファイルに割り当てられた番号を示します。



プリプロセッサを使用している場合は、ソース・リストの複数の行を同じ行番号およびファイル番号で識別する場合があることに注意してください。例えば、ほとんどすべての EXEC CICS ステートメントがソース・リスト内で数行のコードを生成しますが、これらのコードはすべて単一の行およびファイル番号で識別されます。

LIST オプションで生成される疑似アセンブラー・リストでは、最初のファイルのファイル番号はブランクのままになることに注意してください。

デフォルトは NUMBER です。

OBJECT

OBJECT オプションは、コンパイラーがオブジェクト・モジュールを作成することを指定します。バッチ z/OS のもとでは、コンパイラーが SYSLIN DD によって定義されたデータ・セットにオブジェクトを保管し、z/OS UNIX のもとでは、コンパイラーが .o ファイルを作成します。



省略形: OBJ、NOBJ

NOOBJECT オプションを指定すると、コンパイラーはオブジェクト・モジュールを作成しません。しかし、NOOBJECT オブジェクトを指定すると、コンパイラーがすべての未初期化変数の検出だけでなく構文セマンティック解析フェーズも完了するので、NOCOMPILE、NOSEMANTIC、または NOSYNTAX オプションを指定した場合よりも多くのメッセージを作成できます。

NOOBJECT オプションを指定すると、LIST、MAP、OFFSET、および STORAGE オプションは無視されます。

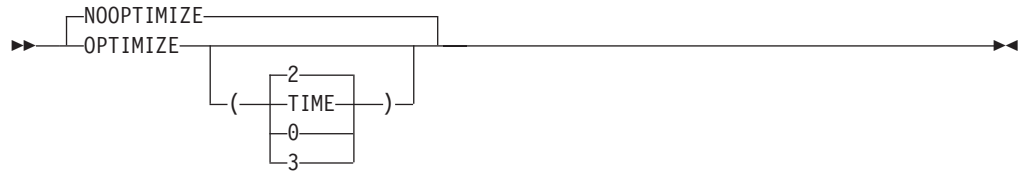
OFFSET

OFFSET オプションは、コンパイラーがそれぞれのプロシージャと BEGIN ブロックについて、プロシージャの 1 次エントリー・ポイントから相対的なオフセット・アドレスを付けて、行番号のテーブルを表示することを指定します。このテーブルは、GONUMBER オプションが使用されていない時に、ランタイム・エラー・メッセージからステートメントを識別するために使用できます。



OPTIMIZE

OPTIMIZE オプションでは、必要な最適化のタイプを指定します。



省略形: OPT、NOPT

OPTIMIZE(0)

高速コンパイルを指定しますが、最適化は禁止します。

OPTIMIZE(2)

さらに効率のよいオブジェクト・プログラムが作成されるために、生成された機械命令を最適化します。このタイプの最適化により、オブジェクト・モジュールに必要な主記憶域の大きさを削減することもできます。

OPTIMIZE(3)

OPTIMIZE(2) のもとで行われたすべての最適化に加えて、追加の最適化を実行します。OPTIMIZE(3) では、コンパイラは通常、特に大容量ブロックで多くの変数を持つプログラムに対し、より小さく効率のよいオブジェクト・コードを生成します。しかし、OPTIMIZE(3) を使用してコンパイルを完了することは、OPTIMIZE(2) を使用した場合よりもかなり多くの時間および領域を必要とすることにもなります。

OPTIMIZE オプションと一緒に、DFT(REORDER) オプションを使用することを強くお勧めします。実際、次の条件がすべて当てはまる場合は、PROCEDURE ブロックまたは BEGIN ブロックに対する OPTIMIZE の効果は大幅に限定されます。

- ORDER オプションがブロックに適用されている。
- ハードウェアによって検出される条件 (ZERODIVIDE など) に対応する ON ユニットがブロックに含まれている。
- ブロックに、これらの ON ユニットからの分岐のターゲットになる (可能性がある) ラベルがある。

OPTIMIZE(2) を指定すると、NOOPTIMIZE の場合よりコンパイル時間が大幅に増えることがあり、所要スペースが大幅に増えることがあります。例えば、OPTIMIZE(2) を指定して大規模なプログラムをコンパイルするには数分かかる場合があります、75M 以上の領域が必要になる可能性があります。

OPTIMIZE(3) の使用により、OPTIMIZE(2) を使用する場合よりもコンパイルに必要な時間と領域が増加します。大規模なプログラムの場合、OPTIMIZE(3) でのプログラムのコンパイル時間が OPTIMIZE(2) の 2 倍より大きくなる可能性があります。

コンパイラは最適化の際にコードを移動して実行時効率を上げる場合があります。その結果、プログラム・リストの中のステートメント番号が、ランタイム・メッセージで使用するステートメント番号と対応しなくなることがあります。

NOOPTIMIZE は OPTIMIZE(0) と同等です。

OPTIMIZE(TIME) は、OPTIMIZE(2) と同等です。

OPTIMIZE(2) または OPTIMIZE(3) を使用すると、TEST オプションの機能が大きく制限されるので注意してください。次に例を示します。

- TEST の HOOK サブオプションを有効にした場合は、ブロック・フックのみが生成されます。
- TEST の NOHOOK サブオプションを有効にした場合は、変数をリストまたは変更しようとして失敗する場合があります (変数が最適化されてレジスターに入っていることがあるため)。また、特定のステートメントで停止しようとする、デバッガーが何度か停止する場合があります (ステートメントが複数の部分に分割されていることがあるため)。

コードのパフォーマンス改善に最適なオプションの選択の詳細については、309 ページの『第 11 章 パフォーマンスの向上』を参照してください。

OPTIONS

OPTIONS オプションは、このコンパイル中に使用されるコンパイラ・オプションを示したリストを、コンパイル・リスト内にコンパイラが組み込むことを指定します。



省略形: OP、NOP

このリストには、デフォルトで適用されたすべてのオプション、EXEC ステートメントの PARM パラメーターまたは呼び出しコマンド (pli) で指定されたオプション、%PROCESS ステートメントで指定されたオプション、z/OS の下で IBM_OPTIONS 環境変数で指定されたオプション、および任意のオプション・ファイルから取り込まれたすべてのオプションが含まれます。

OPTIONS(DOC) を指定すると、OPTIONS リストには、コンパイラがリリースされた時点でこの文書に記述されたオプション (およびサブオプション) のみが組み込まれます。

OPTIONS(ALL) を指定すると、OPTIONS リストには、コンパイラがリリースされた後、PTF によって追加されたオプションも組み込まれます。

OR

OR オプションでは、論理 OR 演算子として最大 7 つの代替記号を指定します。これらの記号は連結演算子としても使用されます。連結演算子は 2 つの連続した論理和記号と定義されます。



注: 引用符の間にブランクをコードしないでください。

OR 記号 (I) の IBM 提供のデフォルト・コード・ポイントは X'4F' です。

char

単一の SBCS 文字。

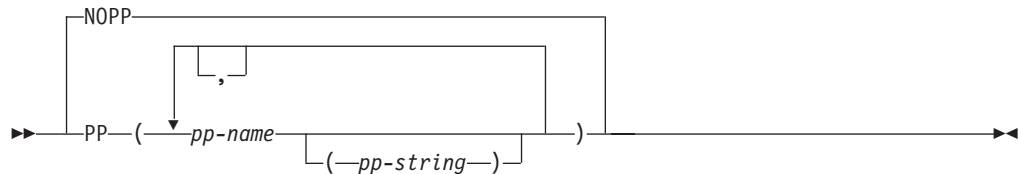
標準の論理 OR 記号 (I) を除き、英字、数字、および「PL/I 言語解説書」に定義されている特殊文字はどれも指定できません。少なくとも有効な文字を 1 文字指定する必要があります。

OR オプションを指定すると、標準 OR 記号は、文字ストリング内の 1 つの文字として指定しない限り認識されなくなります。

例えば OR (V) を指定すると、バックスラッシュ文字 X'E0' が論理 OR 演算子として認識され、2 つの連続したバックスラッシュは連結演算子として認識されます。標準 OR 記号 I、X'4F' は、どちらの演算子としても認識されません。同様に OR (V) を指定すると、バックスラッシュまたは標準 OR 記号のどちらかが論理 OR 演算子として認識され、どちらか片方の記号または両方の記号を使って連結演算子を作成できます。

PP

PP オプションでは、コンパイルの前にどのプリプロセッサを (どのような順序で) 呼び出すかを指定します。



pp-name

特定のプリプロセッサに与えられた名前。現在サポートされているプリプロセッサは、CICS、INCLUDE、MACRO、および SQL だけです。未定義の名前を使用すると診断エラーの原因となります。

pp-string

対応するプリプロセッサのためのオプションを表す最大 100 文字のストリング (引用符で区切られる)。例えば、PP(MACRO('CASE(ASIS)')) は、オプション CASE(ASIS) を指定して MACRO プリプロセッサを呼び出します。

プリプロセッサ・オプションは左から右へ処理されます。2 つのオプションが対立する場合は、最後の (右端の) オプションが使用されます。例えば、オプション・ストリング 'CASE(ASIS) CASE(UPPER)' を指定して MACRO プリプロセッサを呼び出した場合は、オプション CASE(UPPER) が使用されます。

同じプリプロセッサを複数回指定でき、最大 31 のプリプロセッサ・ステップを指定できます。

MACRO オプションまたは PP(MACRO) オプションを指定すると、どちらの場合も、コンパイルの前にマクロ機能が呼び出されます。MACRO と PP(MACRO) を両方指定すると、マクロ機能は 2 回呼び出されます。

PP オプションを複数回指定した場合、コンパイラーはそれらを実質的に連結します。したがって、PP(SQL) PP(CICS) を指定することは、PP(SQL CICS) を指定するのと同じことになります。これは、PP(MACRO SQL('OPTIONS')) と PP(MACRO SQL('OPTIONS DATE(ISO)')) を指定すると、PP オプションは結果として PP(MACRO SQL('OPTIONS') MACRO SQL('OPTIONS DATE(ISO)')) になり、MACRO プリプロセッサと SQL プリプロセッサの両方が 2 回呼び出されることも意味します。この指定を、前の SQL オプションを指定変更しようとして行う場合は、プリプロセッサ・オプションを PP オプションではなく PPSQL オプションを使用して指定した方がよい場合があります (つまり、PP(MACRO SQL) PPSQL('OPTIONS DATE(ISO)') を指定する)。

プリプロセッサについて詳しくは、105 ページの『第 2 章 PL/I プリプロセッサ』を参照してください。

PPCICS

PPCICS オプションは、CICS プリプロセッサが呼び出される場合にそれに渡すオプションを指定します。



したがって、PPCICS('EDF') PP(CICS) を指定することは、PP(CICS('EDF')) を指定するのと同じことになります。

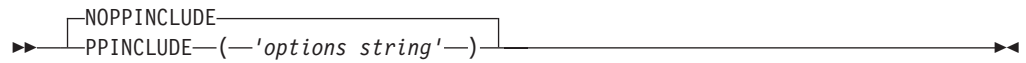
このオプションは、PP(CICS) オプションが指定されなければ有効ではありません。ただし、CICS プリプロセッサが呼び出されるときに使用される CICS プリプロセッサ・オプションのセットを指定する場合は、インストール・オプションの出口でこのオプションを指定することができます。こうすると、PP(CICS) を指定した時点で、PPCICS オプションで指定したオプションのセットが使用されるようになります。

また、PPCICS オプションで指定されたオプションは、プリプロセッサが呼び出されるときに指定されるオプションの形に変更されます。したがって、PPCICS('EDF') PP(CICS('NOEDF')) を指定することは、PP(CICS('EDF NOEDF'))、あるいはより単純な PP(CICS('NOEDF')) を指定するのと同じことになります。

オプション・string の長さは 1000 文字に制限されます。ただし、string は、100 文字を超える場合、オプション・リストに表示されません。

PPINCLUDE

PPINCLUDE オプションは、INCLUDE プリプロセッサが呼び出される場合にそれに渡すオプションを指定します。



したがって、PPINCLUDE('ID(-inc)') PP(INCLUDE) を指定することは、PP(INCLUDE('ID(-inc)')) を指定するのと同じことになります。

このオプションは、PP(INCLUDE) オプションが指定されなければ有効ではありません。ただし、INCLUDE プリプロセッサが呼び出されるときに使用される INCLUDE プリプロセッサ・オプションのセットを指定する場合は、インストール・オプションの出口でこのオプションを指定することができます。こうすると、PP(INCLUDE) を指定した時点で、PPINCLUDE オプションで指定したオプションのセットが使用されるようになります。

また、PPINCLUDE オプションで指定されたオプションは、プリプロセッサが呼び出されるときに指定されるオプションの形に変更されます。したがって、PPINCLUDE('ID(-inc)') PP(INCLUDE('ID(+include)')) を指定することは、PP(INCLUDE('ID(-inc) ID(+include)'))、あるいはより単純な PP(INCLUDE('ID(+include)')) を指定するのと同じことになります。

オプション・stringの長さは 1000 文字に制限されます。ただし、string は、100 文字を超える場合、オプション・リストに表示されません。

PPMACRO

PPMACRO オプションは、MACRO プリプロセッサが呼び出される場合にそれに渡すオプションを指定します。



したがって、PPMACRO('CASE(ASIS)') PP(MACRO) を指定することは、PP(MACRO('CASE(ASIS)')) を指定するのと同じことになります。

このオプションは、PP(MACRO) オプションが指定されなければ有効ではありません。ただし、MACRO プリプロセッサが呼び出されるときに使用される MACRO プリプロセッサ・オプションのセットを指定する場合は、インストール・オプションの出口でこのオプションを指定することができます。こうすると、MACRO または PP(MACRO) オプションを指定した時点で、PPMACRO オプションで指定したオプションのセットが使用されるようになります。

また、PPMACRO オプションで指定されたオプションは、プリプロセッサが呼び出されるときに指定されるオプションの形に変更されます。したがって、PPMACRO('CASE(ASIS)') PP(MACRO('CASE(UPPER)')) を指定することは、PP(MACRO('CASE(ASIS) CASE(UPPER)'))、あるいはより単純な PP(MACRO('CASE(UPPER)'))を指定するのと同じことになります。

オプション・stringの長さは 1000 文字に制限されます。ただし、string は、100 文字を超える場合、オプション・リストに表示されません。

PPSQL

PPSQL オプションは、SQL プリプロセッサが呼び出される場合にそれに渡すオプションを指定します。



したがって、PPSQL('ONEPASS') PP(SQL) を指定することは、PP(SQL('ONEPASS')) を指定するのと同じことになります。

このオプションは、PP(SQL) オプションが指定されなければ有効ではありません。ただし、SQL プリプロセッサが呼び出されるときに使用される SQL プリプロセッサ・オプションのセットを指定する場合は、インストール・オプションの出口でこのオプションを指定することができます。こうすると、PP(SQL) を指定した時点で、PPSQL オプションで指定したオプションのセットが使用されるようになります。

また、PPSQL オプションで指定されたオプションは、プリプロセッサが呼び出されるときに指定されるオプションの形に変更されます。したがって、PPSQL('ONEPASS') PP(SQL('TWO PASS')) を指定することは、PP(SQL('ONEPASS TWO PASS'))、あるいはより単純な PP(SQL('TWO PASS')) を指定するのと同じことになります。

オプション・string の長さは 1000 文字に制限されます。ただし、string は、100 文字を超える場合、オプション・リストに表示されません。

PPTRACE

PPTRACE オプションを指定すると、プリプロセッサ用にデック・ファイルが書き出されるとき、そのファイルの中の各非空白行の前に %LINE ディレクティブの行が追加されます。このディレクティブは、その非空白行が帰属するオリジナルのソース・ファイルと行を示します。



PRECTYPE

PRECTYPE オプションは、オペランドが FIXED であり、少なくとも 1 つが FIXED BIN である場合に、MULTIPLY、DIVIDE、ADD、および SUBTRACT 組み込み関数の属性をコンパイラが取り出す方法を決定します。



ANS

PRECTYPE(ANS) を指定すると、BIF(x, y, p) および BIF(x, y, p, 0) の中の値 p は 2 進数の桁数を指定しているものと解釈され、演算は 2 進演算として実行され、結果の属性は FIXED BIN(p,0) になります。

ただし、BIF(x, y, p, q) で q が非ゼロである場合、演算は 10 進演算として実行され、結果の属性は FIXED DEC(t,u) になります (ここで、t および u は、p および q の 10 進の等価値、つまり $t = \text{ceil}(p / 3.32)$ および $u = \text{ceil}(q / 3.32)$ です)。この場合、x、y、p、および q は、事実上、すべて 10 進数に変換されます (これは、x と y だけを変換し、しかも q がゼロであってもそのようにする、次のサブオプションとは対照的です)。この状態では、コンパイラーは通知メッセージ IBM1053 を出します。

DECDIGIT

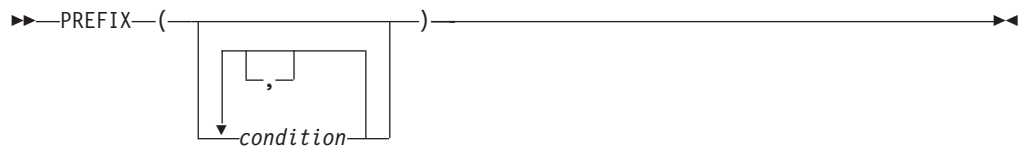
PRECTYPE(DECDIGIT) を指定すると、BIF(x, y, p) および BIF(x, y, p, 0) の中の値 p は 10 進数の桁数を指定しているものと解釈され、演算は 2 進演算として実行され、結果の属性は FIXED BIN(s) になります (ここで s は、p に対する対応する 2 進の等価値、つまり $s = \text{ceil}(3.32 * p)$ です)。q が非ゼロである BIF(x, y, p, q) のインスタンスでは、PRECTYPE(DECDIGIT) を指定した結果は、下記の PRECTYPE(DECRESULT) を指定した場合と同じです。

DECRESULT

PRECTYPE(DECRESULT) を指定すると、BIF(x, y, p) の中の値 p および BIF(x, y, p, q) の中の値 p と q は 10 進数の桁数を指定しているものと解釈され、演算は 10 進演算として実行され、結果の属性は、それぞれ FIXED DEC(p,0) または FIXED DEC(p,q) になります。結果は、DECIMAL 組み込み関数が x と y に適用された場合に生成されるものと同じになります。

PREFIX

PREFIX オプションを指定すると、ソース・プログラムの変更を必要とせずに、指定した PL/I 条件をコンパイル中のコンパイル単位の中で使用可能にしたり使用不可にしたりできます。指定した条件接頭語は、最初の PACKAGE ステートメントまたは PROCEDURE ステートメントの先頭に付けられます。



条件 (condition)

「PL/I 言語解説書」で説明されているように、PL/I プログラム内で使用可能/使用不可にできる任意の条件。

デフォルト PREFIX(CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE UNDERFLOW ZERODIVIDE)

PROCEED

PROCEED オプションを指定すると、前にプリプロセッサが発行したメッセージの重大度に応じて、プリプロセッサによる処理の完了後にコンパイラーが停止します。



省略形: PRO、NPRO

PROCEED

NOPROCEED(S) と同等です。

NOPROCEED

プリプロセッサがコンパイルを終えた後、処理を停止します。

NOPROCEED(S)

このプリプロセスの段階で重大エラーまたは回復不能エラーが検出された場合は、プリプロセッサおよびコンパイラーの呼び出しは継続されません。

NOPROCEED(E)

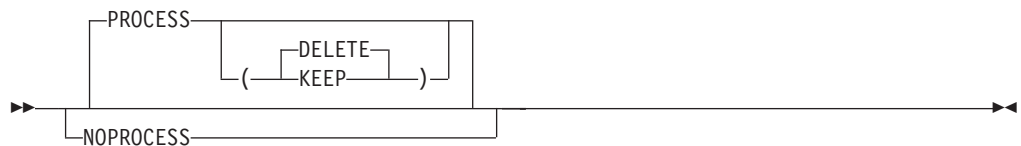
このプリプロセスの段階でエラー、重大エラー、または回復不能エラーが検出された場合は、プリプロセッサおよびコンパイラーの呼び出しは継続されません。

NOPROCEED(W)

このプリプロセスの段階で警告、エラー、重大エラー、または回復不能エラーが検出された場合は、プリプロセッサおよびコンパイラーの呼び出しは継続されません。

PROCESS

PROCESS オプションは、*PROCESS ステートメントが許可されるかどうか、および許可される場合に、それらのステートメントが MDECK ファイルに書き込まれるかどうかを決定します。



NOPROCESS オプションを指定すると、コンパイラーは、すべての *PROCESS ステートメントに E レベル・メッセージのフラグを立てます。

PROCESS(KEEP) オプションを指定すると、コンパイラーは *PROCESS ステートメントにフラグを立てることをせず、すべての *PROCESS ステートメントがコンパイラーによって MDECK 出力に保存されます。

PROCESS(DELETE) オプションを指定すると、コンパイラーは *PROCESS ステートメントにフラグを立てることをせず、どの *PROCESS ステートメントもコンパイラーによって MDECK 出力に保存されません。

QUOTE

QUOTE オプションでは、引用符文字として使用できる代替記号を最大 7 つ指定します。

▶▶QUOTE—(—'——'—)————▶▶

注: 引用符の間にブランクをコードしないでください。

QUOTE 記号用の IBM 提供のデフォルト・コード・ポイントは、''' です。

char

単一の SBCS 文字。

標準の QUOTE 記号 (") を除き、英字、数字、および「PL/I 言語解説書」に定義されている特殊文字はどれも指定できません。少なくとも有効な文字を 1 文字指定する必要があります。

QUOTE オプションは、GRAPHIC も指定されている場合は無視されます。

REDUCE

REDUCE オプションは、埋め込みバイトに上書きすることになったとしても、構造体へのヌル・ストリングの割り当てを減らしてより単純な操作にすることをコンパイラーに許可します。

また REDUCE オプションは、構造体に POINTER フィールドが含まれていても、一致する構造体の割り当てを減らして単純な集合移動にすることをコンパイラーに許可します。

▶▶————▶▶

NOREDUCE オプションを指定した場合、コンパイラーは構造体に対するヌル・ストリングの割り当てを分解して、構造体の基本メンバーに対してヌル・ストリングを連続して割り当てようになります。

REDUCE オプションを使用すると、ヌル・ストリングを構造体に割り当てるために生成されるコードの行数が少なくなり、その結果として通常はコンパイルが高速になり、コードの実行速度が大きく向上します。しかし、埋め込みバイトはゼロにリセットされることがあります。

例えば次の構造体では、*field12* と *field13* の間に 1 バイトの埋め込みがあります。

```
dc1
1 sample ext,
  5 field10      bin fixed(31),
  5 field11      bin fixed(15),
  5 field12      bit(8),
  5 field13      bin fixed(31);
```

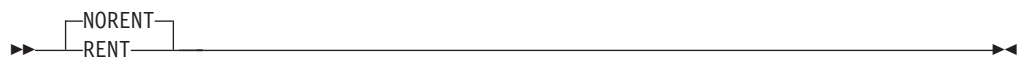
ここで、割り当て *sample = ''*; について考えてみます。

NOREDUCE オプションを指定した場合、4 つの割り当てが生成され、埋め込みバイトは変更されません。

一方 REDUCE を指定した場合、割り当てが減って 1 つの操作になりますが、埋め込みバイトはゼロにリセットされます。

NOREDUCE オプションを指定すると、コンパイラーは OS PL/I コンパイラーや PL/I for MVS コンパイラーに近い動作をします。これらのコンパイラーは、構造体に POINTER フィールドが含まれていなければ、一致する構造体の割り当てを減らして単純な集合移動にします。NOREDUCE オプションを指定すると、このコンパイラーは同じように動作するようになります。

RENT



コードが静的変数を変更しなければ、そのコードは「本質的に再入可能」です。

RENT オプションを指定すると、コンパイラーは本質的に再入可能でないコードを検出して、再入可能にします。再入可能性の詳細については、「z/OS 言語環境プログラム プログラミング・ガイド」を参照してください。RENT オプションを使用する場合、生成されたオブジェクト・モジュールをリンケージ・エディターによって直接処理することはできないので、プリリンカーまたは PDSE を使用する必要があります。

NORENT オプションを指定すると、コンパイラーは再入不可コードから再入可能コードを特に生成しません。本質的に再入可能なコードは、そのまま再入可能です。

RENT オプションを指定してコンパイルされたプログラムを 1 つ以上含むモジュール (MAIN または FETCHABLE のどちらか) をリンクする場合は、リンク・ステップで DYNAM=DLL および REUS=RENT を指定する必要があります。

NORENT および LIMITS(EXTNAME(*n*)) (*n* ≤ 7 を指定) オプションを指定した場合、コンパイラーによって生成されるテキスト・デックのフォーマットは、従来の PL/I コンパイラーによって生成されるものと同じです。このため、PDS スタイルのロード・モジュールの作成にプリリンカーは不要です。その他のオプションを使用する場合は、プリリンカーまたは PDSE のいずれかを使用しなければなりません。

NORENT オプションで生成されたコードは、NOWRITABLE オプションも指定していなければ再入可能にはなりません。

NORENT を使用すると、コンパイラーの機能の一部が使用できなくなります。特に、次の機能が使用できません。

- DLL は構築できません。
- 再入可能な書き込み可能静的変数はサポートされません。
- STATIC ENTRY VARIABLE は INITIAL 値を持つことはできません。

次の制約事項に従う RENT と NORENT コードは、混在できます。

- RENT を指定してコンパイルされたコードは、EXTERNAL STATIC 変数を共有している場合、NORENT でコンパイルされたコードと混在できません。
- RENT を指定してコンパイルされたコードは、NORENT でコンパイルされたコード内の ENTRY VARIABLE セットを呼び出すことができません。
- RENT を指定してコンパイルされたコードは、NORENT でコンパイルされたコードでフェッチされた ENTRY CONSTANT を呼び出すことができません。
- RENT を指定してコンパイルされたコードは、以下の条件のいずれかが当てはまれば、NORENT でコンパイルされたコードを含むモジュールをフェッチできます。
 - フェッチされたモジュールのすべてのコードが NORENT でコンパイルされている
 - モジュールへのエントリー・ポイントを含んでいるコードが RENT でコンパイルされている
- NORENT コードを指定してコンパイルされたコードは、RENT でコンパイルされた任意のコードを含むモジュールをフェッチできません。
- NORENT WRITABLE を指定してコンパイルされたコードは、任意の外部 CONTROLLED 変数または任意の外部 FILE を共有している場合、NORENT NOWRITABLE でコンパイルされたコードと混在できません。

上記の制約事項に従えば、以下を行うこともできます。

- 例えば *mnorent* という NORENT ルーチンと、*mrent* という RENT ルーチンを静的にリンクして呼び出します。
- RENT ルーチンの *mrent* は、その後、RENT でコンパイルされたエントリー・ポイントを備えて、別にリンクされたモジュールをフェッチして CALL します。

RESEXP

RESEXP オプションは、これによってある条件が発生してコンパイルが S レベル・メッセージで終了するとしても、コンパイラーがコンパイル時にすべての制限付き式を評価できるように指定します。





NORESEXP コンパイラー・オプションでは、コンパイラーは、INITIAL 値文節の式も含めて宣言されたすべての制限付き式を評価します。

例えば、NORESEXP オプションでは、コンパイラーは次のステートメントにフラグを立てません (ZERODIVIDE 例外が実行時に出されます)。

```
if preconditions_not_met then
  x = 1 / 0;
```

RESPECT

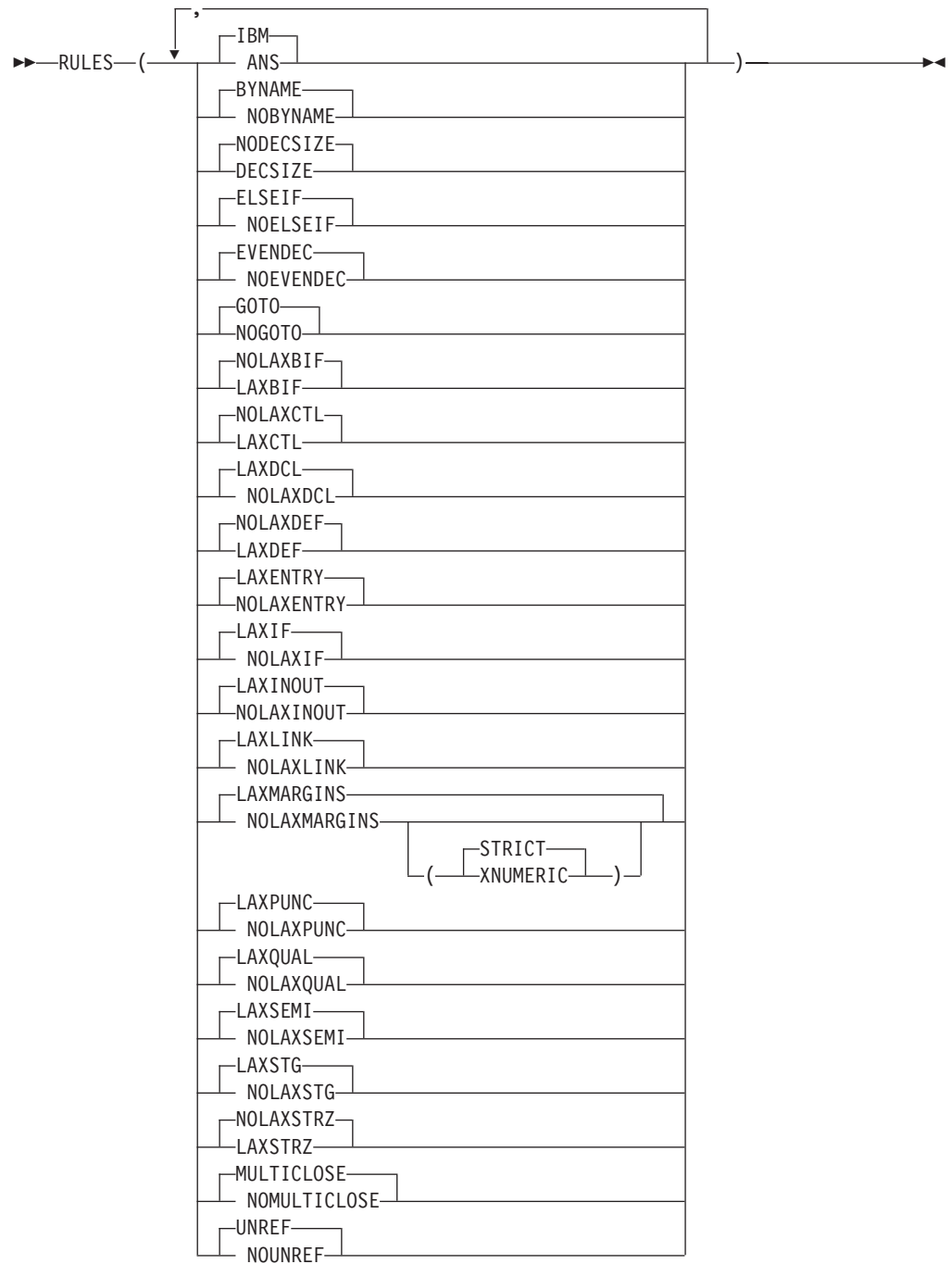
RESPECT オプションを指定すると、コンパイラーは DATE 属性のすべての指定を受け入れ、DATE 組み込み関数の結果に DATE 属性を適用します。

►►—RESPECT—(—)—

デフォルトの RESPECT() を使用すると、コンパイラーは DATE 属性の任意の指定を無視し、DATE 組み込み関数の結果に DATE 属性が適用されなくなります。

RULES

RULES オプションを指定すると、ある種の言語機能を使用可能または使用禁止にすることができ、代替の選択肢があればセマンティクスを選択できます。これは一般プログラミング・エラーの診断に役立ちます。



IBM | ANS

IBM サブオプションの場合:

- スtring・データを必要とする演算では、**BINARY** 属性を持つデータは **BIT** に変換されます。
- 算術演算または算術比較での変換は、「*PL/I* 言語解説書」で説明されているように行われます。

- ADD、DIVIDE、MULTIPLY、および SUBTRACT 組み込み関数の場合の変換は、スケールされた固定 2 進数として指定された演算がスケールされた固定 10 進数として評価されることを除けば、「PL/I 言語解説書」で説明されているように行われます。
- FIXED BIN 宣言では、ゼロ以外のスケール因数が使用できます。
- 精度処理組み込み関数 (ADD、BINARY、など) の結果が FIXED BIN 属性を持つ場合は、ゼロ以外のスケール因数を指定するか、暗黙指定することができます。
- MAX または MIN 組み込み関数の引数がすべて UNSIGNED FIXED BIN の場合でも、結果は常に SIGNED になります。
- 2 つの UNSIGNED FIXED BIN オペランドを用いて加算、乗算、または除算を行った場合でも、結果は SIGNED 属性を持ちます。
- 2 つの UNSIGNED FIXED BIN オペランドに MOD または REM 組み込み関数を適用した場合でも、結果は SIGNED 属性を持ちます。

ANS サブオプションの場合:

- スtring・データを必要とする演算では、BINARY 属性を持つデータは CHARACTER に変換されます。
- 算術演算または算術比較での変換は、「PL/I 言語解説書」で説明されているように行われます。
- ADD、DIVIDE、MULTIPLY、および SUBTRACT 組み込み関数の場合の変換は、「PL/I 言語解説書」で説明されているように行われます。
- ゼロ以外のスケール因数は、FIXED BIN 宣言では使用できません。
- 精度処理組み込み関数 (ADD、BINARY、など) の結果が FIXED BIN 属性を持つ場合は、指定または暗黙指定するスケール因数はゼロでなければなりません。
- MAX または MIN 組み込み関数の引数がすべて UNSIGNED FIXED BIN の場合でも、結果は常に UNSIGNED になります。
- 2 つの UNSIGNED FIXED BIN オペランドを用いて加算、乗算、または除算を行った場合でも、結果は UNSIGNED 属性を持ちます。
- 2 つの UNSIGNED FIXED BIN オペランドに MOD または REM 組み込み関数を適用した場合でも、結果は UNSIGNED 属性を持ちます。

また、RULES(ANS) では、旧コンパイラでは無視された、次のエラーにより、E レベル・メッセージが生成されます。

- String 定数を STRING 組み込み関数の引数として指定している。
- 配列参照内の添え字として指定しているアスタリスクが多すぎる。
- CONTROLLED 変数を POINTER 変数で修飾している (CONTROLLED 変数が BASED であったかのように)。

BYNAME | NOBYNAME

NOBYNAME を指定すると、コンパイラは E レベル・メッセージを伴うすべての BYNAME 割り当てにフラグを付けます。

DECSIZE | NODECSIZE

DECSIZE を指定すると、代入によって SIZE 条件が発生した場合に SIZE 条件

が使用不可であると、コンパイラーは FIXED DECIMAL 変数に対する FIXED DECIMAL 式のすべての代入にフラグを立てます。

RULES(DECSIZE) を指定すると、SIZE が使用不可であると、X が FIXED DECIMAL である場合に $X = X + 1$ という形式のすべてのステートメントにフラグが立てられることになるため、コンパイラーによって大量のメッセージが生成される場合があります。

ELSEIF | NOELSEIF

NOELSEIF を指定すると、直後に IF ステートメントが続く ELSE ステートメントに対しコンパイラーがフラグを立て、SELECT ステートメントとして書き直すように提案します。

一連のネストされた IF-THEN-ELSE ステートメントではなく SELECT ステートメントの使用を実施する場合に、このオプションを使用すると便利です。

EVENDEC | NOEVENDEC

NOEVENDEC を指定すると、コンパイラーは、偶数精度を指定するすべての FIXED DECIMAL 宣言にフラグを立てます。

GOTOINOGOTO

NOGOTO を指定すると、BEGIN ブロック外のすべての GOTO ステートメントにフラグが立てられます。

LAXBIF | NOLAXBIF

LAXBIF を指定すると、空のパラメーター・リストを使用しない場合でも、コンパイラーは NULL のようなコンテキスト宣言を組み込み関数のために作成します。

LAXCTL | NOLAXCTL

LAXCTL を指定すると、固定エクステントを使用して CONTROLLED 変数を宣言しても、CONTROLLED 変数を異なるエクステントに割り当てることができます。NOLAXCTL を指定すると、CONTROLLED 変数を異なるエクステントに割り当てる場合に、そのエクステントをアスタリスクとして指定するか、非定数式として指定する必要があります。

NOLAXCTL が指定されていると、次のコードは不正になります。

```
dc1 a bit(8) ctl;  
alloc a;  
alloc a bit(16);
```

ただし、次のコードは NOLAXCTL が指定されていても有効です。

```
dc1 b bit(n) ctl;  
dc1 n fixed bin(31) init(8);  
alloc b;  
alloc b bit(16);
```

LAXDCL | NOLAXDCL

LAXDCL を指定すると、暗黙宣言が可能になります。NOLAXDCL を指定すると、BUILTIN の場合と SYSIN および SYSPRINT ファイルの場合を除いて、暗黙宣言およびコンテキスト宣言はすべて禁止になります。

LAXDEF | NOLAXDEF

LAXDEF を指定すると、いわゆる無許可の定義が、コンパイラーのメッセージなしに受け入れられます (コンパイラーが通常作成する E レベル・メッセージも出ません)。

LAXENTRY | NOLAXENTRY

LAXENTRY を指定すると、非プロトタイプ・エンタリー宣言が許可されます。NOLAXENTRY を指定すると、コンパイラーは、すべての非プロトタイプ・エンタリー宣言 (つまり、パラメーター・リストを指定しないすべての ENTRY 宣言) のフラグを立てます。なお、これは、ENTRY にパラメーターがない場合には、単なる ENTRY としてではなく ENTRY() として宣言する必要があることを意味します。

LAXIF | NOLAXIF

RULES(NOLAXIF) を指定すると、コンパイラーは、属性 BIT(1) NONVARYING を持たないすべての IF、WHILE、UNTIL、および WHEN 文節にフラグを立てます。

NOLAXIF を指定すると、以下はすべてにフラグが立てられます。

```
dcl i fixed bin;  
dcl b bit(8);  
.  
.  
.  
if i then ...  
if b then ...
```

LAXINOUT | NOLAXINOUT

NOLAXINOUT を指定すると、コンパイラーはすべての ASSIGNABLE BYADDR パラメーターが入力 (かつ場合によっては出力) パラメーターであると想定するため、そのようなパラメーターが初期化されていないと、警告を出します。

LAXLINK | NOLAXLINK

NOLAXLINK を指定すると、以下のいずれかが一致しない場合に、コンパイラーは、2 つの ENTRY 変数または定数の代入または比較のすべてにフラグを立てます。

- パラメーター記述リスト

例えば、A1 が ENTRY(CHAR(8)) と宣言され、A2 が ENTRY(POINTER) VARIABLE と宣言されている場合、RULES(NOLAXLINK) を指定すると、コンパイラーは、A1 を A2 に代入しようとするフラグを立てます。

- RETURNS 属性

例えば、A3 が ENTRY RETURNS(FIXED BIN(31)) と宣言され、A4 が RETURNS 属性なしで ENTRY VARIABLE と宣言されている場合、RULES(NOLAXLINK) を指定すると、コンパイラーは、A3 を A4 に代入しようとするフラグを立てます。

- LINKAGE およびその他の OPTIONS サブオプション

例えば、A5 が ENTRY OPTIONS(ASM) と宣言され、A6 が OPTIONS 属性なしで ENTRY VARIABLE と宣言されている場合、RULES(NOLAXLINK) を指定すると、コンパイラーは A5 を A6 に代入しようとするフラグを立てます (A5 の宣言の中の OPTIONS(ASM) は A5 が LINKAGE(SYSTEM) を持つことが暗黙指定されるが、A6 には OPTIONS 属性がないため、デフォルトで LINKAGE(OPTLINK) を持つものと想定されるためです)。

LAXMARGINS | NOLAXMARGINS

NOLAXMARGINS を指定すると、STRICT および XNUMERIC サブオプションの設定に応じて、右マージンの後に非ブランク文字がある行にコンパイラーがフラグを立てます。このオプションは、誤って右マージンへ押し出された終了コメントなどのコードの検出に役立ちます。

いずれかのプリプロセッサと共に NOLAXMARGINS および STMT オプションを使用すると、NOLAXMARGINS オプションによってフラグを立てられるステートメントは、ステートメント番号ゼロとして報告されます (ステートメントの番号付けはすべてのプリプロセッサが終了した後でのみ行われますが、マージンの外側のテキストの検出はソースが読み取られるとすぐに行われるためです)。

STRICT

STRICT サブオプションを指定すると、右マージンの後に非ブランク文字がある行にコンパイラーがフラグを立てます。

XNUMERIC

XNUMERIC サブオプションを指定すると、右マージンがカラム 72 で、カラム 73 から 80 までのすべてのカラムに数字が含まれている場合を除き、右マージンの後に非ブランク文字がある行にコンパイラーがフラグを立てます。

LAXPUNC | NOLAXPUNC

NOLAXPUNC を指定すると、コンパイラーは想定される句読点が欠落している場所に E レベル・メッセージのフラグを立てます。

ステートメント "I = (1 * (2));" を例にとると、コンパイラーはセミコロンの前に右側の閉じ括弧を入れるべきであったと想定します。RULES(NOLAXPUNC) を指定した場合、このステートメントに対しては E レベル・メッセージのフラグが立てられ、指定していない場合は W レベル・メッセージのフラグが立てられます。

LAXQUAL | NOLAXQUAL

NOLAXQUAL を指定すると、コンパイラーは、レベル 1 ではなく、またドット修飾のない構造体メンバーへの参照にフラグを付けます。次の例を考えてみてください。

```
dc1
  1 a,
    2 b fixed bin,
    2 c fixed bin;

c   = 15;    /* would be flagged */
a.c = 15;    /* would not be flagged */
```

LAXSEMI | NOLAXSEMI

NOLAXSEMI を指定すると、コンパイラーは、コメント内に現れるすべてのセミコロンにフラグを立てます。

LAXSTG | NOLAXSTG

NOLAXSTG を使用すると、コンパイラーは、B がパラメーターであったとしても (これが中核です)、変数 A が ADDR(B) および STG(A) > STG(B) の BASED として宣言されている場所にフラグを立てます。

コンパイラーは、B が AUTOMATIC または STATIC ストレージ内にあった場合でも、この種の問題にフラグを立てたと考えられます。しかし、デフォルトでは、B がパラメーターである場合にフラグを立てます (一部に、実引数を記述しないプレースホルダー (置き換え) 属性を使用して B を宣言する場合があるため)。パラメーターと引数の宣言が合致している (または合致している必要がある) 場合は、RULES(NOLAXSTG) を指定すると、ストレージ・オーバーレイの問題をより多く検出できる場合があります。

LAXSTRZ | NOLAXSTRZ

LAXSTRZ を指定すると、コンパイラーは、余分のビットがすべてゼロである (または、余分の文字がすべてブランクである) 場合に、長すぎる定数値に初期化された、または割り当てられたビット変数または文字変数にフラグを立てません。

MULTICLOSE | NOMULTICLOSE

NOMULTICLOSE を指定すると、コンパイラーはステートメントの複数のグループのクローズを強制するすべてのステートメントにフラグを付け、E レベル・メッセージを出します。

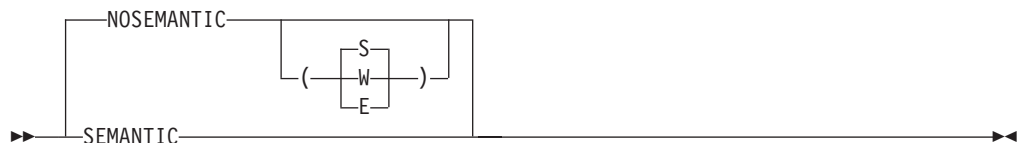
UNREF | NOUNREF

NOUNREF を指定すると、参照されないレベル 1 AUTOMATIC 変数、および参照されるサブエレメントを含まない構造体または共用体であるレベル 1 AUTOMATIC 変数に、コンパイラーがフラグを立てます。

デフォルト: RULES (IBM BYNAME NODECSIZE EVENDEC ELSEIF GOTO NOLAXBIF NOLAXCTL LAXDCL NOLAXDEF LAXIF LAXINOUT LAXLINK LAXPUNC LAXMARGINS(STRICT) LAXQUAL LAXSEMI LAXSTG NOLAXSTRZ MULTICLOSE UNREF)

SEMANTIC

SEMANTIC オプションを指定すると、コンパイラーのセマンティック検査段階の実行は、処理のこの段階の前に出されたメッセージの重大度に依存します。



省略形: SEM、NSEM

SEMANTIC

NOSEMANTIC(S) と同等。

NOSEMANTIC

処理は構文検査の後で停止されます。セマンティック検査は実行されません。

NOSEMANTIC (S)

重大エラーまたは回復不能エラーが検出された場合は、セマンティック検査は行われません。

NOSEMANTIC (E)

エラー、重大エラー、または回復不能エラーが検出された場合は、セマンティック検査は行われません。


NOSEMANTIC (W)

警告、エラー、重大エラー、または回復不能エラーが検出された場合は、セマンティック検査は行われません。

ある種の重大エラーが見つかった場合は、セマンティック検査は実行されません。すべての参照が正しく解決されることをコンパイラーが確認できない場合 (例えば、組み込み関数またはエントリーの参照に引数が少なすぎる場合)、組み込み関数またはエントリーの参照の中の引数の妥当性は検査されません。

SERVICE

SERVICE オプションは、オブジェクト・モジュール内にストリングを配置します。このストリングは、オブジェクトのリンク先であるロード・モジュールとともにメモリーにロードされ、また LE ダンプにトレースバックが組み込まれている場合、このストリングはそのトレースバックにも組み込まれます。

▶▶  SERVICE—(—'service string' —)————▶▶

省略形: SERV、NOSERV

ストリングの長さは 64 文字に制限されます。

ロケールが異なってもストリングを読み取ることができるように、インバリアント文字セットからの文字だけを使用する必要があります。

SOURCE

SOURCE オプションは、コンパイラーが、コンパイラー・リスト内にソース・プログラムのリストを組み込むことを指定します。リストされるソース・プログラムは、オリジナルのソース入力か、あるいは (任意のプリプロセッサが使用される場合は) プリプロセッサの出力です。

▶▶  SOURCE————▶▶

省略形: S、NS

SPILL

SPILL オプションは、コンパイルに使用する予備域のサイズを指定します。一度に使用されるレジスターの数が多すぎる場合、コンパイラーはレジスターの一部を予備域と呼ばれる一時記憶域にダンプします。

▶▶  SPILL—(size)————▶▶

省略形: SP

STATIC

Diagram illustrating a static member function call. The call is marked as **STATIC** and **FULL**. A bracket labeled **SHORT** indicates the scope of the call.

INTERNAL STATIC は、使用される場合のみオブジェクト・モジュールに保管されます。

INITIAL の All INTERNAL STATIC は、オブジェクト・モジュールに保管されます。

STDSYS

Timing diagram showing the relationship between STDSYS and NOSTDSYS signals. STDSYS is a long pulse starting at the beginning of the sequence. NOSTDSYS is a short pulse that occurs early in the sequence, before STDSYS starts.

STMT

Timing diagram showing the relationship between NOSTMT and STMT signals. NOSTMT is a short pulse that occurs at the start of the STMT signal. STMT is a long pulse that follows NOSTMT.

74 Enterprise PL/I for z/OS プログラミング・ガイド

STMT オプションを指定すると、ソース・リストには論理ステートメント番号とソース・ファイル番号の両方が入ります。

GOSTMT オプションがないことに注意してください。実行時に情報を生成し、エラーの発生個所を識別する唯一のオプションは、GONUMBER オプションです。GONUMBER オプションが使用されているとき、ランタイム・エラー・メッセージの中の「ステートメント (statement)」という用語は、NUMBER コンパイラー・オプションによって使用される行番号を参照しますが、これは、STMT オプションの実行下であっても同じであることにも注意してください。

STORAGE

STORAGE オプションは、それぞれのプロシーチャーおよび開始ブロックによって使用されるストレージの要約を、リストの一部としてコンパイラーに作成させます。

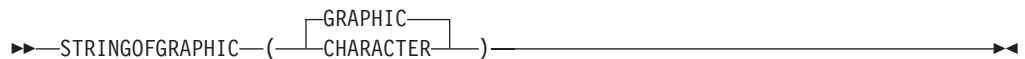


省略形: STG、NSTG

STORAGE 出力には、コンパイルの内部統計に使用されたストレージの量も含まれます。

STRINGOFGGRAPHIC

STRINGOFGGRAPHIC オプションは、GRAPHIC 集合に適用されたときの STRING 組み込み関数の結果が、CHARACTER または GRAPHIC のいずれの属性を持つかを決定します。



省略形: CHAR、G

CHARACTER

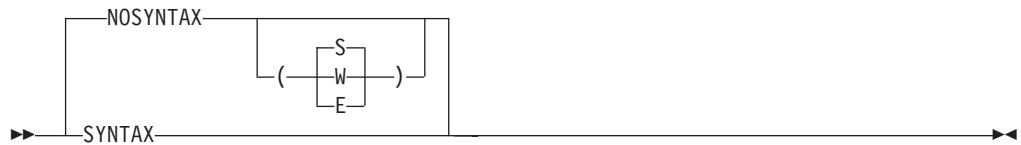
STRINGOFGGRAPHIC(CHAR) を指定すると、STRING 組み込み関数が UNALIGNED NONVARYING GRAPHIC 変数の配列または構造体に適用される場合、結果は CHARACTER 属性を持ちます。

GRAPHIC

STRINGOFGGRAPHIC(GRAPHIC) を指定すると、STRING 組み込み関数が GRAPHIC 変数の配列または構造体に適用される場合、結果は GRAPHIC 属性を持ちます。

SYNTAX

SYNTAX オプションは、回復不能エラーが生じない限り、MACRO オプションを指定した場合、プリプロセスの後でコンパイラーが、引き続き構文検査に移ることを指定します。コンパイラーが引き続きコンパイルを行うかどうかは、NOSYNTAX オプションで指定されたエラーの重大度によって決まります。



省略形: SYN、NSYN

SYNTAX

重大エラーまたは回復不能エラーが起こらない限り、プリプロセス後に構文検査が継続されます。SYNTAX は NOSYNTAX(S) と同等です。

NOSYNTAX

プリプロセス後、処理は無条件に停止されます。

NOSYNTAX(W)

警告、エラー、重大エラー、または回復不能エラーが検出された場合は、構文検査は行われません。

NOSYNTAX(E)

コンパイラーはエラー、重大エラー、または回復不能エラーを検出すると、構文検査を行いません。

NOSYNTAX(S)

コンパイラーは重大エラーまたは回復不能エラーを検出すると、構文検査を行いません。

NOSYNTAX オプションでコンパイルが終了すると、相互参照リスト、属性リスト、およびソース・プログラムの後に続くその他のリストは作成されません。

このオプションを使用すると、プリプロセッサを使用する PL/I プログラムをデバッグするときに無駄な操作を省略することができます。

NOSYNTAX オプションが有効な場合は、CICS、XOPT または XOPTS オプションを介した CICS プリプロセッサの指定がすべて無視されます。これを使用すると、CICS 変換プログラムを起動する前に MACRO プリプロセッサが起動されます。

SYSPARM

SYSPARM オプションを指定すると、マクロ機能組み込み関数 SYSPARM が戻す文字列の値を指定できます。

▶▶SYSPARM—(—'string' —)————▶▶

文字列

長さは最大 64 文字です。デフォルトはヌル・文字列です。

マクロ機能の詳細については、「PL/I 言語解説書」を参照してください。

SYSTEM

SYSTEM オプションは、MAIN PL/I プロシーチャーにパラメーターを渡すのに使用するフォーマットを指定し、また一般的に、プログラムが実行されているホスト・システムを示します。



表 4 は、使用できるパラメーター・リストのタイプと、指定したホスト・システムの下でどのようにプログラムが実行されるかを示しています。また、NOEXECOPS の暗黙の設定値も示しています。MAIN プロシーチャーは、このテーブル内で有効とされているパラメーター・リストのタイプのみを受信しなければなりません。追加の SYSTEM オプションのランタイム情報は、「z/OS 言語環境プログラム プログラミング・ガイド」にあります。.

表 4. SYSTEM オプション・テーブル

SYSTEM オプション	パラメーターの タイプ・リスト	プログラムの 稼働タイプ	NOEXECOPS の暗黙指定の有無
SYSTEM(MVS)	1 つの CHARACTER ストリング、または パラメーターなし。	z/OS アプリケーシ ョン・プログラム	NO
	または、任意のパラ メーター・リスト。		YES
SYSTEM(CICS)	ポインター	CICS® トランザクシ ョン	YES
SYSTEM(IMS)	ポインター	IMS™ アプリケーシ ョン・プログラム	YES
SYSTEM(OS)	z/OS UNIX パラメー ター・リスト	z/OS UNIX アプリケ ーション・プログラ ム	YES
SYSTEM(TSO)	CCPL を指すポイン ター	TSO コマンド・プロ セッサ	YES

SYSTEM(IMS) では、すべてのポインターが BYVALUE で渡されると想定されますが、SYSTEM(MVS) では、BYADDR で渡されると想定されます。

CICS 環境で実行される MAIN プロシーチャーは、SYSTEM(CICS) または SYSTEM(MVS) を指定してコンパイルする必要があります。

SYSTEM(MVS) を指定してコンパイルされ、ランタイム・オプションが渡されずに実行される、DB2 ストアド・プロシーチャーなどのコードの MAIN プロシーチャーの OPTIONS オプションには NOEXECOPS を指定することを強く推奨します。

TERMINAL

TERMINAL オプションは、コンパイル時に作成された診断メッセージおよび通知メッセージを端末に表示するかどうかを決めます。

注: このオプションは、z/OS UNIX 環境でのコンパイルにだけ適用されます。



省略形: TERM、NTERM

TERMINAL

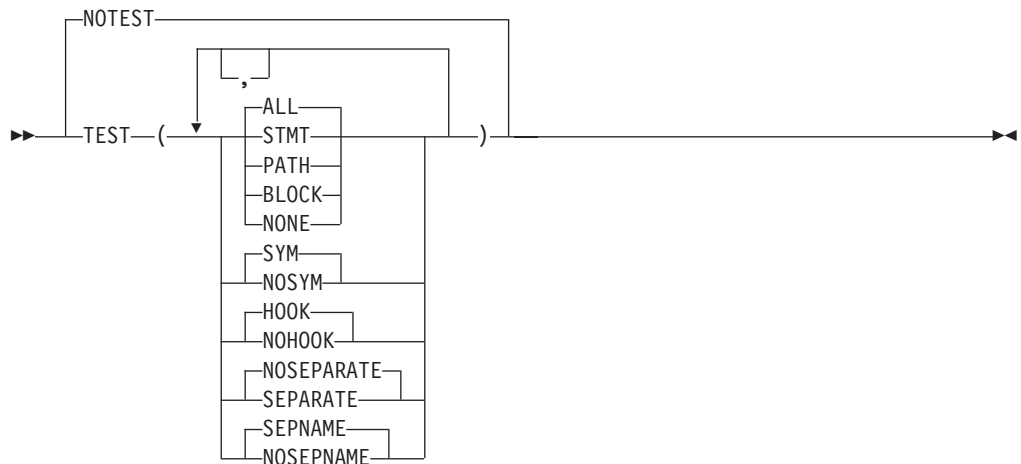
メッセージは端末に表示されます。

NOTERMINAL

通知コンパイラー・メッセージまたは診断コンパイラー・メッセージは端末に表示されません。

TEST

TEST オプションは、コンパイラーがオブジェクト・コードの一部として生成する検査機能のレベルを指定します。この機能を使うと、テスト・フックの位置を制御したり、シンボル・テーブルを生成するかどうかを制御することができます。



省略形: AALL、ACICS、AMACRO、ASQL

STMT

コンパイラーはステートメント・テーブルを生成します。HOOK サブオプションが有効になっている場合は、ステートメント境界とブロック境界にフックを挿入します。

PATH

コンパイラーはステートメント・テーブルを生成します。HOOK サブオプションが有効になっている場合は、以下の場所にフックを挿入します。

- 反復 DO ステートメントに囲まれた最初のステートメントの前
- IF ステートメントの真の部分に含まれる最初のステートメントの前

- IF ステートメントの偽の部分に含まれる最初のステートメントの前
- SELECT グループの真の WHEN または OTHERWISE ステートメントに含まれる最初のステートメントの前
- ユーザー・ラベルに続くステートメントの前 (ラベルの付いた FORMAT ステートメントを除く) ステートメントに複数のラベルがある場合は、フックが 1 つのみ挿入されます。
- CALL または関数参照 (ルーチンに制御が渡される前と後の両方)
- ブロック境界

BLOCK

コンパイラーはステートメント・テーブルを生成します。HOOK サブオプションが有効になっている場合は、ブロック境界 (ブロック入り口とブロック出口) にフックを挿入します。

ALL

コンパイラーはステートメント・テーブルを生成します。HOOK サブオプションが有効になっている場合は、可能なすべての場所にフックを挿入してステートメント・テーブルを生成します。

注: opt(2) の下では、フックはブロック境界だけに設定されます。

NONE

プログラムにフックを入れません。

SYM

変数を名前で検査するためのシンボル・テーブルを作成します。

NOSYM

シンボル・テーブルは生成されません。

NOTEST

すべてのテスト情報の生成を抑止します。

HOOK

TEST サブオプション ALL、STMT、BLOCK、または PATH のいずれかが有効である場合に、コンパイラーによって、生成されたコードにフックが挿入されます。

NOHOOK

コンパイラーによって、生成されたコードにフックは挿入されません。

デバッグ・ツールがオーバーレイ・フックを生成する場合、サブオプション ALL、PATH、STMT、または BLOCK の 1 つを指定しなければなりません。HOOK は指定する必要はなく、事実上 NOHOOK を指定することをお勧めします。

NOHOOK を指定すると、ENTRY および EXIT のブレークポイントは、デバッグ・ツールが停止する PATH ブレークポイントにすぎなくなります。

SEPARATE

コンパイラーは、生成するデバッグ情報のほとんどを別個のデバッグ・ファイルに書き込みます。このオプションを使用する場合、TEST オプションを有効にすると、コンパイラーが作成するオブジェクト・デックのサイズがかなり小さくなります。

プログラムに GET または PUT DATA ステートメントが含まれる場合、別個のデバッグ・ファイルに含まれるデバッグ情報の量が少なくなります。これらのステートメントについては、シンボル・テーブル情報をオブジェクト・デックに書き込む必要があるためです。

生成されるデバッグ情報には、コンパイラーに渡されたソースの圧縮版が常に含まれています。これは、ソースは SYSIN DD * で指定したものであっても、これまでのジョブ・ステップで作成された一時データ・セットであってもかまわないことを意味します (例えば、以前の SQL プリコンパイラーや CICS プリコンパイラーの出力がソースであってもかまいません)。

このサブオプションをバッチ・コンパイルで使用する場合は、そのコンパイルの JCL に SYSDEBUG を表す DD カードが含まれていて、その DD カードは RECFM=FB および 80 <= LRECL <= 1024 のデータ・セットを指定していなければなりません。

このサブオプションを LINEDIR コンパイラー・オプションと一緒に使用することはできません。

NOSEPARATE

コンパイラーは、生成するデバッグ情報のすべてをオブジェクト・デックに書き込みます。

このオプションを指定すると、生成されるデバッグ情報には、コンパイラーに渡されたソースの圧縮版は含まれません。これは、プログラムをデバッグしようとするときに DebugTool で見つけることができるデータ・セットにソースがなければならぬことを意味します。

SEPNAME

コンパイラーは、別個のデバッグ・ファイルの名前をオブジェクト・デックに書き込みます。

SEPARATE オプションが有効になっていない場合、このオプションは無視されます。

NOSEPNAME

コンパイラーは、別個のデバッグ・ファイルの名前をオブジェクト・デックに書き込みません。

SEPARATE オプションが有効になっていない場合、このオプションは無視されます。

注:

- opt(2) または opt(3) を指定すると、フックはブロック境界にのみ設定されます。
- SEPARATE コンパイラー・オプションを使用してコンパイルしたコードをデバッグするには、Debug Tool バージョン 6 以降を使用する必要があります。
- 範囲が連結データ・セットに及ぶ入力ファイルは、サポートされていません。

TEST(NONE,NOSYM) を指定すると、コンパイラーはこのオプションを NOTEST に設定します。

TEST(NONE,SYM) は使用しないでください。この設定を指定する意図が不明です。TEST(ALL,SYM,NOHOOK) または TEST(STMT,SYM,NOHOOK) を指定するとよいでしょう。

NOTEST と TEST(NONE,NOSYM) 以外の TEST オプションは、いずれもプログラム・テストのためのアテンション割り込み機能を提供します。

プログラムに、呼び出したい ATTENTION ON ユニットがある場合は、次のオプションのどちらかを用いてプログラムをコンパイルしなければなりません。

- INTERRUPT オプション
- NOTEST または TEST(NONE,NOSYM) 以外の TEST オプション

注: ATTENTION は TSO の下でだけサポートされます。

TEST オプションでは GONUMBER が暗黙指定されます。

TEST オプションでは、オブジェクト・コードのサイズが増大してパフォーマンスに影響が出ることがあるため、フックの数と位置に限度を設けなければならないことがあります。

TEST オプションを指定した場合、インライン化は行われません。

REFER の構造体はシンボル・テーブルでサポートされています。

TEST(SYM) が有効な場合は、DebugTool の自動モニター機能を有効にするために、コンパイラーはテーブルを生成します。TEST(SEPARATE) オプションが有効になっていなければ、これらのテーブルのために、オブジェクト・モジュールのサイズがかなり大きくなる場合があります。DebugTool の自動モニター機能を活動化すると、これらのテーブルを使用して、ステートメントで使用される変数の値がステートメントの実行前に表示されます。これは、変数が計算可能なタイプであるか、POINTER、OFFSET、または HANDLE の属性を持っていることが条件となります。ステートメントが割り当てステートメントである場合は、ターゲットの値も表示されます。ただし、ターゲットの初期設定や割り当てがあらかじめ行われていない場合は、その値に意味がありません。

名前に * を使用して宣言されている変数はすべて、DebugTool の使用時には表示されません。また、* が親構造体または副構造の名前として使用されている場合は、そのすべての子も表示されません。このような目的のために、名前を付けないでおく任意の構造体エレメントの名前に、単一下線を使用することをお勧めします。

AFTERMACRO、AFTERSQL、および AFTERALL サブオプションの異なる影響の例として、PP オプションが PP(MACRO('INCONLY'), SQL, MACRO) であったと仮定してください。そうすると、

- TEST(SEP,AFTERMACRO) が指定されている場合、リストおよび「デバッグ・ツール・ソース (Debug Tool source)」ウィンドウのソースが、MACRO プリプロセッサの 2 番目の呼び出しが生成した MDECK からきたかのように表示されます。
- TEST(SEP,AFTERSQL) が指定されている場合、リストおよび「デバッグ・ツール・ソース (Debug Tool source)」ウィンドウのソースは、SQL プリプロセッサ

一の呼び出しが生成した MDECK からきたかのように表示されます (したがって、%DCL およびその他のマクロ・ステートメントはまだ表示されます)。

- TEST(SEP,AFTERALL) が指定されている場合は、MACRO プリプロセッサが PP オプションの最後であるため、「source」は TEST(SEP,AFTERMACRO) オプションの下にきます。

TUNE

TUNE オプションでは、実行可能プログラムが最適化されるアーキテクチャーを指定します。このオプションにより、最適化プログラムは命令のスケジューリングなどの、アーキテクチャーの相違の利点を利用できます。

▶▶ TUNE—(——)——▶▶

注: TUNE レベルが ARCH より低い場合、TUNE は強制的に ARCH になります。




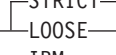

TUNE レベルに指定できる現行値は、次のとおりです。

- 5 すべてのモデルで実行可能であるが、ARCH(5) で指定されたモデルに最適化されたコードを生成します。
- 6 すべてのモデルで実行可能であるが、ARCH(6) で指定されたモデルに最適化されたコードを生成します。
- 7 すべてのモデルで実行可能であるが、ARCH(7) で指定されたモデルに最適化されたコードを生成します。
- 8 すべてのモデルで実行可能であるが、ARCH(8) で指定されたモデルに最適化されたコードを生成します。

5 より小さい TUNE 値を指定すると、コンパイラーがその値を 5 に再設定します。

USAGE

USAGE オプションを使用すると、さまざまなセマンティクスを、選択された組み込み関数用に選択できます。

▶▶ USAGE—(——
HEX—(——)
ROUND—(——)
SUBSTR—(——)
UNSPEC—(——)——▶▶

HEX(SIZE | CURRENTSIZE)

HEX(SIZE) サブオプションで、HEX が VARYING または VARYINGZ ストリングに適用された場合、ストリングで使用されているストレージの最大容量を示す 16 進ストリングが返されます。

HEX(CURRENTSIZE) サブオプションで、HEX が VARYING または VARYINGZ スtringに適用された場合、Stringで使用されているストレージの現在容量を示す 16 進Stringが返されます。

ROUND(IBM | ANS)

ROUND(IBM) サブオプションを指定すると、ROUND 組み込み関数の最初の引数が FLOAT 属性を持っている場合、2 番目の引数は無視されます。

ROUND(ANS) サブオプションを指定すると、ROUND 組み込み関数は、「PL/I 言語解説書」で説明されているようにインプリメントされます。

SUBSTR(STRICT | LOOSE)

SUBSTR(STRICT) サブオプションでは、x が CHARACTER タイプである場合、SUBSTR(x,y,z) 組み込み関数参照は、長さが MIN(z, MAXLENGTH(x)) であるStringを返します。

SUBSTR(LOOSE) サブオプションでは、同じ参照で、長さが z であるStringが返されます。

SUBSTR(LOOSE) サブオプションは、x が CHAR(1) BASED 変数である SUBSTR(x,y,z) 参照があるものの場合に役立つことがあります。

UNSPEC(IBM | ANS)

UNSPEC(IBM) サブオプションを指定すると、UNSPEC を構造体に適用することはできません。配列に適用すると、ビット・Stringの配列が戻されます。

UNSPEC(ANS) サブオプションを指定すると、UNSPEC を構造体に適用できます。構造体または配列に適用すると、UNSPEC はシングル・ビット・Stringを戻します。

デフォルト: USAGE(HEX(SIZE) ROUND(IBM) SUBSTR(STRICT) UNSPEC(IBM))

WIDECHAR

WIDECHAR オプションは、WIDECHAR データが保管されるフォーマットを指定します。



BIGENDIAN

WIDECHAR データをビッグ・エンディアン・フォーマットで保管するように指示します。例えば、UTF-16 文字 '1' の WIDECHAR 値は '0031'x として保管されます。

LITTLEENDIAN

WIDECHAR データをリトル・エンディアン・フォーマットで保管するように指示します。例えば、UTF-16 文字 '1' の WIDECHAR 値は '3100'x として保管されます。

WX 定数は、常にビッグ・エンディアン・フォーマットで指定する必要があります。したがって、WIDECHAR(LITTLEENDIAN) オプションを指定した場合、値 '1' は '3100'x として保管されますが、この値は常に '0031'wx として指定する必要があります。

WINDOW

WINDOW オプションは、各種の日付関連の組み込み関数で 사용되는 **w** ウィンドウ引数を設定します。



- w** 固定ウィンドウの開始を表す符号なし整数、または「スライディング」ウィンドウを指定する負の整数。例えば、WINDOW(-20) は、プログラムを実行する 20 年前に開始されるウィンドウを示します。

WRITABLE

WRITABLE オプションは、静的ストレージを書き込み可能として扱うことをコンパイラーに指定します (また、そうする場合は、その結果のコードが再入不可になります)。



このオプションは、RENT オプションを指定してコンパイルされたプログラムには効果がありません。

NORENT WRITABLE オプションを使用すると、コンパイラーは次の目的で静的ポインターを使用できます。

- CONTROLLED 変数を追跡するスタック用の基数として使用する
- FILE を表すストレージ用のハンドルとして使用する

したがって、NORENT WRITABLE オプションを指定すると、CONTROLLED 変数を使用するモジュール、または入出力を実行するモジュールは再入可能にはなりません。

NORENT NOWRITABLE オプションでは、コンパイラーが静的ポインターを以下のようにすることが必要です。

- CONTROLLED 変数を追跡するスタック用の基数として使用する
- FILE を表すストレージ用のハンドルとして使用する

したがって、NORENT NOWRITABLE オプションを指定すると、CONTROLLED 変数を使用するモジュール、または入出力を実行するモジュールは再入可能になります。

FWS および PRV サブオプションは、以下のように、コンパイラーが CONTROLLED 変数を扱う方法を決定します。

FWS

EXTERNAL プロシージャに入るときに、コンパイラーはライブラリー呼び出

しを行って、プロシージャー (およびすべてのサブプロシージャー) 内で CONTROLLED 変数をアドレス指定するために使用できるストレージを見つけます。

PRV

コンパイラーは、CONTROLLED 変数をアドレス指定するために、旧 OS PL/I コンパイラーで使用していたものと同じ、疑似レジスター変数メカニズムを使用します。

このため、NORENT NOWRITABLE(PRV) オプションを指定すると、旧コードと新規コードで CONTROLLED 変数を共用することができます。

ただし、NORENT NOWRITABLE(PRV) オプションを指定すると、CONTROLLED 変数の使用が、旧コンパイラーにおけるすべての制約を受けることになることも意味します。

NORENT NOWRITABLE(FWS) オプションを指定すると、アプリケーションが以下のとおりである場合に、RENT または WRITABLE オプションを指定してコンパイルされたようには実行されないことがあります。

- アプリケーションで CONTROLLED 変数を使用している
- アプリケーションで FILE CONSTANT を FILE VARIABLE に割り当てている

NORENT NOWRITABLE(FWS) を指定した場合のアプリケーションのパフォーマンスは、多くの CONTROLLED 変数を多くの PROCEDURE で使用していると特に悪くなる場合があります。

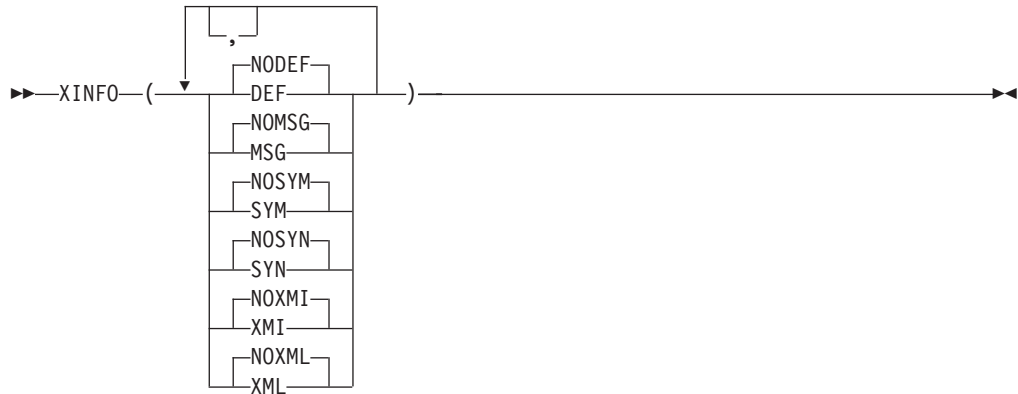
NOWRITABLE オプションを指定すると、PROCEDURE の外部の PACKAGE で、以下を宣言できない場合があります。

- CONTROLLED 変数
- FETCHABLE ENTRY 定数
- FILE 定数

NORENT WRITABLE を指定してコンパイルされたコードは、任意の外部 CONTROLLED 変数を共用する、NORENT NOWRITABLE を指定してコンパイルされたコードと混在できません。一般に、WRITABLE でコンパイルされたコードと NOWRITABLE でコンパイルされたコードを混在しないでください。

XINFO

XINFO オプションは、現行コンパイル単位に関する追加の情報が入ったファイルを追加して生成するようにコンパイラーに指定します。



DEF

定義 SIDEDECK ファイルが作成されます。このファイルは、コンパイル単位について、次のすべてをリストします。

- 定義済み EXTERNAL プロシーチャー
- 定義済み EXTERNAL 変数
- 静的に参照される EXTERNAL ルーチンおよび変数
- 動的に呼び出されたフェッチされたモジュール

バッチ環境では、このファイルは SYSDEFSD DD ステートメントによって指定されたファイルに書き込まれます。z/OS UNIX Systems Services の環境では、このファイルはオブジェクト・デックと同じディレクトリーに書き込まれ、拡張子 "def" が付けられます。

例えば、次のプログラムがあるとします。

```
defs: proc;
  dcl (b,c) ext entry;
  dcl x ext fixed bin(31) init(1729);
  dcl y ext fixed bin(31) reserved;
  call b(y);
  fetch c;
  call c;
end;
```

この場合、次の def ファイルが生成されます。

```
EXPORTS CODE
  DEFS
EXPORTS DATA
  X
IMPORTS
  B
  Y
  FETCH
  C
```

def ファイルを使用して、アプリケーションの依存性グラフを作成したり、相互参照分析を行ったりすることができます。

NODEF

定義 SIDEDECK ファイルは作成されません。

MSG

メッセージ情報が ADATA ファイルに生成されます。ADATA ファイルのフォーマットについての詳細は、付録を参照してください。

バッチでは、ADATA ファイルは、SYSADATA DD ステートメントによって指定されたファイルに生成されます。z/OS UNIX では、ADATA はオブジェクト・ファイルと同じディレクトリーに生成され、adt という拡張子を持ちます。

NOMSG

メッセージ情報は ADATA ファイルに生成されません。MSG も SYM も指定しないと、ADATA ファイルは生成されません。

SYM

シンボル情報が ADATA ファイルに生成されます。ADATA ファイルのフォーマットについての詳細は、付録を参照してください。

バッチでは、ADATA ファイルは、SYSADATA DD ステートメントによって指定されたファイルに生成されます。z/OS UNIX では、ADATA ファイルはオブジェクト・ファイルと同じディレクトリーに生成され、adt という拡張子を持ちます。

NOSYM

シンボル情報は ADATA ファイルに生成されません。

SYN

構文情報が ADATA ファイルに生成されます。ADATA ファイルのフォーマットについての詳細は、付録を参照してください。XINFO(SYN) オプションを指定すると、コンパイラーが必要とするストレージ量 (メモリーと生成されるファイルの両方) が大幅に増えることがあります。

バッチでは、ADATA ファイルは、SYSADATA DD ステートメントによって指定されたファイルに生成されます。z/OS UNIX では、ADATA ファイルはオブジェクト・ファイルと同じディレクトリーに生成され、adt という拡張子を持ちます。

NOSYN

構文情報は ADATA ファイルに生成されません。

XMI

XMI サイド・ファイルが作成されます。他のツールによる場合を除いて、この XMI は読み取られたり、解釈されたりするよう設計されていません。

バッチ環境では、このファイルは SYSXMI DD ステートメントによって指定されたファイルに書き込まれます。z/OS UNIX Systems Services の環境では、このファイルはオブジェクト・デックと同じディレクトリーに書き込まれ、拡張子「xmi」が付けられます。

NOXMI

XMI サイド・ファイルは作成されません。

XML

XML サイド・ファイルが作成されます。この XML ファイルには、以下が含まれます。

- コンパイル用のファイル参照テーブル
- コンパイルされたプログラムのブロック構造
- コンパイル時に作成されたメッセージ

バッチ環境では、このファイルは SYSEXMLSD DD ステートメントによって指定されるファイルに書き込まれます。z/OS UNIX Systems Services の環境では、このファイルはオブジェクト・デックと同じディレクトリーに書き込まれ、拡張子 "xml" が付けられます。

作成された XML の DTD ファイルは次のとおりです。

```
<?xml encoding="UTF-8"?>

<!ELEMENT PACKAGE ((PROCEDURE)*,(MESSAGE)*,FILEREFERNCETABLE)>
<!ELEMENT PROCEDURE (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT BEGINBLOCK (BLOCKFILE,BLOCKLINE,(PROCEDURE)*,(BEGINBLOCK)*)>
<!ELEMENT MESSAGE (MSGNUMBER,MSGLINE?,MSGFILE?,MSGTEXT)>
<!ELEMENT FILE (FILENUMBER,INCLUDEDFROMFILE?,INCLUDEDONLINE?,FILENAME)>
<!ELEMENT FILEREFERNCETABLE (FILECOUNT,FILE+)>

<!ELEMENT BLOCKFILE (#PCDATA)>
<!ELEMENT BLOCKLINE (#PCDATA)>
<!ELEMENT MSGNUMBER (#PCDATA)>
<!ELEMENT MSGLINE (#PCDATA)>
<!ELEMENT MSGFILE (#PCDATA)>
<!ELEMENT MSGTEXT (#PCDATA)>
<!ELEMENT FILECOUNT (#PCDATA)>
<!ELEMENT FILENUMBER (#PCDATA)>
<!ELEMENT FILENAME (#PCDATA)>
<!ELEMENT INCLUDEDFROMFILE (#PCDATA)>
<!ELEMENT INCLUDEDONLINE (#PCDATA)>
```

NOXML

XML サイド・ファイルは作成されません。

XML

XML オプションを使用して、XMLCHAR 組み込み関数によって生成される XML の名前の大/小文字を選択します。

►►XML(—(CASE—(—UPPER
ASIS—))—)►►

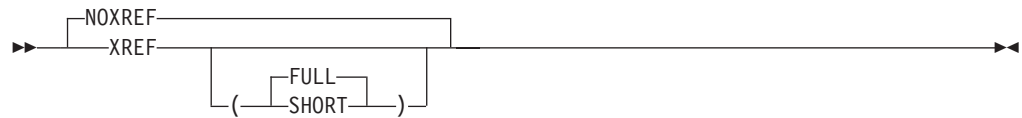
CASE(UPPER | ASIS)

CASE(UPPER) サブオプションを指定した場合、XMLCHAR 組み込み関数によって生成された XML 内の名前がすべて大文字になります。

CASE(ASIS) サブオプションを指定した場合、XMLCHAR 組み込み関数によって生成される XML 内の名前の大/小文字は、宣言で使用したとおりになります。なお、マクロ・プリプロセッサ・オプション CASE(ASIS) を使用しないで MACRO プリプロセッサを使用した場合は、コンパイラが表示するソースの名前はすべて大文字になり、XML(CASE(ASIS)) オプションを指定しても機能しません。

XREF

XREF オプションは、プログラム内で使用する名前の相互参照テーブル、ならびにその名前が宣言または参照されるステートメントの数をコンパイラ・リストに入れることを指定します。



省略形: X、NX

FULL

すべての ID と属性をコンパイラー・リストに入れます。

SHORT

参照されない ID をコンパイラー・リストから省きます。

XREF オプションを使用して作成された相互参照リストに入れられない名前は、END ステートメントのラベル参照だけです。例えば、プロシージャ PROC1 のステートメント番号 20 が END PROC1; であると想定します。この場合、ステートメント番号 20 は PROC1 用の相互参照リストには入りません。

XREF オプションと ATTRIBUTES オプションを両方とも指定すると、2 つのリストが組み合わされます。SHORT と FULL が対立する場合は、最後に指定したオプションで使用方法が決まります。例えば ATTRIBUTES (SHORT) XREF (FULL) を使用するとその結果は、リストを組み合わせた FULL オプションになります。

相互参照テーブルのフォーマットと内容については、97 ページの『相互参照テーブル』を参照してください。

オプションの中のブランク、コメント、およびストリング

オプションを指定するときにブランクを 1 つ使用できるところでは、どこでも希望するだけの数のブランクまたはコメントを指定することもできます。

ただし、%PROCESS の行またはオプション・ファイルの行にコメントを指定する場合、そのコメントは、開始したのと同じ行で終了しなければなりません。

同様に、コマンド行または PARM= 指定の中でコメントを開始する場合、そのコメントもその行で終わらなければなりません。

同じ規則がストリングにも適用されます。つまり、%PROCESS の行またはオプション・ファイルの行にストリングを指定する場合、そのストリングは、開始したのと同じ行で終了しなければなりません。同様に、コマンド行または PARM= 指定の中でストリングを開始する場合、そのストリングもその行で終わらなければなりません。

デフォルト・オプションの変更

デフォルト設定のコンパイラー・オプションを変更する場合は、コンパイラーのインストール時に、サンプル・ジョブ IBMZWIOF を編集して実行依頼する必要があります。

このジョブで、あらかじめ適用しておくオプションを指定し、これら以外のオプションは後で追加するという形にすることで、デフォルト・オプションを実質的に変更することができます。またこのジョブでは、後から追加して最終的に適用するオ

プシオンを指定することで、デフォルト・オプションを実質的に変更することができます。この方法を使用すると、このジョブで指定したオプションが指定変更されることがありません。

マクロ・プリプロセッサのデフォルト・オプションを変更する場合は、インストール時にこのジョブの一部として、該当する PPMACRO オプションを指定することによって行うこともできます。 PPCICS オプションと PPSQL オプションを使用すれば、それぞれ CICS プリプロセッサと SQL プリプロセッサについて変更することができます。

詳しくは、サンプル・ジョブに記述されている説明を参照してください。

%PROCESS ステートメントまたは *PROCESS ステートメントでのオプションの指定

%PROCESS または *PROCESS をプログラムで使用でき、どちらも等しく受け入れられます。本書では、整合性と読みやすさのために、%PROCESS だけを取り上げます。

%PROCESS ステートメントは各外部プロシージャの開始を識別し、コンパイルごとにコンパイラ・オプションを指定できるようにします。隣接する %PROCESS ステートメント内で指定するオプションは、ソース・ステートメントの入力の終わりまでのコンパイルに対してか、または次の %PROCESS ステートメントまでのコンパイルに対して用いられます。

%PROCESS ステートメントでオプションを指定するには、次のようにコーディングします。

```
%PROCESS options;
```

ここで *options* はコンパイラ・オプションのリストです。オプション・リストの終わりにはセミコロンを入れなければならず、また、オプション・リストはデフォルトの右側ソース・マージンを超えてはなりません。パーセント記号 (%) またはアスタリスク (*) がレコードの最初の列になければなりません。キーワード PROCESS は、次のバイト (桁) 内か、任意の数のブランクの後に置くことができます。オプション・キーワードは、コンマまたは少なくとも 1 つのブランクを使って区切らなければなりません。

文字数を制限するものはレコードの長さだけです。オプションをどれも指定したくない場合は、次のようにコーディングします。

```
%PROCESS;
```

%PROCESS ステートメントを次のレコードまで続けなければならない場合は、リストの前半部分を任意の区切り文字の後で終了してから、次のレコードへ移ります。キーワードやキーワード引数を複数のレコードに分割することはできません。

%PROCESS ステートメントを複数行にまたがって続けることも、または、新たに %PROCESS ステートメントを開始することもできます。隣接する複数の %PROCESS ステートメントの例を次に示します。

```
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST TEST ;
```

コンパイル時オプション、その省略形構文、および IBM 提供のデフォルトは、5 ページの表 3 に示してあります。

%PROCESS ステートメントがあるかどうかをコンパイラーが判別する方法は、初期ソース・ファイルのフォーマット設定によって決まります。

F または FB フォーマット

レコードの先頭文字が "*" または "%" の場合、コンパイラーは次の非ブランク文字が "PROCESS" かどうかを確認します。

V または VB フォーマット

レコードの先頭文字が数値の場合、コンパイラーは最初の 8 文字がシーケンス番号であると想定し、9 番目の文字が "*" または "%" であれば、次の非ブランク文字が "PROCESS" であるかどうかを確認します。しかし、先頭文字が数値でない場合は、先頭文字が "*" または "%" であれば、コンパイラーは次の非ブランク文字が "PROCESS" であるかどうかを確認します。

U フォーマット

レコードの先頭文字が "*" または "%" の場合、コンパイラーは次の非ブランク文字が "PROCESS" かどうかを確認します。

% ステートメントの使用

コンパイラーの操作を指示するステートメントは、パーセント (%) 記号で始まります。 % ステートメントを使用すると、ソース・プログラム・リストを制御し、ソース・プログラムに外部ストリングを組み込むことができます。 % ステートメントにはラベルや条件接頭語が付いてはなりません。また、複合ステートメントの単位にすることもできません。 % ステートメントはおのこの、1 行内に単独で入ってなければなりません。

% 制御ステートメント—%INCLUDE、%PRINT、%NOPRINT、%OPTION、%PAGE、%POP、%PUSH、および %SKIP— それぞれの使用法を以下にリストします。これらのステートメントの完全な説明は、「PL/I 言語解説書」を参照してください。

%INCLUDE

文字またはグラフィックス、あるいはその両方の外部ストリングを、ソース・プログラムに組み込むようコンパイラーに指示します。

%PRINT

SOURCE および INSOURCE のリストの印刷を再開するよう、コンパイラーに指示します。

%NOPRINT

SOURCE および INSOURCE のリストの印刷を、 %PRINT ステートメントが見つかるまで延期するようコンパイラーに指示します。

%OPTION

選択されたサブセットのコンパイラー・オプションの 1 つをソース・コードの 1 つのセグメントに指定します。

%PAGE

プログラム・リスト内の %PAGE ステートメントの直後のステートメントを、次ページの最初の行に印刷するようコンパイラーに指示します。

%POP 最新の **%PUSH** により保存された **%PRINT**、**%NOPRINT**、および **%OPTION** の状況を復元するようコンパイラーに指示します。

%PUSH

%PRINT、**%NOPRINT**、および **%OPTION** の現行状況を後入れ先出し方式でプッシュダウン・スタックに保存します。

%SKIP スキップしたい行数を指定します。

%INCLUDE ステートメントの使用

%INCLUDE ステートメントは、コンパイル単位内の指定された点に追加の **PL/I** ファイルを組み込むために使用します。**%INCLUDE** を使ってライブラリーにあるソース・テキストを **PL/I** プログラムに組み込む方法についての説明は、「**PL/I** 言語解説書」にあります。

バッチ・コンパイルの場合

ライブラリー とは、メンバーと呼ばれるその他のデータ・セットを保管するのに使用できる **z/OS** 区分データ・セットです。**%INCLUDE** ステートメントを使って **PL/I** プログラムに挿入しようとするソース・テキストは、ライブラリー内のメンバーとして存在していなければなりません。163 ページの『ソース・ステートメント・ライブラリー (SYSLIB)』では、ソース・ステートメント・ライブラリーをコンパイラーに対して定義するプロセスについてさらに説明します。

次のステートメント

```
%INCLUDE DD1 (INVERT);
```

は、**DD1** という名前の **DD** ステートメントで定義されたライブラリーのメンバー **INVERT** 内のソース・ステートメントを、ソース・プログラムに連続して挿入することを指定します。コンパイル・ジョブ・ステップには、適切な **DD** ステートメントが入っていなければなりません。

dd 名を省略すると、**SYSLIB** という **dd** 名がとられます。その場合、**SYSLIB** という名前で **DD** ステートメントを組み込む必要があります。(IBM 提供のカatalog式プロシージャの場合は、コンパイル・プロシージャ・ステップにこの名前の **DD** ステートメントは組み込まれていません。)

z/OS UNIX コンパイルの場合

実際の組み込みファイルの名前は、**UPPERINC** を指定しない限り、小文字でなければなりません。例えば、`%include sample` という **include** ステートメントを使用した場合、コンパイラーは、`sample.inc` というファイルを検出しますが `SAMPLE.inc` というファイルは検出しません。`%include SAMPLE` という **include** ステートメントを使用した場合でも、コンパイラーは `sample.inc` を検索します。

コンパイラーは次の順序で **INCLUDE** ファイルを探します。

1. 現行ディレクトリー
2. **-I** フラグまたは **INCDIR** コンパイラー・オプションで指定されたディレクトリー
3. `/usr/include` ディレクトリー
4. **INCPDS** コンパイラー・オプションで指定された **PDS**

コンパイラーが検出した最初のファイルがソースに組み込まれます。

%INCLUDE ステートメントによってソース・テキスト内に %PROCESS ステートメントが組み込まれると、コンパイル・エラーになります。

図 1 は、%INCLUDE ステートメントを使って、プロシージャ TEST 内に FUN 用のソース・ステートメントを組み込む方法を示しています。ライブラリー HPU8.NEWLIB が修飾名 PLI.SYSLIB を持つ DD ステートメントに定義されていますが、これはこのジョブ用のカタログ式プロシージャのステートメントに付け加えられます。ソース・ステートメント・ライブラリーが SYSLIB という名前の DD ステートメントで定義されているため、%INCLUDE ステートメントに DD 名を入れる必要はありません。

ソース・プログラムや、入れようとするテキストにマクロ・ステートメントがまったく入っていないければ、プリプロセッサを呼び出す必要はありません。

```
//OPT4#9      JOB
//STEP3       EXEC IBMZCBG,PARM.PLI='INC,S,A,X,NEST'
//PLI.SYSLIB DD DSN=HPU8.NEWLIB,DISP=OLD
//PLI.SYSIN DD *
TEST: PROC OPTIONS(MAIN) REORDER;
  DCL ZIP PIC '99999';          /* ZIP CODE          */
  DCL EOF BIT INIT('0'B);
  ON ENDFILE(SYSIN) EOF = '1'B;
  GET EDIT(ZIP) (COL(1), P'99999');
  DO WHILE(~EOF);
    PUT SKIP EDIT(ZIP, CITYFUN(ZIP)) (P'99999', A(16));
    GET EDIT(ZIP) (COL(1), P'99999');
  END;
  %PAGE;
  %INCLUDE FUN;
END;                          /* TEST              */
//GO.SYSIN DD *
95141
95030
94101
//
```

図 1. ライブラリーからのソース・ステートメントの組み込み

%OPTION ステートメントの使用

%OPTION ステートメントを使用すると、選択されたコンパイラーのオプションの設定を変更することができます。変更は、次に %POP; ステートメントが読み取られるまで有効です。

現行では、%OPTION ステートメントは LANGLVL オプションを SAA2 から SAA (またはその反対) へ変更するためだけに使用します。従って構文は次のようになります。

►►—%OPTION—LANGLVL—(——)——►►

例えば、LANGLVL(SAA) オプションを指定するコンパイルの中で LANGLVL(SAA2) フィーチャーを使用したコードがあれば、「コンパイラーにこのコードを受け入れさせる」の部分を次の手順で囲むことができます。

1. %PUSH; ステートメントを挿入します。
2. %OPTION LANGLVL(SAA2) ステートメントを挿入します。
3. LANGLVL(SAA2) フィーチャーを使用したコードを挿入します。
4. %POP; ステートメントを挿入します。

コンパイラー・リストの使用

コンパイルのときにコンパイラーは、大半がオプションのリストを生成しますが、そのリストには、ソース・プログラム、コンパイル、およびオブジェクト・モジュールに関する情報が入っています。次のリストの説明は、印刷ページ上の外観について述べています。

注:

コンパイラー・リストは利用可能ですが、これはプログラミング・インターフェースではないため、変更される可能性があります。

当然のことながら、特定の処理段階に達する前にコンパイルが終了してしまえば、それに対応したリストは作成されません。

見出し情報

リストの最初のページは、製品番号、コンパイラー・バージョン番号、コンパイラーがビルドされた時を指定するストリング、およびコンパイルの開始日時で識別されます。このページおよび以降のページには番号が付けられます。

次にリストには、このコンパイルに指定されたすべてのオプションが表示されます。これらのオプションは、NOOPTIONS オプションが指定されていても表示されます。示される内容と順序は、以下のとおりです。

- 初期インストール・オプション (インストール時のオプション・セットで、あらかじめ適用されるオプションです。これら以外のオプションは、後で追加されます)。
- z/OS UNIX 環境では、IBM_OPTIONS 環境変数で指定されたオプション。
- コンパイラーに渡されるパラメーター・ストリング (z/OS UNIX 環境でのコマンド行またはバッチ環境での PARM= に含まれる) で指定されたオプション。
- コンパイラー・パラメーター・ストリングで指定されたオプション・ファイルで指定されたオプション。

これには、各オプション・ファイルの名前とその内容が、コンパイラーが読み取ったそのままの形式で含まれます。

- ソース内の *PROCESS または %PROCESS 行で指定されたオプション。
- 最終インストール・オプション (インストール時のオプション・セットで、他のすべてのオプションの後に適用されるオプションです)。

リストの終わり付近には、コンパイル時にエラーや警告状態がなにも検出されなかった旨のステートメントか、または 1 つ以上のエラーが検出された旨のメッセージが入ります。メッセージのフォーマットについては、102 ページの『メッセージと戻りコード』で説明します。リストの最後から 2 番目の行は、コンパイルに要した時間を示します。リストの最後の行は、*END OF COMPILATION OF xxxx* です。*xxxx* は外部プロシージャ名です。NOSYNTAX コンパイラー・オプションを指定したり、コンパイルの初期段階でコンパイラーが異常終了すると、外部プロシージャ名の *xxxx* が切り捨てられ、この行は *END OF COMPILATION* となります。

以下のセクションでは、リストのオプションの部分を出てくる順に説明します。

コンパイルに使用するオプション

OPTIONS オプションを指定すると、コンパイルに指定されたオプション (デフォルト・オプションを含む) の完全なリストが、次のページから示されます。コンパイル時に最終的に有効になるすべてのオプションの設定値が示されます。オプションの設定値が、初期のインストール・オプションが適用された後に、デフォルトの設定値とは異なる場合は、その行に + マークが付けられます。

プリプロセッサ入力

MACRO オプションと INSOURCE オプションを両方とも指定すると、コンパイラーは、各行の左側に順次番号を付けてから、プリプロセッサへの入力を 1 行に 1 レコードずつリストします。

プリプロセッサがエラーまたはエラーの可能性を検出すると、そのページまたは入力リストに続くページにメッセージが印刷されます。このメッセージのフォーマットは、102 ページの『メッセージと戻りコード』に説明のあるコンパイラー・メッセージのフォーマットと同じです。

SOURCE プログラム

SOURCE オプションを指定すると、コンパイラーは 1 行に 1 レコードずつリストします。これらのレコードには、ソース行番号とソース・ファイル番号が常に含まれます。ただし、ファイルに含まれる行数が 999,999 以上の場合、コンパイラーは大きすぎることを示すフラグをファイルに立て、ソース行番号として下位の 6 桁のみをリストします。

入力レコードにプリンター制御文字、%SKIP ステートメント、または %PAGE ステートメントが入っている場合は、それらの指定にしたがって行のスペーシングが行われます。%NOPRINT ステートメントと %PRINT ステートメントを使って、リストの印刷を停止したり再開したりすることができます。

MACRO オプションを指定すると、ソース・リストでは、1 次入力データ・セットの %INCLUDE ステートメントの代わりに組み込まれたテキストが印刷されます。

ステートメントのネスト・レベル

NEST オプションを指定すると、次の例のように、見出し LEV と NT の下のステートメントまたは行番号の右側に、ブロック・レベルと DO レベルがそれぞれ印刷されます。

Line.	File	LV	NT	
1.0				A: PROC OPTIONS(MAIN);
2.0	1			B: PROC;
3.0	2			DCL K(10,10) FIXED BIN (15);
4.0	2			DCL Y FIXED BIN (15) INIT (6);
5.0	2			DO I=1 TO 10;
6.0	2	1		DO J=1 TO 10;
7.0	2	2		K(I,J) = N;
8.0	2	2		END;
9.0	2	1		BEGIN;
10.0	3	1		K(1,1)=Y;
11.0	3	1		END;
12.0	2	1		END B;
13.0	1			END A;

ATTRIBUTE と相互参照テーブル

ATTRIBUTES オプションを指定すると、コンパイラーは、ソース・プログラム内の ID リストの入った属性テーブルを、それぞれの宣言属性とデフォルト属性を付けて印刷します。

XREF オプションを使用すると、コンパイラーは、ID が入っているステートメントのファイル番号と行番号を併記したうえで、ソース・プログラム内の ID のリストの入った相互参照テーブルを印刷します。

ATTRIBUTES と XREF を両方指定すると、2 つのテーブルが組み合わされます。これらのテーブル内で ID を明示的に宣言すると、コンパイラーはその DECLARE のファイル番号と行番号をリストします。コンテキストで宣言された変数は +++++ とマークされ、それ以外の暗黙的に宣言された変数は ***** とマークされます。

属性テーブル

コンパイラーは、属性 INTERNAL および REAL を決して組み込みません。それぞれと相対立する属性の EXTERNAL と COMPLEX が現れない限り、これらの属性を想定することができます。

ファイル ID に関しては、属性 FILE が常に現れ、属性 EXTERNAL は適用時に現れます。それ以外の場合は、コンパイラーは明示的に宣言された属性のみをリストします。

OPTIONS 属性は、ENTRY 属性が適用されない限り表示されず、結果として次のオプションのみが (状況に応じて) 表示されます。

- ASSEMBLER
- COBOL
- FETCHABLE
- FORTRAN
- NODESCRIPTOR
- RETCODE

コンパイラーは、配列用の次元属性をまず印刷します。境界は配列宣言のとおり印刷されますが、式はアスタリスクに置き換えられます。ただし、式がコンパイラーにより定数に還元された場合は、定数の値が印刷されます。

文字ストリング、ビット・ストリング、漢字ストリング、または区域変数の場合、コンパイラーは宣言のとおり長さを印刷しますが、式はアスタリスクに置き換えられます。ただし、式がコンパイラーにより定数に還元された場合は、定数の値が印刷されます。

相互参照テーブル

相互参照テーブルと属性テーブルを組み合わせると、名前に対応する属性のリストは、ファイル番号と行番号により識別されます。次の場合は、相互参照テーブルの **Sets:** 部分に ID が示されます。

- 代入ステートメントのターゲットの場合
- DO ループでループ制御変数として使用される場合
- ALLOCATE ステートメントまたは LOCATE ステートメントの SET オプションで使用される場合
- DISPLAY ステートメントの REPLY オプションで使用される場合

ATTRIBUTES と XREF を指定すると、2 つのテーブルが組み合わされます。

未参照の ID がある場合、それらは別個のテーブルに示されます。

集合長さテーブル

集合長さテーブルは、AGGREGATE オプションを使用して取得します。このテーブルには、配列ではなく構造体が組み込まれています。この構造体は非固定エクステンントを保持し、構造体内部の要素のサイズとオフセットは、不正確であるか、または * として指定されます。リストされた集合の場合、テーブルには次の情報が入っています。

- 宣言された集合のロケーション。
- 集合名と集合内の各要素。
- 集合の先頭からの各要素のバイト・オフセット。
- 各要素の長さ。
- 各集合、構造体、および副構造体の全長。
- 各要素の次元の合計数。

データ長テーブルに示されているデータ・オフセットを解釈する場合は注意が必要です。奇数オフセットは、ハーフワード位置合わせ、フルワード位置合わせ、またはダブルワード位置合わせが行われていないデータ・要素を必ずしも表しません。ある構造体またはその要素用に位置合わせ済みの属性を指定したり暗示したりすると、テーブルの初めに対して適切な位置合わせが行われていることをテーブルが示さなくても、適切な位置合わせ要件はその構造体内の他の要素に合致します。

2 つの構造体要素間に埋め込みがある場合は、/*PADDING*/ というコメントと、適切な診断情報が表示されます。

ステートメント・オフセット・アドレス

LIST コンパイル・オプションを使用すると、コンパイラーはコンパイラー・リストに疑似アセンブラー・リストを組み込みます。このリストには、BLKOFF コンパイラー・オプションの設定によって異なる意味を持つオフセットが命令ごとに含まれています。

- **BLKOFF** オプションを指定した場合、このオフセットは、命令が属している関数またはサブルーチン用の 1 次エントリー・ポイントからの命令のオフセットになります。つまり、このオプションのもとでは、オフセットはそれぞれの新規ブロックごとに再設定されます。
- **NOBLKOFF** オプションを指定した場合、このオフセットは、コンパイル単位の開始位置からの命令のオフセットになります。つまり、このオプションのもとではオフセットは累積されます。

疑似アセンブラー・リストには、各ブロックのコードの終わりに現行モジュールの開始位置からのブロックのオフセットも含まれています (それぞれのステートメントに表示されるオフセットを、ブロックまたはモジュールのオフセットに変換できるようにするため)。

これらのオフセットと、ランタイム・エラー・メッセージで指定されたオフセットを用いて、このメッセージが当てはまるステートメントを判別することができます。

OFFSET オプションは、それぞれのステートメントに対して、そのステートメントに属する最初の命令のオフセットを提供するテーブルを作成します。

99 ページの図 2 に示す例では、メッセージは **SUB1** の入り口からオフセット +58 で条件が生じたことを示しています。このオフセットは、コンパイラー・リストの抜粋では行番号 8 と関連して示されています。このエラーがあることを示すステートメントからの実行時出力は、99 ページの図 3 のように示されます。

```

Compiler Source
Line.File
2.0      TheMain: proc options( main );
3.0      call sub1();
4.0      Sub1: proc;
5.0      dcl (i, j) fixed bin(31);
6.0
7.0      i = 0;
8.0      j = j / i;
9.0      end Sub1;
10.0     end TheMain;

. . .

OFFSET OBJECT CODE          LINE# FILE#  P S E U D O  A S S E M B L Y  L I S T I N G
000000                                00002 |          T H E M A I N   D S      0D

. . .

00004C  5800  C1F4          00002 |          L      r0,_CEECAA_(,r12,500)
000050  5000  D098          00002 |          ST     r0,#_CEECAACRENT_1(,r13,152)
000054  5810  D098          00000 |          L      r1,#_CEECAACRENT_1(,r13,152)
000058  5820  3062          00000 |          L      r2,=Q(@STATIC)(,r3,98)
00005C  4152  1000          00000 |          LA     r5,=Q(@STATIC)(r2,r1,0)
000060  18BD                      00003 |          LR     r11,r13
000062  5800  D098          00003 |          L      r0,#_CEECAACRENT_1(,r13,152)
000066  5000  C1F4          00003 |          ST     r0,_CEECAA_(,r12,500)
00006A  58F0  3066          00003 |          L      r15,=A(SUB1)(,r3,102)
00006E  05EF                      00003 |          BALR   r14,r15
000070                      00010 |          DS     0H
000070  5810  5000          00010 |          L      r1,IBMQEFSH(,r5,0)
000074  58F0  1008          00010 |          L      r15,&Func_&WSA(,r1,8)
000078  5800  100C          00010 |          L      r0,&Func_&WSA(,r1,12)
00007C  5000  C1F4          00010 |          ST     r0,_CEECAA_(,r12,500)
000080  05EF                      00010 |          BALR   r14,r15
000082                      00010 |          DS     0H
000082  5800  D098          00002 |          L      r0,#_CEECAACRENT_1(,r13,152)
000086  5000  C1F4          00002 |          ST     r0,_CEECAA_(,r12,500)

. . .

000000                                00004 |          S U B 1       D S      0D

. . .

000048  4100  0000          00007 |          LA     r0,0
00004C  5000  D098          00007 |          ST     r0,I(,r13,152)
000050  5840  D09C          00008 |          L      r4,J(,r13,156)
000054  8E40  0020          00008 |          SRDA   r4,32
000058  1D40                      00008 |          DR     r4,r0
00005A  1805                      00008 |          LR     r0,r5
00005C  5000  D09C          00008 |          ST     r0,J(,r13,156)

```

図2. ステートメント番号の検索 (コンパイラー・リストの例)

Message :

```

IBM0301S ONCODE=320 The ZERODIVIDE condition was raised.
      From entry point SUB1 at compile unit offset +000000058 at
      address 0D3012C0.

```

図3. ステートメント番号の検索 (ランタイム・メッセージの例)

ダンプと ON ユニット SNAP エラー・メッセージ内に示された項目オフセットをこのテーブルと対比すれば、誤りのあるステートメントを見つけ出すことができます。このステートメントを識別するには、メッセージ内で指名されたブロックに関連したテーブルのセクションを探し出してから、メッセージ内のオフセット以下かまたはそれと等しい最大オフセットを見つけ出します。このオフセットに関連したステートメント番号が求める番号です。

ストレージ・オフセット・リスト

MAP コンパイル・オプションを使用すると、コンパイラーはコンパイラー・リストにストレージ・オフセット・リストを組み込みます。ストレージ・オフセット・リストは、以下のレベル 1 変数がプログラムで使用された場合に、これらのストレージ内のロケーションを示します。

- AUTOMATIC
- CONTROLLED (PARAMETER を除く)
- FETCHABLE でない STATIC (ENTRY CONSTANT を除く)

このリストには、コンパイラーが生成した一時データの一部も組み込まれます。

調節可能エクステンを持つ AUTOMATIC 変数が使用された場合は、このテーブルに、以下の 2 つの項目があります。

- 変数名の前に '_addr' が付いた項目 - 変数のアドレスのロケーションを示します
- 変数名の前に '_desc' が付いた項目 - 変数の記述子のアドレスのロケーションを示します

STATIC 変数および CONTROLLED 変数を使用した場合、ストレージ・ロケーションは RENT/NORENT コンパイラー・オプションに依存し、NORENT オプション指定した場合は、CONTROLLED 変数のロケーションも WRITABLE/NOWRITABLE コンパイラー・オプションに依存します。

ストレージ・オフセット・リストの最初の列は、*IDENTIFIER* というラベルが付けられ、第 4 列にロケーションが表示される変数の名前を含んでいます。

ストレージ・オフセット・リストの 2 番目の列は、*DEFINITION* というラベルが付けられ、"*B-F:N*" という形式のストリングを含んでいます。

- ここで、*B* は変数が宣言されたブロックの番号です。

このブロック番号に応じてブロック名をブロック名リストで検索できます。このブロック名リストはストレージ・オフセット・リスト (および、存在する場合は、疑似アセンブリー・リスト) の前にあります。

- ここで、*F* は変数が宣言されたソース・ファイルの番号です。

このファイル番号に対応するファイル名を、ファイル参照テーブル (コンパイラー・リスト全体の終わりのあたりにある) で検索できます。

- ここで、*N* は変数がソース・ファイル内で宣言されたソース行の番号です。

ストレージ・オフセット・リストの 3 番目の列は、*ATTRIBUTES* というラベルが付けられ、変数のストレージ・クラスを示します。

ストレージ・オフセット・リストの 4 番目の列は、ラベルがなく、変数のロケーションを検索する方法を示します。

このストレージ・オフセット・リストはブロック別および変数名別にソートされ、ユーザー変数のみも含まれます。また、MAP オプションを指定すると、コンパイラーは以下のマップも生成します。

- すべての STATIC 変数をリストした「静的マップ」(16 進オフセット別にソート)。
- ブロックごとにすべての AUTOMATIC 変数をリストした「自動マップ」(16 進オフセット別にソート)。

PL/I 言語のマッピング規則で、構造体を、構造体が始まっていると思われる場所から最大で 8 バイト分オフセットする必要がある場合があります。例えば、次のように宣言された AUTOMATIC 構造体 A を考えてみます。

```
dc1
  1 A,
    2 B char(2),
    2 C fixed bin(31);
```

C は、4 バイト境界に位置合わせする必要があり、この構造体には 2 バイトの埋め込みが必要です。ただし、PL/I はその 2 バイトを、B の後ではなく B の前に配置します。構造体の先頭の前にあるこの 2 バイトの「埋め込み」は、構造体の「ハング・バイト」と呼ばれます。

これらのハング・バイトは、コンパイラーによって生成された「自動マップ」にも反映されます。「ストレージ・オフセット・リスト」は、以下のように、ハング・バイトを含めずに、A のオフセットと長さを表示します。

```
A      Class = automatic,   Location = 186 : 0xBA(r13),      Length = 6
```

一方、「自動マップ」は、以下のように、ハング・バイトも含めて、A のオフセットと長さを表示します。

OFFSET (HEX)	LENGTH (HEX)	NAME
98	8	#MX_TEMP1
A0	18	_Sfi
B8	8	A

最後に、構造体におけるフィールドの最も厳しい位置合わせは 8 バイトの位置合わせであり、フィールドの最小サイズは 1 ビットであるため、考えられる最大のハングは、以下の構造体の場合のように、7 バイトと 7 ビットになります。

```
dc1
  1 X,
    2 Y bit(1),
    2 Z float bin(53);
```

ファイル参照テーブル

ファイル参照テーブルは、コンパイル時に読み取られたファイルに関する以下の情報をリストする 3 つの列で構成されます。

- コンパイラーによってファイルに割り当てられた番号

- ファイルの組み込み元データ
- ファイルの名前

最初にリストされるファイルはソース・ファイルなので、組み込み元の列にある最初の項目はブランクです。この列内の続く項目は `include` ステートメントの行番号を示し、その後にピリオドと、組み込まれるものが入っているソース・ファイルのファイル番号が付きます。

ファイルが PDS または PDSE のメンバーである場合、ファイル名は完全修飾データ・セット名とメンバー名を示します。

ファイルがサブシステム (ライブラリアンなど) を介して組み込まれた場合、ファイル名の形式は `DD:ddname(member)` になります。ただし、

- *ddname* は、`%INCLUDE` ステートメントに指定された DD 名 (あるいは、DD 名が指定されなかった場合は SYSLIB)
- *member* は、`%INCLUDE` ステートメントに指定されたメンバー名

メッセージと戻りコード

プリプロセッサまたはコンパイラーがエラーまたはエラーの可能性を検出すると、メッセージが生成されます。プリプロセッサが生成したメッセージは、プリプロセッサが処理したステートメントのリストの直後のリストに印刷されます。`%NOTE` ステートメントを使用すると、プリプロセス段階でユーザー独自のメッセージを生成できます。このようなメッセージは、特定の置換が何回行われたかを示すのに使用できます。コンパイラーが生成したメッセージはリストの最後に印刷されます。

メッセージを生成しないコンパイルであっても、コンパイラー・メッセージがリストされるはずの行に、「コンパイラー・メッセージはありません (no compiler messages)」というメッセージが含まれます。

メッセージは次のフォーマットで示されます。

PPPnnnnI X

PPP はメッセージの発信元を識別する接頭部 (例えば、IBM という接頭部の場合は PL/I コンパイラー)、nnnn は 4 桁のメッセージ番号、X は重大度コードを示します。メッセージはすべて重大度により格付けされます。重大度コードは、I、W、E、S、および U です。

各コンパイル・ジョブまたはジョブ・ステップごとにコンパイラーは、操作がどの程度成功または失敗したかをオペレーティング・システムに示すための戻りコードを生成します。z/OS の場合、このコードはステップの終わり メッセージに現れますが、その前には各ステップ別のジョブ制御ステートメントとジョブ・スケジューラー・メッセージのリストが入っています。

表 5 は、重大度コードおよび同等な戻りコードの説明です。

表 5. *PL/I* エラー・コードと戻りコードの説明

重大度 コード	戻り コード	メッセージ のタイプ	説明
I	0000	通知	コンパイルされたプログラムは正常に実行されます。非効率になる可能性のあるコードや、その他の注意すべき条件があると、コンパイラーはユーザーに通知します。
W	0004	警告	構文的には有効でも、ステートメントにエラー（警告対象）がある場合があります。コンパイルされたプログラムは正常に実行されても、予期に反する結果になったり、著しく非効率になったりする場合があります。
E	0008	エラー	コンパイラーにより修正されたエラー。コンパイルされたプログラムは正常に実行されても、予期に反する結果になる場合があります。
S	0012	重大	コンパイラーにより修正されないエラー。プログラムがコンパイルされ、オブジェクト・モジュールが生成されても、そのモジュールを使用してはなりません。
U	0016	回復不能	コンパイルを強制終了させるエラー。オブジェクト・モジュールは正常には作成されません。

注: コンパイラー・メッセージはこれらの重大度レベルによりグループ別に印刷されます。

コンパイラーは、表 6 に示すように、FLAG オプションで指定されたメッセージの重大度に等しいかより大きい重大度を持つメッセージだけをリストします。

表 6. リストされたメッセージの最低重大度を選択するための FLAG オプションの使用

メッセージのタイプ	オプション
通知	FLAG(I)
警告	FLAG(W)
エラー	FLAG(E)
重大エラー	FLAG(S)
回復不能エラー	常にリストされる

各メッセージのテキスト、説明、およびプログラマーに推奨する処置については、「Enterprise *PL/I* メッセージおよびコード」を参照してください。

第 2 章 PL/I プリプロセッサ

PL/I コンパイラを使用すると、プログラム内で必要に応じた組み込みプリプロセッサを 1 つ以上選択できます。インクルード・プリプロセッサ、マクロ・プリプロセッサ、SQL プリプロセッサ、または CICS プリプロセッサを選択でき、またこれらの呼び出し順も選択できます。

- インクルード・プリプロセッサは、特殊なインクルード・ディレクティブを処理し、外部ソース・ファイルを取り込みます。
- マクロ・プリプロセッサは、% ステートメントとマクロに基づいてソース・プログラムを変更します。
- SQL プリプロセッサは、ソース・プログラムを変更し、EXEC SQL ステートメントを PL/I ステートメントに変換します。
- CICS プリプロセッサは、ソース・プログラムを変更し、EXEC CICS ステートメントを PL/I ステートメントに変換します。

各プリプロセッサはいくつかのオプションをサポートしており、必要に合わせて処理を調整できます。

他の 3 つのコンパイル時オプション MDECK、INSOURCE、SYNTAX は、PP オプションと一緒に指定したときだけ意味を持ちます。これらのオプションの詳細については、51 ページの『MDECK』、42 ページの『INSOURCE』、および 75 ページの『SYNTAX』を参照してください。

インクルード・プリプロセッサ

インクルード・プリプロセッサを使用すると、PL/I ディレクティブ `%INCLUDE` 以外のインクルード・ディレクティブを使用して、外部ソース・ファイルをプログラムに取り込むことができます。

次の構文図は、`INCLUDE` プリプロセッサによってサポートされるオプションを示しています。

```

▶▶—PP—(—INCLUDE—(—'—ID(<directive>)—'—)—)—————▶▶

```

ID インクルード・ディレクティブの名前を指定します。最初の一続きの非空白文字としてのこのディレクティブで始まる行は、インクルード・ディレクティブとして扱われます。

指定するディレクティブの後に、1 つ以上の空白、およびインクルード・メンバー名が必要で、最後にオプションでセミコロンを付けることができます。

`ddname(membername)` の構文はサポートされません。

次の例では、1 つ目のインクルード・ディレクティブは有効で、2 つ目のものは無効です。

```

++include payroll
++include syslib(payroll)

```

次の 1 つ目の例では、`-INC` (および場合によっては先行空白) から始まる行がインクルード・ディレクティブとして扱われます。

```

pp( include( 'id(-inc)'))

```

次の 2 つ目の例では、`++INCLUDE` (および場合によっては先行空白) から始まる行がインクルード・ディレクティブとして扱われます。

```

pp( include( 'id(++include)'))

```

マクロ・プリプロセッサ

マクロを使用すると、インプリメンテーションの詳細と演算対象のデータを隠し、演算だけを表すように、共通に使用される PL/I コードを書くことができます。汎用のサブルーチンと対照的に、マクロでは個別用途のコードだけを生成できます。

コンパイラのマクロ・プリプロセス機能の説明は、「PL/I 言語解説書」にあります。

マクロ・プリプロセッサは、MACRO オプションまたは PP(MACRO) オプションを指定することによって起動できます。PP(MACRO) はオプションを付けずに指定することも、以下にリストするいずれかのオプションを付けて指定することもできます。

これらすべてのオプションのデフォルトでは、マクロ・プリプロセッサが OS PL/I V2R3 マクロ・プリプロセッサと同じように動作するようになっています。

オプションを指定する場合、リストは引用符 (単一または二重で一致させる) で囲まなければならない。例えば、FIXED(BINARY) オプションを指定するには、PP(MACRO('FIXED(BINARY)')) と指定します。

複数のオプションを指定したい場合は、コンマおよび/または、1 つ以上のブランクで分離する必要があります。例えば、CASE(ASIS) および RESCAN(UPPER) オプションを指定するには、PP(MACRO('CASE(ASIS) RESCAN(UPPER)')) または、PP(MACRO("CASE(ASIS),RESCAN(UPPER)")) と指定することができます。オプションは、任意の順序で指定することができます。

マクロ・プリプロセッサのオプション

マクロ・プリプロセッサは、次のオプションをサポートします。

FIXED

このオプションは、FIXED 変数のデフォルト基数を指定します。

➡➡FIXED—(—

DECIMAL
BINARY

) —————➡➡

DECIMAL

FIXED 変数の属性は REAL FIXED DEC(5) になります。

BINARY

FIXED 変数の属性は REAL SIGNED FIXED BIN(31) になります。

CASE

このオプションは、プリプロセッサが入力テキストを大文字に変換するかどうかを指定します。

➡➡CASE—(—

UPPER
ASIS

) —————➡➡

ASIS

入力テキストは「現状のまま」です。

UPPER

入力テキストを大文字に変換します。

INCONLY

このオプションは、プリプロセッサで %INCLUDE および %XINCLUDE ステートメントのみを処理する必要があることを指定します。このオプションが有効になると、マクロとして INCLUDE も XINCLUDE も使用できなくなります。

- プロシージャ名
- ステートメント・ラベル
- 変数名

NOINCONLY

このオプションは、プリプロセッサですべてのプリプロセッサ・ステートメントを処理する必要があり、%INCLUDE および %XINCLUDE ステートメントのみではないことを指定します。このオプションおよび INCONLY オプションは同時に指定できません。また、互換性のため、NOINCONLY がデフォルトです。

RESCAN

このオプションは、テキストの再スキャンのとき、プリプロセッサが ID の大文字小文字をどのように処理するかを指定します。

▶▶—RESCAN—(—

ASIS
UPPER

—)————▶▶

UPPER

再スキャンは大文字小文字を区別しません。

ASIS

再スキャンは大文字小文字を区別します。

このオプションの影響を見るため、次のコード・フラグメントについて考えてみましょう。

```
%dcl eins char ext;
%dcl text char ext;

%eins = 'zwei';

%text = 'EINS';
display( text );

%text = 'eins';
display( text );
```

PP(MACRO('RESCAN(ASIS)')) で 2 番目の表示ステートメントをコンパイルすると、値 text は eins に置き換えられますが、RESCAN(ASIS) が指定されていると、eins とマクロ変数 eins では前者が asis (現状のまま) で後者が uppercase (大文字) であるために一致せず、これ以上の置き換えは行われません。したがって、次のテキストが生成されます。

```
DISPLAY( zwei );

DISPLAY( eins );
```

しかし、PP(MACRO('RESCAN(UPPER)')) で 2 番目の表示ステートメントをコンパイルすると、text の値は eins に置き換えられますが、RESCAN(UPPER) が指定されていると、eins と、マクロ変数 eins の両方が uppercase (大文字) なので合致し、さらに置き換えが行われます。したがって、次のテキストが生成されます。

```
DISPLAY( zwei );
```

```
DISPLAY( zwei );
```

つまり、RESCAN(UPPER) は、大文字小文字の区別を無視し、RESCAN(ASIS) は、大文字小文字を区別します。

DBCS

このオプションは、テキスト置換時にプリプロセッサが DBCS を正規化するかどうかを指定します。

```

▶▶ DBCS ( ( INEXACT
            EXACT ) )
▶▶

```

EXACT

入力テキストは「現状のまま」です。プリプロセッサは <kk.B> と <kk>B を別の名前として扱います。

INEXACT

入力テキストは「正規化」されます。プリプロセッサは <kk.B> と <kk>B を同じ名前の 2 つのバージョンとして扱います。

マクロ・プリプロセッサの例

プリプロセッサを使用してソース・デックを作成する簡単な例を 110 ページの図 4 に示します。ソース・ステートメントは、プリプロセッサ変数 USE に割り当てられた値に応じて、サブルーチン (CITYSUB) または関数 (CITYFUN) のどちらかを表します。

SYSPUNCH に使用する DSNAME には、プリプロセッサ出力が入るソース・プログラム・ライブラリーを指定します。通常、コンパイルが続行され、プリプロセッサ出力がコンパイルされます。

```
//OPT4#8 JOB
//STEP2 EXEC IBMZC,PARM.PLI='MACRO,MDECK,NOCOMPILE,NOSYNTAX'
//PLI.SYSPUNCH DD DSN=HPU8.NEWLIB(FUN),DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN DD *
/* GIVEN ZIP CODE, FINDS CITY */
%DCL USE CHAR;
%USE = 'FUN' /* FOR SUBROUTINE, %USE = 'SUB' */ ;
%IF USE = 'FUN' %THEN %DO;
CITYFUN: PROC(ZIPIN) RETURNS(CHAR(16)) REORDER; /* FUNCTION */
          %END;
          %ELSE %DO;
CITYSUB: PROC(ZIPIN, CITYOUT) REORDER; /* SUBROUTINE */
          DCL CITYOUT CHAR(16); /* CITY NAME */
          %END;
DCL (LBOUND, HBOUND) BUILTIN;
DCL ZIPIN PIC '99999'; /* ZIP CODE */
DCL 1 ZIP_CITY(7) STATIC, /* ZIP CODE - CITY NAME TABLE */
      2 ZIP PIC '99999' INIT(
          95141, 95014, 95030,
          95051, 95070, 95008,
          0), /* WILL NOT LOOK AT LAST ONE */
      2 CITY CHAR(16) INIT(
          'SAN JOSE', 'CUPERTINO', 'LOS GATOS',
          'SANTA CLARA', 'SARATOGA', 'CAMPBELL',
          'UNKNOWN CITY'); /* WILL NOT LOOK AT LAST ONE */
DCL I FIXED BIN(31);
DO I = LBOUND(ZIP,1) TO /* SEARCH FOR ZIP IN TABLE */
      HBOUND(ZIP,1)-1 /* DON'T LOOK AT LAST ELEMENT */
      WHILE(ZIPIN ^= ZIP(I));
END;
%IF USE = 'FUN' %THEN %DO;
RETURN(CITY(I)); /* RETURN CITY NAME */
          %END;
          %ELSE %DO;
CITYOUT=CITY(I); /* RETURN CITY NAME */
          %END;
END;
```

図4. ソース・デックを作成するためのマクロ・プリプロセッサの使用

SQL プリプロセッサ

一般に、PL/I プログラムのコーディングは、プログラムが DB2 データベースにアクセスする場合もしない場合も同じです。ただし、DB2 データの検索、更新、挿入、および削除を行ったり、他の DB2 サービスを使用したりするには、SQL ステートメントを使用する必要があります。PL/I アプリケーション内では、動的および静的の EXEC SQL ステートメントを使用できます。

DB2 とやり取りするには、次の作業を行う必要があります。

- 必要な SQL ステートメントをコーディングし、EXEC SQL で区切る。
- DB2 プリコンパイラーを使用するか、DB2 for z/OS バージョン 7 リリース 1 以降を使用している場合は、PL/I PP(SQL()) コンパイラー・オプションを指定してコンパイルする。

EXEC SQL サポートを利用するためには、まず DB2 システムへのアクセス権限が必要です。権限については、担当の DB2 データベース管理者にお問い合わせください。

PL/I SQL プリプロセッサは、PL/I コンパイラーと同様に DBCS をサポートするようになりました。GRAPHIC PL/I コンパイラー・オプションが有効になっている場合は、一部のソース言語エレメントを DBCS および SBCS 文字を使用して記述できます。特に、ソース・プログラムの以下の場所で DBCS 文字を使用できます。

- コメント内
- ステートメント・ラベルおよび ID の一部として
- G または M リテラルで

プログラミングとコンパイルに関する考慮事項

PL/I SQL プリプロセッサを使用すると、組み込み SQL ステートメントを含むソース・プログラムはコンパイル時に PL/I コンパイラーによって処理され、ユーザーは別個のプリコンパイル・ステップを使用する必要がありません。別個のプリコンパイル・ステップの使用も引き続きサポートされますが、PL/I SQL プリプロセッサを使用することをお勧めします。PL/I SQL プリプロセッサを使用すると、デバッグ中に SQL ステートメントだけが表示されるように (生成された PL/I ソースは表示されない)、デバッグ・ツールによる対話式デバッグが強化されています。ただし、SQL プリプロセッサを使用するには、DB2 for z/OS バージョン 7 リリース 1 以降が必要です。

さらに、PL/I SQL プリプロセッサを使用すると、SQL プログラムに対する DB2 プリコンパイラーの制限が一部解消されます。PL/I SQL プリプロセッサを使用して SQL ステートメントを処理すると、次のことが可能になります。

- 構造化ホスト変数の完全修飾名を使用する
- トップレベルのソース・ファイル内だけでなく、ネストされた PL/I プログラムの任意のレベルで SQL ステートメントを組み込む
- ネストされた SQL INCLUDE ステートメントを使用する

PL/I SQL プリプロセッサ・オプションを使用してコンパイルを行うと、オブジェクト・モジュールやリストなど通常の PL/I コンパイラー出力とともに、DB2 データベース要求モジュール (DBRM) が生成されます。DB2 バインド・プロセスへの入力になる DBRM データ・セットには、プログラム内の SQL ステートメントとホスト変数に関する情報が入っています。ただし、バインドまたは実行時の処理の場合、DBRM の中のすべての情報が重要であるというわけではありません。例えば、DBRM の中の HOST 値は、PL/I 以外の言語を指定するものであり、気にする理由は何もありません。この値は、HOST 値に対するインストール・デフォルトとして他の言語が選択されることを意味するだけであって、PL/I のプログラムのバインドまたは実行時の処理には影響しません。

SQL プリプロセッサ

PL/I コンパイラ・リストには、PL/I SQL プリプロセッサが生成したエラー診断情報 (SQL ステートメントの構文エラーなど) が含まれています。

PL/I SQL プリプロセッサを使用するには、次のことを行う必要があります。

- ・プログラムのコンパイル時に次のオプションを指定する。

```
PP(SQL('options'))
```

このコンパイラ・オプションは、組み込み PL/I SQL プリプロセッサを起動するようにコンパイラに指示します。SQL キーワードの後に、SQL 処理オプションのリストを括弧で囲んで指定します。これらのオプションはコンマまたはスペースで区切ることができ、オプションのリストは引用符で囲む必要があります (単一引用符か二重引用符を使用し、同じ種類の引用符で囲む必要があります)。

例えば PP(SQL('DATE(USA),TIME(USA)')) は、DATE および TIME の両データ・タイプに対して USA フォーマットを使用するようにプリプロセッサに指示します。

また、LOB サポートを使用するには次のオプションを指定する必要があります。

```
LIMITS( FIXEDBIN(31,63) FIXEDDEC(31) )
```

- ・コンパイル・ステップ用の JCL に、次のデータ・セットに対する DD ステートメントを組み込む。

- DB2 ロード・ライブラリー (*prefix.SDSNLOAD*)

PL/I SQL プリプロセッサは、SQL ステートメントの処理を行うために DB2 モジュールを呼び出します。このため、DB2 ロード・ライブラリーのデータ・セット名を、コンパイル・ステップ用の STEPLIB 連結に組み込む必要があります。

- SQL INCLUDE ステートメント用のライブラリー

ソース・プログラムへの 2 次入力を指定する SQL INCLUDE *member-name* ステートメントがプログラムにある場合は、*member-name* を含むデータ・セットの名前を、コンパイル・ステップ用の SYSLIB 連結に組み込む必要があります。

- DBRM ライブラリー

PL/I プログラムをコンパイルすると、DB2 データベース要求モジュール (DBRM) が生成されるので、DBRM を書き込む先のデータ・セットを指定するために、DBRMLIB DD ステートメントが必要です。

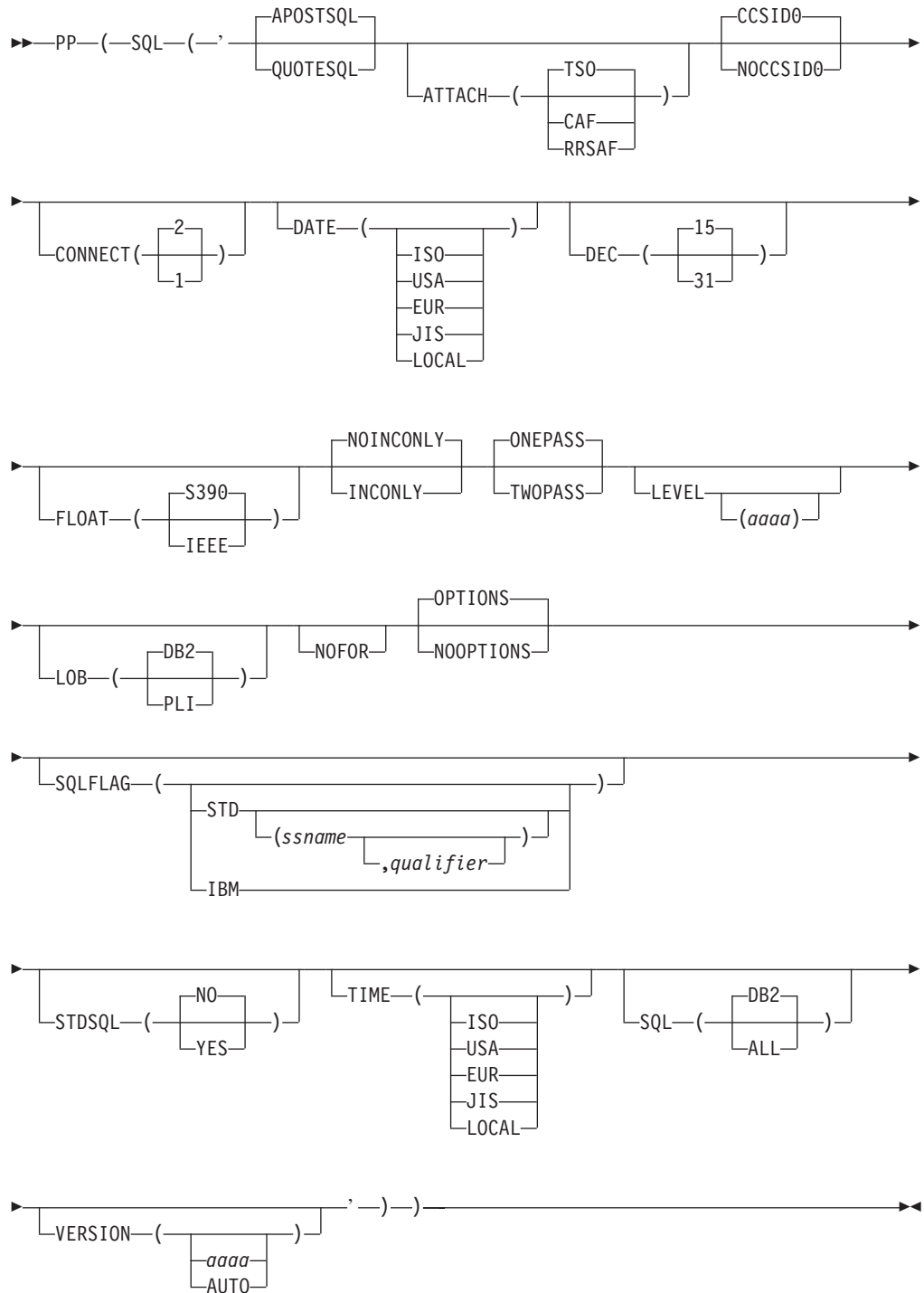
- 例えば、JCL には次のような行を指定します。

```
//STEPLIB DD DSN=DSN710.SDSNLOAD,DISP=SHR
//SYSLIB DD DSN=PAYROLL.MONTHLY.INCLUDE,DISP=SHR
//DBRMLIB DD DSN=PAYROLL.MONTHLY.DBRMLIB.DATA(MASTER),DISP=SHR
```

SQL プリプロセッサ・オプション

SQL プリプロセッサ・オプションを指定する場合、オプションのリストは引用符で囲む必要があります (単一引用符か二重引用符を使用し、同じ種類の引用符で囲む必要があります)。例えば、DATE(ISO) オプションを指定する場合、PP(SQL('DATE(ISO)')) と指定する必要があります。

次の構文図は、SQL プリプロセッサによってサポートされるオプションをすべて示しています。



これらの PL/I SQL プリプロセッサ・オプションに加えて、*PP(SQL('options'))* コンパイラ・オプションに関する DB2 コプロセッサ・オプションを渡すこと

SQL プリプロセッサ

もできます。DB2 コプロセッサ・オプションの詳細については、「DB2 *Universal Database for z/OS* アプリケーション・プログラミングおよび SQL ガイド」を参照してください。

この表では、相互排他的なオプションは縦棒 (|) によって分離されています。また括弧 ([]) は、囲まれたオプションが省略可能であることを示しています。

APOSTSQL

アポストロフィ (') を、ストリング区切り文字として、また、引用符 (") を SQL ステートメント内の SQL のエスケープ文字として認識します。

DB2 プリコンパイラを使用した、以前の PL/I プログラムとの互換性のためには、APOSTSQL を選択しなければなりません。

APOSTSQL と QUOTESQL は相互排他的なオプションです。

デフォルト設定は APOSTSQL です。

ATTACH(TSO|CAF|RRSAF)

アプリケーションが DB2 へのアクセスに使用する接続機能を指定します。

TSO、CAF および RRSAF があります。接続機能をロードするアプリケーションは、ダミーの DSNHLI エントリー・ポイントをコーディングする代わりに、このオプションを使用して適切な接続機能を指定できます。

デフォルトは ATTACH(TSO) です。

CCSID0

CCSID0 は、CCSID 値がホスト変数に割り当てられないことを指定します。

ご使用のプログラムが FOR BIT DATA 列を BIT データでないデータ・タイプで更新する場合は、CCSID0 を選択する必要があります。CCSID0 は、ホスト変数が CCSID に関連しないことを DB2 に示して、割り当てを許可します。そうでなければ、BIT データではない CCSID に関連したホスト変数が FOR BIT DATA 列に割り当てられ、DB2 にエラーが発生します。

DB2 プリコンパイラを使用した、以前の PL/I プログラムとの互換性のためには CCSID0 を選択しなければなりません。

CCSID0 と NOCCSID0 は相互排他的なオプションです。

デフォルト設定は CCSID0 です。

CONNECT(2|1)

タイプ 1 またはタイプ 2 のどちらの CONNECT ステートメント規則を適用するかを決定します。

- CONNECT(2) は、CONNECT (タイプ 2) ステートメントの規則を適用します。
- CONNECT(1) は、CONNECT (タイプ 1) ステートメントの規則を適用します。

デフォルトは CONNECT(2) です。

このオプションの詳細については、「DB2 SQL 解説書」を参照してください。

CONNECT オプションの省略形は CT です。

DATE(ISO|USA|EUR|JIS|LOCAL)

ロケーションのデフォルトとして指定されたフォーマットに関係なく、日付の出

力を常に特定のフォーマットで戻すように指定します。これらのフォーマットの詳細については、「DB2 SQL 解説書」を参照してください。

デフォルトは、DB2 のインストール時に指定した「Application Programming Defaults Panel 2」のフィールド「DATE FORMAT」の値です。

日付出力ルーチンがない場合、LOCAL オプションは使用できません。

DEC(15|31)

10 進算術演算の最大精度を指定します。

デフォルトは、DB2 のインストール時に指定した「Application Programming Defaults Panel 1」のフィールド「DECIMAL ARITHMETIC」の値です。

FLOAT(S390|IEEE)

浮動小数点ホスト変数の内容を System/390 16 進フォーマットにするか、IEEE フォーマットにするかを決定します。この FLOAT オプションが PL/I コンパイラの DEFAULT(HEXADEC|IEEE) オプションと異なる場合は、エラー・メッセージが出されます。

デフォルト設定は FLOAT(S390) です。

GRAPHIC

ソース・コードが混合データを使用する場合があること、および X'0E' と X'0F' が EBCDIC データ用の特殊制御文字 (シフトアウトおよびシフトイン) であることを指示します。

GRAPHIC と NOGRAPHIC は相互排他的なオプションです。デフォルトは、DB2 のインストール時に指定した「Application Programming Defaults Panel 1」のフィールド「MIXED DATA」の値です。

INONLY

このオプションは、SQL プリプロセッサが EXEC SQL INCLUDE ステートメントのみを処理することを指定します。このオプションが有効になっている場合は、SQL プリプロセッサによってコードは生成されません。このオプションおよび NOINONLY オプションは同時に指定できません。また、互換性のため、NOINONLY がデフォルトです。

LEVEL[(aaaa)]

モジュールのレベルを定義します。ただし、aaaa は 7 文字までの英数字値です。このオプションは、通常の使用にはお勧めしません。また、「DSNH CLIST」パネルと「DB2I」パネルは、このオプションをサポートしません。

サブオプション (aaaa) は省略できます。作成される整合性トークンはブランクになります。

LEVEL オプションの省略形は L です。

LOB(DB2|PLI)

SQL プリプロセッサによって生成される LOB (ラージ・オブジェクト) DECLARE および DEFINE ステートメントのフォーマットを決定します。

LOB(DB2) を指定すると、生成される LOB DECLARE ステートメントは、DB2 プリコンパイラが生成する形式と整合性のあるものになります。

Enterprise PL/I V3R7 以降、すべての SQL TYPE 宣言で生成されるコードは、LOCATOR、ROWID、および *LOB_FILE タイプも含め、DB2 プリコンパイラ

ーの出力とも整合性を持つことになります。 DB2 プリコンパイラーから移行する場合は、このオプションを選択してください。

例えば、このオプションを指定すると、以下のステートメントは、

```
Dcl BLOB_VAR1 Sql Type Is BLOB(32000);
```

次のように変換されます。

```
DCL
/*$*$*$
Sql Type Is BLOB(32000)
$*$*$*/
1 BLOB_VAR1,
3 BLOB_VAR1_LENGTH FIXED BIN(31),
3 BLOB_VAR1_DATA CHAR(32000);
```

LOB(PLI) を指定すると、生成される LOB DEFINE ステートメントは、ワークステーション PL/I コンパイラーが生成する形式と整合性のあるものになります。メインフレームとワークステーションの両方のプラットフォームで PL/I を使用する場合、プラットフォーム間の整合性を確保するには、このオプションを選択してください。例えば、このオプションを指定すると、以下のステートメントは、

```
Dcl BLOB_VAR1 Sql Type Is BLOB(32000);
```

次のように変換されます。

```
DEFINE STRUCTURE
1 BLOB$$x,
2 BLOB_VAR1_LENGTH FIXED BIN(31),
2 BLOB_VAR1_DATA,
3 BLOB_VAR1_DATA1(1) CHAR(32000);
DCL BLOB_VAR1 TYPE BLOB$$x ;
```

デフォルトは LOB(DB2) です。

NOCCSID0

NOCCSID0 は、ホスト変数に CCSID 値を割り当てることを許可します。

ご使用のプログラムが FOR BIT DATA 列を BIT データでないデータ・タイプで更新する場合は、CCSID0 を選択する必要があります。CCSID0 は、ホスト変数が CCSID に関連しないことを DB2 に示して、割り当てを許可します。そうでなければ、BIT データではない CCSID に関連したホスト変数が FOR BIT DATA 列に割り当てられ、DB2 にエラーが発生します。

DB2 プリコンパイラーを使用した、以前の PL/I プログラムとの互換性のためには CCSID0 を選択しなければなりません。

NOCCSID0 と CCSID0 は相互排他的なオプションです。

デフォルト設定は CCSID0 です。

NOFOR

静的 SQL の場合に NOFOR を使用すると、DECLARE CURSOR ステートメントの FOR UPDATE OF 文節の FOR UPDATE が不要になります。NOFOR を使用すると、プログラムは DB2 更新権限のある列に対して位置決め更新を実行できます。

NOFOR を使用しない場合、プログラムが DB2 更新権限のある列に対して位置決め更新を行うには、列リストのない FOR UPDATE を DECLARE CURSOR

ステートメントに指定する必要があります。列リストのない FOR UPDATE 文節は、静的または動的の SQL ステートメントに対して設定されます。

NOFOR の使用の有無に関係なく、列リストを付けた FOR UPDATE OF を指定することによって、更新対象をこの文節に指定した列だけに制限でき、また更新ロックの獲得を指定できます。

オプション STDSQL(YES) を使用すると、NOFOR が暗黙指定されます。

作成される DBRM が非常に大きい場合は、NOFOR を指定する際に余分のストレージを用意するか、列リストのない FOR UPDATE 文節を使用する必要があります。

NOGRAPHIC

ストリング内で X'OE' と X'OF' を制御文字でないものとして使用することを指示します。

GRAPHIC と NOGRAPHIC は相互排他的なオプションです。デフォルトは、DB2 のインストール時に指定した「Application Programming Defaults Panel 1」のフィールド「MIXED DATA」の値です。

NOINONLY

このオプションは、SQL プリプロセッサが EXEC SQL INCLUDE ステートメントだけではなく、すべてのステートメントを処理することを指定します。このオプションおよび INONLY オプションは同時に指定できません。また、互換性のため、NOINONLY がデフォルトです。

NOOPTIONS

SQL プリプロセッサ・オプションのリストを抑止します。

NOOPTIONS オプションの省略形は **NOOPTN** です。

ONEPASS

2 つのパスを作成するための余分な処理時間がかからないように、1 パスで処理を行います。ONEPASS オプションを使用する場合、宣言は SQL 参照の前になければなりません。

ONEPASS と TWOPASS は相互排他的なオプションです。

デフォルトは ONEPASS です。

ONEPASS オプションの省略形は **ON** です。

OPTIONS

SQL プリプロセッサ・オプションをリストします。

デフォルトは OPTIONS です。

OPTIONS オプションの省略形は **OPTN** です。

QUOTESQL

引用符 (") をストリング区切り文字として、また、アポストロフィ (') を SQL ステートメント内の SQL のエスケープ文字として認識します。

DB2 プリコンパイラを使用した、以前の PL/I プログラムとの互換性のためには、APOSTSQL を選択しなければなりません。

QUOTESQL と APOSTSQL は相互排他的なオプションです。

デフォルト設定は APOSTSQL です。

SQL(ALL|DB2)

DB2 for z/OS によって認識されない SQL ステートメントがソースに含まれているかどうかを指示します。

DB2 for z/OS 以外のサーバー上で、DRDA アクセスを使用して SQL ステートメントを実行するアプリケーション・プログラムの場合は、SQL(ALL) をお勧めします。SQL(ALL) は、プログラム内の SQL ステートメントが DB2 for z/OS 用のものとは限らないことを指定します。したがって、SQL ステートメント・プロセッサは DB2 構文規則に準拠しないステートメントを受け入れます。SQL ステートメント・プロセッサは、分散リレーショナル・データベース・アーキテクチャー (DRDA) 規則に従って SQL ステートメントを解釈し、処理します。プログラムが通常 ID として IBM SQL 予約語の使用を試みた場合、SQL ステートメント・プロセッサは通知メッセージを出します。SQL(ALL) は、SQL ステートメント・プロセッサの制限に影響を与えません。

デフォルトの SQL(DB2) を指定すると、SQL ステートメントが解釈され、構文が DB2 for z/OS による使用に適しているかどうか検査されます。データベース・サーバーが DB2 for z/OS である場合は、SQL(DB2) が推奨されます。

SQLFLAG(IBMISTD[(*ssname*[,*qualifier*])])

SQL ステートメントの構文検査に使用する標準を指定します。ステートメントが標準に準拠していない場合、SQL ステートメント・プロセッサは、出力リストに通知メッセージ (フラグ) を書き込みます。SQLFLAG オプションは、SQL や STDSQL など、他の SQL ステートメント・プロセッサ・オプションからは独立しています。

IBM は、SQL ステートメントを IBM SQL バージョン 1 の構文と突き合わせて検査します。

STD は、SQL ステートメントを 1992 年 ANSI/ISO SQL 標準のエントリー・レベルの構文と突き合わせて検査します。バージョン 7 より前のリリースでは、オプションとして 86 を使用できます。

ssname は、セマンティクス検査を要求し、指定の DB2 サブシステム名をカタログへのアクセスに使用します。*ssname* を指定しない場合、SQL ステートメント・プロセッサは構文だけを検査します。

qualifier は、フラグに使用する修飾子を指定します。*qualifier* を指定する場合は、まず *ssname* を必ず指定する必要があります。*qualifier* を指定しない場合のデフォルトは、SQL ステートメント・プロセッサを開始したプロセスの許可 ID です。

STDSQL(NO|YES)

出力ステートメントが準拠する必要がある規則を指示します。

STDSQL(YES) は、プリコンパイルされるソース・プログラム内の SQL ステートメントが、SQL 標準の特定規則に準拠する必要があることを指示します。

STDSQL(NO) は、DB2 規則への準拠を指示します。

デフォルトは、DB2 のインストール時に指定した「Application Programming Defaults Panel 2」のフィールド「STD SQL LANGUAGE」の値です。

STDSQL(YES) は、NOFOR オプションを自動的に暗黙指定します。

TIME(ISO|USA|EUR|JIS|LOCAL)

ロケーションのデフォルトとして指定されたフォーマットに関係なく、時刻の出

力を常に特定のフォーマットで戻すように指定します。これらのフォーマットの詳細については、「DB2 SQL 解説書」を参照してください。

デフォルトは、DB2 のインストール時に指定した「Application Programming Defaults Panel 2」のフィールド「TIME FORMAT」の値です。

日付出力ルーチンがない場合、LOCAL オプションは使用できません。

TWOPASS

処理を 2 パスで行います。このため、宣言は参照の前になくても構いません。

ONEPASS と TWOPASS は相互排他的なオプションです。

デフォルトは ONEPASS です。

TWOPASS オプションの省略形は TW です。

VERSION(*aaaa*lAUTO)

パッケージ、プログラム、および作成される DBRM のバージョン ID を定義します。VERSION を指定すると、SQL ステートメント・プロセッサはプログラムと DBRM 内にバージョン ID を作成します。この ID は、ロード・モジュールと DBRM のサイズに影響します。DBRM をプランまたはパッケージにバインドする際に、DB2 はバージョン ID を使用します。

プリコンパイル時にバージョンを指定しない場合、デフォルトのバージョン ID は空ストリングです。AUTO を指定すると、SQL ステートメント・プロセッサは整合性トークンを使用してバージョン ID を生成します。整合性トークンがタイム・スタンプである場合は、タイム・スタンプが ISO 文字フォーマットに変換され、バージョン ID として使用されます。使用されるタイム・スタンプは、System/370 Store Clock 値に基づいています。

DB2 V9 以降のデータベースに対して PL/I プログラムをコンパイルすると、リストに示されたオプションは、次の 2 つのカテゴリに分けられています。

使用された SQL プリプロセッサ・オプション (SQL Preprocessor Options Used)

コンパイル時に有効だった PL/I SQL プリプロセッサ・オプションのリスト。

使用された DB2 for z/OS コプロセッサ・オプション (DB2 for z/OS Coprocessor Options used)

コンパイル時に有効だった DB2 for z/OS コプロセッサ・オプションのリスト。これらのオプションの判別方法については、「DB2 Universal Database for z/OS アプリケーション・プログラミングおよび SQL ガイド」を参照してください。

PL/I アプリケーション内での SQL ステートメントのコーディング

「DB2 Universal Database for z/OS SQL 解説書」に定義されている言語を使用して、PL/I アプリケーション内で SQL ステートメントをコーディングできます。SQL コード特有の要件について、以下に説明します。

SQL 連絡域の定義

SQL ステートメントを含む PL/I プログラムには、SQLCODE 変数 (STDSQL(86) プリプロセッサ・オプションを使用する場合)、または SQL 連絡域 (SQLCA) を組み込む必要があります。図 5 に示すように、SQLCA の一部は SQLCODE 変数と SQLSTATE 変数です。

- SQLCODE の値は、各 SQL ステートメントの実行後にデータベース・サービスによって設定されます。アプリケーションは SQLCODE 値を検査して、最後の SQL ステートメントが正常に実行されたかどうか判別できます。
- SQLSTATE 変数は、SQL ステートメントの結果を分析する際に、SQLCODE 変数の代替として使用できます。SQLCODE 変数と同様に、SQLSTATE 変数は各 SQL ステートメントの実行後にデータベース・サービスによって設定されます。

SQLCA 宣言をインクルードするには、次のように EXEC SQL INCLUDE ステートメントを使用する必要があります。

```
exec sql include sqlca;
```

SQLCA 構造体は、SQL 宣言セクション内で定義してはなりません。SQLCODE と SQLSTATE の宣言の範囲には、プログラム内の SQL ステートメントの範囲がすべて含まれている必要があります。

```
Dcl
1 sqlca,
2 sqlcaid      char(8),          /* Eyecatcher = 'SQLCA ' */
2 sqlcabc      fixed binary(31), /* SQLCA size in bytes = 136 */
2 sqlcode      fixed binary(31), /* SQL return code */
2 sqlerrmc     char(70) var,     /* Error message tokens */
2 sqlerrp      char(8),          /* Diagnostic information */
2 sqlerrd(0:5) fixed binary(31), /* Diagnostic information */
2 sqlwarn,     /* Warning flags */
3 sqlwarn0     char(1),
3 sqlwarn1     char(1),
3 sqlwarn2     char(1),
3 sqlwarn3     char(1),
3 sqlwarn4     char(1),
3 sqlwarn5     char(1),
3 sqlwarn6     char(1),
3 sqlwarn7     char(1),
2 sqltext,
3 sqlwarn8     char(1),
3 sqlwarn9     char(1),
3 sqlwarna     char(1),
3 sqlstate     char(5);         /* State corresponding to SQLCODE */
```

図 5. SQLCA の PL/I 宣言

SQL 記述子域の定義

次のステートメントは SQLDA を必要とします。

```
PREPARE statement-name INTO descriptor-name FROM host-variable
EXECUTE...USING DESCRIPTOR descriptor-name
FETCH...USING DESCRIPTOR descriptor-name
OPEN...USING DESCRIPTOR descriptor-name
```

DESCRIBE *statement-name* INTO *descriptor-name*

SQLCA とは異なり、1 つのプログラム内に複数の SQLDA が存在でき、任意の有効な名前を SQLDA に付けることができます。SQLDA をインクルードするには、次のように EXEC SQL INCLUDE ステートメントを使用する必要があります。

```
exec sql include sqlda;
```

SQLDA は、SQL 宣言セクション内で定義してはなりません。

```
Dcl
1 Sqlda based(Sqldaptr),
2 sqldaid char(8), /* Eye catcher = 'SQLDA ' */
2 sqldabc fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
2 sqln fixed binary(15), /* Number of SQLVAR elements*/
2 sqld fixed binary(15), /* # of used SQLVAR elements*/
2 sqlvar(Sqlsize refer(sqln)), /* Variable Description */
3 sqltype fixed binary(15), /* Variable data type */
3 sqllen fixed binary(15), /* Variable data length */
3 sqldata pointer, /* Pointer to variable data value*/
3 sqlind pointer, /* Pointer to Null indicator*/
3 sqlname char(30) var ; /* Variable Name */

Dcl
1 Sqlda2 based(Sqldaptr),
2 sqldaid2 char(8), /* Eye catcher = 'SQLDA ' */
2 sqldabc2 fixed binary(31), /* SQLDA size in bytes=16+44*SQLN*/
2 sqln2 fixed binary(15), /* Number of SQLVAR elements*/
2 sqld2 fixed binary(15), /* # of used SQLVAR elements*/
2 sqlvar2(Sqlsize refer(sqln2)), /* Variable Description */
3 sqlbiglen,
4 sqllongl fixed binary(31),
4 sqlrsvd1 fixed binary(31),
3 sqldata1 pointer,
3 sqltname char(30) var;

dcl Sqlsize fixed binary(15); /* number of sqlvars (sqln) */
dcl Sqldaptr pointer;
dcl Sqltripld char(1) initial('3');
dcl Sqldoubled char(1) initial('2');
dcl Sqsingled char(1) initial(' ');
```

図 6. SQL 記述子域の PL/I 宣言

SQL ステートメントの組み込み

プログラムの最初のステートメントは、PROCEDURE または PACKAGE ステートメントでなければなりません。実行可能ステートメントを置くことができる任意の場所で、プログラムに SQL ステートメントを追加できます。それぞれの SQL ステートメントは EXEC (または EXECUTE) SQL で始まり、セミコロン (;) で終わる必要があります。

例えば、UPDATE ステートメントは次のようにコーディングされます。

```
exec sql update DSN8710.DEPT
set Mgrno = :Mgr_Num
where Deptno = :Int_Dept;
```

コメント: SQL ステートメントのほかに、ブランクを入力できる場所では組み込み SQL ステートメントにコメントを組み込むことができます。

Enterprise PL/I V3R6 から、SQL ステートメントに組み込まれた SQL スタイルのコメント ('--') がサポートされています。

SQL ステートメントの継続: SQL ステートメントの行継続規則は、他の PL/I ステートメントと同じです。

コードの組み込み: SQL ステートメントまたは PL/I ホスト変数の宣言ステートメントを組み込むには、ソース・コード内でステートメントを組み込む場所に、次の SQL ステートメントを配置します。

```
exec sql include member;
```

マージン: SQL ステートメントは、列 m から n までにコーディングする必要があります。ただし m と n は、MARGINS(m,n) コンパイラ・オプションで指定されます。

名前: ホスト変数には、任意の有効な PL/I 変数名を使用することができます。ホスト変数名の長さは、LIMITS(NAME(n)) コンパイラ・オプションに指定した値 n を超えてはなりません。

ステートメント・ラベル: END DECLARE SECTION ステートメント、および INCLUDE text-file-name ステートメントを例外として、実行可能 SQL ステートメントには PL/I ステートメントと同様にラベル接頭部を付けることができます。

WHENEVER ステートメント: SQL WHENEVER ステートメントの GOTO 文節のターゲットは、PL/I ソース・コード内のラベルでなければならない、WHENEVER ステートメントによって影響を受ける SQL ステートメントすべてのスコープ内になければなりません。

ホスト変数の使用

SQL ステートメント内で使用するホスト変数は、すべて明示的に宣言する必要があります。ONEPASS オプションが有効になっている場合、SQL ステートメント内で使用するホスト変数は、最初に SQL ステートメント内で使用する前に宣言しておく必要があります。

さらに、次の制限があります。

- SQL ステートメント内では、すべてのホスト変数の前にコロン (:) を付ける必要があります。
- ホスト変数の名前は、プログラム内で固有でなければなりません (ホスト変数が異なるブロックやプロシージャ内にある場合でも)。
- ホスト変数を使用する SQL ステートメントは、変数の宣言を行ったステートメントのスコープ内になければなりません。
- ホスト変数を配列として宣言することはできません。ただし、配列がホスト構造体に関連付けられている場合に、標識変数の配列を使用することは可能です。

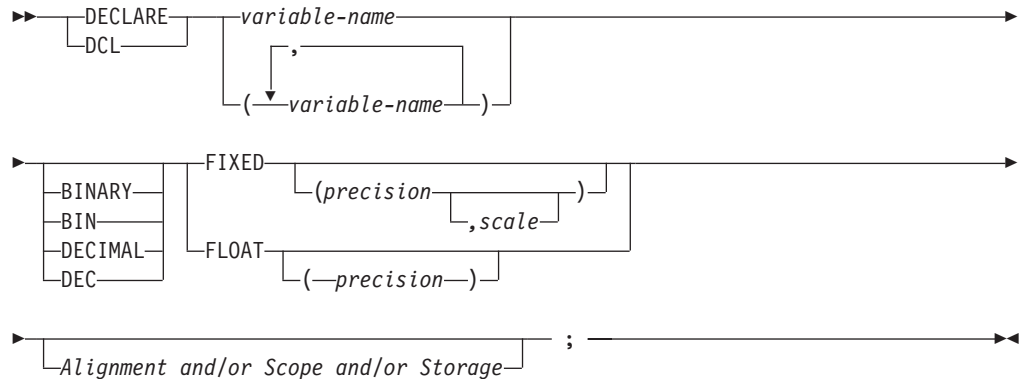
ホスト変数の宣言: ホスト変数の宣言は、通常の PL/I 変数宣言と同じ場所で行うことができます。

有効な PL/I 宣言のサブセットだけが、有効なホスト変数宣言として認識されます。プリプロセッサは、PL/I DEFAULT ステートメントに指定されたデータ属性デフォルトを使用しません。変数の宣言が認識されない場合は、ステートメントがその変数を参照すると、次のようなメッセージが出されることがあります。

'The host variable token ID is not valid'

変数の名前とデータ属性だけがプリプロセッサによって使用され、位置合わせ、スコープ、およびストレージの属性は無視されます。

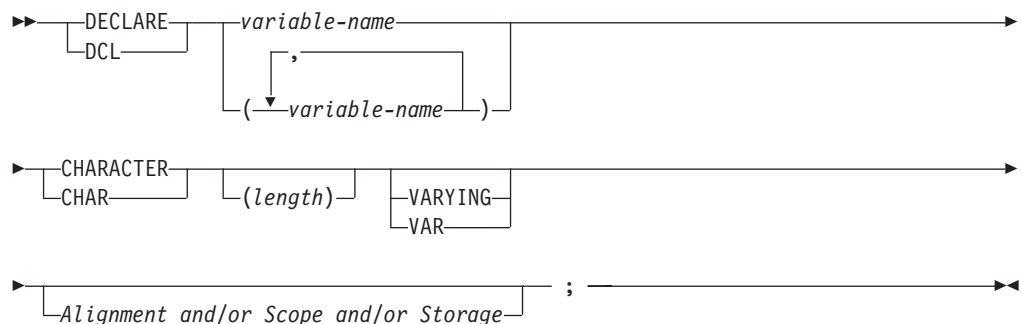
数値ホスト変数: 次の図は、有効な数値ホスト変数宣言の構文を示しています。



注

- BINARY/DECIMAL と FIXED/FLOAT は、任意の順序で指定できます。
- 精度とスケール属性を BINARY/DECIMAL の後に指定することもできます。
- *scale* の値は、DECIMAL FIXED の場合だけ指定できます。
- 詳しくは、126 ページの表 7 を参照してください。

文字ホスト変数: 次の図は、有効な文字ホスト変数の構文を示しています。

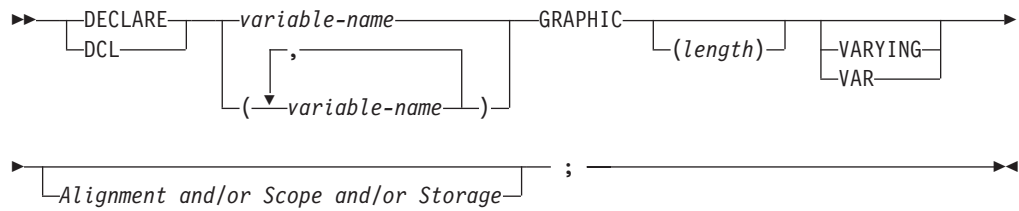


注

- 非可変文字ホスト変数の場合、*length* は SQL CHAR データの最大長を超えない定数でなければなりません。
- 可変長文字ホスト変数の場合、*length* は SQL LONG VARCHAR データの最大長を超えない定数でなければなりません。

グラフィック・ホスト変数: 次の図は、有効なグラフィック・ホスト変数の構文を示しています。

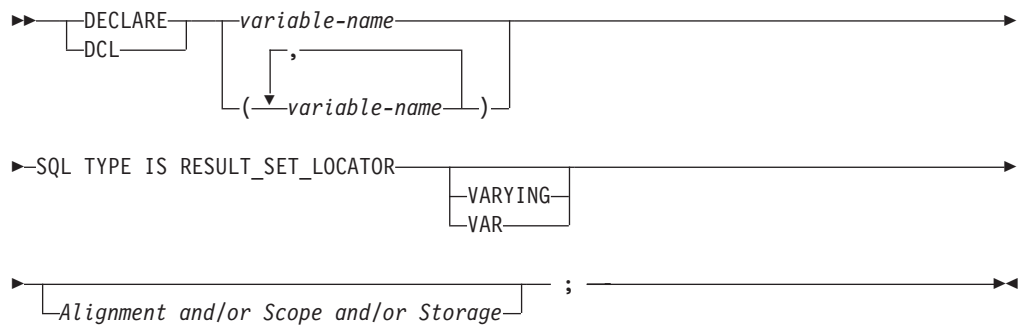
SQL プリプロセッサ



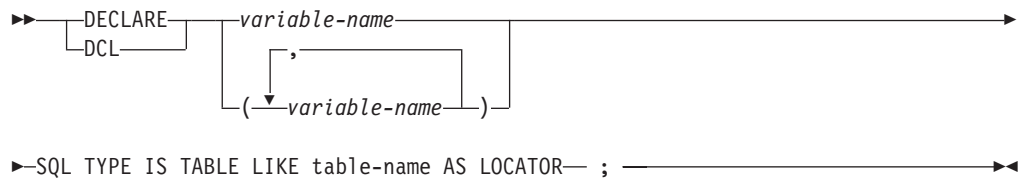
注

- 非可変グラフィック・ホスト変数の場合、*length* は SQL GRAPHIC データの最大長を超えない定数でなければなりません。
- 可変長グラフィック・ホスト変数の場合、*length* は SQL LONG VARGRAPHIC データの最大長を超えない定数でなければなりません。

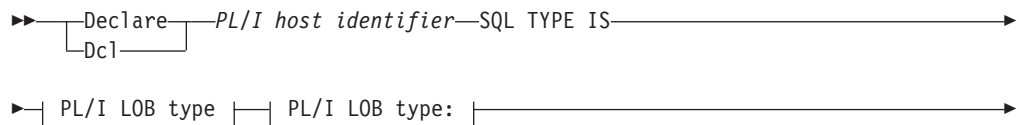
結果セット・ロケータ: 次の図は、有効な結果セット・ロケータの宣言の構文を示しています。

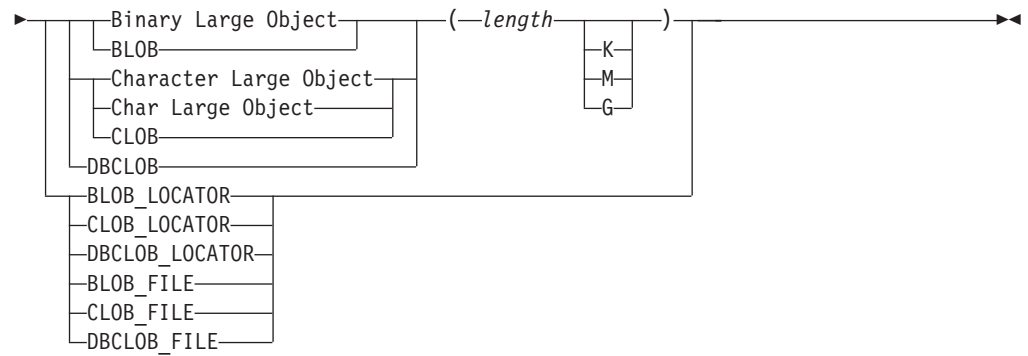


テーブル・ロケータ: 次の図は、有効なテーブル・ロケータの構文を示しています。

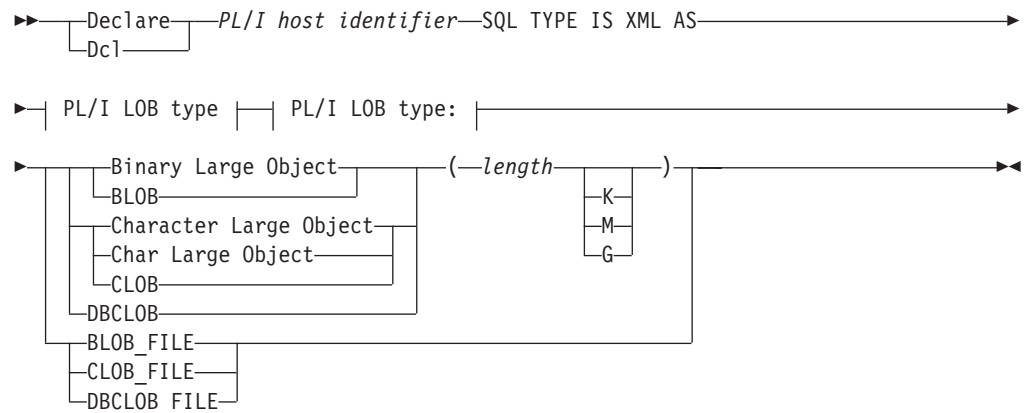


LOB 変数およびロケータ: 次の図は、BLOB、CLOB、および DBCLOB の各ホスト変数およびロケータを宣言するための構文を示しています。





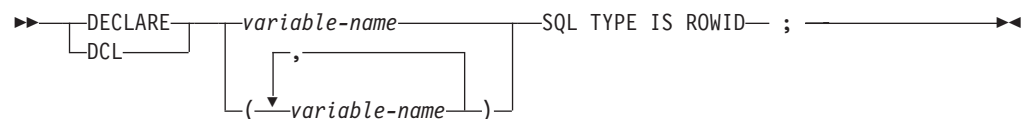
XML ファイル参照および LOB 変数: 次の図は、新しい「XML AS」ファイル参照と LOB 変数タイプの宣言の構文を示しています。



以下の定数宣言が SQL プリプロセッサによって生成されます。これらの宣言は、ファイル参照ホスト変数を使用する場合に、ファイル・オプション変数を設定するときに利用できます。

```
DCL SQL_FILE_READ      FIXED BIN(31) VALUE(2);
DCL SQL_FILE_CREATE    FIXED BIN(31) VALUE(8);
DCL SQL_FILE_OVERWRITE FIXED BIN(31) VALUE(16);
DCL SQL_FILE_APPEND    FIXED BIN(31) VALUE(32);
```

ROWID: 次の図は、ROWID 変数の有効な宣言の構文を示しています。



SQL および PL/I の同等なデータ・タイプの判別

ホスト変数の基本 SQLTYPE および SQLLEN は、次の表のとおりに決定されます。ホスト変数が標識変数とともに指定される場合、SQLTYPE は基本 SQLTYPE に 1 を加えた値です。

SQL プリプロセッサ

表 7. PL/I 宣言から生成される SQL データ・タイプ

PL/I データ・タイプ	ホスト変数の SQLTYPE	ホスト変数の SQLLEN	SQL データ・タイプ
BIN FIXED(n), $n < 16$	500	2	SMALLINT
BIN FIXED(n), n の範囲は 16 から 31 まで	496	4	INTEGER
BIN FIXED(n), n の範囲は 32 から 63 まで	492	8	BIGINT
DEC FIXED(p,s) $0 \leq p \leq 15$ および $0 \leq s \leq p$	484	p (バイト 1) s (バイト 2)	DECIMAL(p,s)
BIN FLOAT(p), $1 \leq p \leq 21$	480	4	REAL または FLOAT(n) $1 \leq n \leq 21$
BIN FLOAT(p), $22 \leq p \leq 53$	480	8	DOUBLE PRECISION または FLOAT(n) $22 \leq n \leq 53$
DEC FLOAT(m), $1 \leq m \leq 6$	480	4	FLOAT (単精度)
DEC FLOAT(m), $7 \leq m \leq 16$	480	8	FLOAT (倍精度)
CHAR(n),	452	n	CHAR(n)
CHAR(n) VARYING, $1 \leq n \leq 255$	448	n	VARCHAR(n)
CHAR(n) VARYING, $n > 255$	456	n	VARCHAR(n)
GRAPHIC(n), $1 \leq n \leq 127$	468	n	GRAPHIC(n)
GRAPHIC(n) VARYING, $1 \leq n \leq 2000$	464	n	VARGRAPHIC(n)
GRAPHIC(n) VARYING, $n > 2000$	472	n	LONG VARGRAPHIC

表 8. メタ PL/I 宣言から生成される SQL データ・タイプ

PL/I データ・タイプ	ホスト変数の SQLTYPE	ホスト変数の SQLLEN	SQL データ・タイプ
SQL TYPE IS BLOB(n) $1 < n < 2147483647$	404	n	BLOB(n)
SQL TYPE IS CLOB(n) $1 < n < 2147483647$	408	n	CLOB(n)
SQL TYPE IS DBCLOB(n) $1 < n < 1073741823$ (2)	412	n	DBCLOB(n) (2)
SQL TYPE IS ROWID	904	40	ROWID
SQL TYPE IS VARBINARY(n) $1 < n < 32704$	908	n	VARBINARY(n)
SQL TYPE IS BINARY(n) $1 < n < 255$	912	n	BINARY(n)
SQL TYPE IS BLOB_FILE	916	267	BLOB ファイル参照 (1)
SQL TYPE IS CLOB_FILE	920	267	CLOB ファイル参照 (1)
SQL TYPE IS DBCLOB_FILE	924	267	DBCLOB ファイル参照 (1)
SQL TYPE IS BLOB_LOCATOR	960	4	BLOB ロケータ (1)
SQL TYPE IS CLOB_LOCATOR	964	4	CLOB ロケータ (1)
SQL TYPE IS DBCLOB_LOCATOR	968	4	DBCLOB ロケータ (1)

表 8. メタ PL/I 宣言から生成される SQL データ・タイプ (続き)

PL/I データ・タイプ	ホスト変数の SQLTYPE	ホスト変数の SQLLEN	SQL データ・タイプ
SQL TYPE IS RESULT_SET_LOCATOR	972	4	結果セット・ロケータ
SQL TYPE IS TABLE LIKE table-name AS LOCATOR	976	4	テーブル・ロケータ (1)

注:

1. このデータ・タイプを列タイプとして使用しないでください。
2. n は 2 バイト文字の数です。

次の表を使用して、特定の SQL データ・タイプと同等な PL/I データ・タイプを判別できます。

表 9. SQL データ・タイプと PL/I 宣言の対応

SQL データ・タイプ	同等な PL/I 宣言	注
SMALLINT	BIN FIXED(15)	
INTEGER	BIN FIXED(31)	
BIGINT	BIN FIXED(63)	
DECIMAL(p,s)	DEC FIXED(p) または DEC FIXED(p,s)	p = precision および s = scale; $1 \leq p \leq 31$ および $0 \leq s \leq p$
REAL または FLOAT(n)	BIN FLOAT(p) または DEC FLOAT(m)	$1 \leq n \leq 21$ 、 $1 \leq p \leq 21$ および $1 \leq m \leq 6$
DOUBLE PRECISION、DOUBLE、または FLOAT(n)	BIN FLOAT(p) または DEC FLOAT(m)	$22 \leq n \leq 53$ 、 $22 \leq p \leq 53$ および $7 \leq m \leq 16$
CHAR(n)	CHAR(n)	$1 \leq n \leq 32767$
VARCHAR(n)	CHAR(n) VAR	
GRAPHIC(n)	GRAPHIC(n)	n は、2 バイト文字の数を示す (バイト数ではない) 1 から 127 までの正整数
VARGRAPHIC(n)	GRAPHIC(n) VAR	n は、2 バイト文字の数を示す (バイト数ではない) 正整数。 $1 \leq n \leq 2000$
LONG VARGRAPHIC	GRAPHIC(n) VAR	$n > 2000$
DATE	CHAR(n)	n の最小値は 10
TIME	CHAR(n)	n の最小値は 8
TIMESTAMP	CHAR(n)	n の最小値は 26

表 10. SQL データ・タイプとメタ PL/I 宣言の対応

SQL データ・タイプ	同等な PL/I 宣言	注
結果セット・ロケータ	SQL TYPE IS RESULT_SET_LOCATOR	このデータ・タイプは、結果セットの受け取りに だけ使用します。このデータ・タイプを列タイプ として使用しないでください。

SQL プリプロセッサ

表 10. SQL データ・タイプとメタ PL/I 宣言の対応 (続き)

SQL データ・タイプ	同等な PL/I 宣言	注
テーブル・ロケータ	SQL TYPE IS TABLE LIKE <i>table-name</i> AS LOCATOR	このデータ・タイプは、ユーザー定義の関数またはストアド・プロシージャ内だけで、変位テーブルの行を受け取るために使用します。このデータ・タイプを列タイプとして使用しないでください。
BLOB ロケータ	SQL TYPE IS BLOB_LOCATOR	このデータ・タイプは、BLOB 列のデータを操作するためにだけ使用します。このデータ・タイプを列タイプとして使用しないでください。
CLOB ロケータ	SQL TYPE IS CLOB_LOCATOR	このデータ・タイプは、CLOB 列のデータを操作するためにだけ使用します。このデータ・タイプを列タイプとして使用しないでください。
DBCLOB ロケータ	SQL TYPE IS DBCLOB_LOCATOR	このデータ・タイプは、DBCLOB 列のデータを操作するためにだけ使用します。このデータ・タイプを列タイプとして使用しないでください。
BLOB ファイル参照	SQL TYPE IS BLOB_FILE	このデータ・タイプは、BLOB ファイルの参照としてのみ使用します。このデータ・タイプを列タイプとして使用しないでください。
CLOB ファイル参照	SQL TYPE IS CLOB_FILE	このデータ・タイプは、CLOB ファイルの参照としてのみ使用します。このデータ・タイプを列タイプとして使用しないでください。
DBCLOB ファイル参照	SQL TYPE IS DBCLOB_FILE	このデータ・タイプは、DBCLOB ファイルの参照としてのみ使用します。このデータ・タイプを列タイプとして使用しないでください。
BLOB(<i>n</i>)	SQL TYPE IS BLOB(<i>n</i>)	1< <i>n</i> <2147483647
CLOB(<i>n</i>)	SQL TYPE IS CLOB(<i>n</i>)	1< <i>n</i> <2147483647
DBCLOB(<i>n</i>)	SQL TYPE IS DBCLOB(<i>n</i>)	<i>n</i> は 2 バイト文字の数です。 1< <i>n</i> <1073741823
ROWID	SQL TYPE IS ROWID	
XML AS	SQL TYPE IS XML AS ...	XML バージョンの BLOB、CLOB、DBCLOB、BLOB_FILE、CLOB_FILE、または DBCLOB_FILE を記述するのに使用します。

ラージ・オブジェクト (LOB) サポートに関する追加情報

LOB に関する一般情報

LOBS、CLOBS、および BLOBS は、最大で 2,147,483,647 バイト (2 ギガバイト) の長さにすることができます。2 バイト CLOBS は、1,073,741,823 文字 (1 ギガバイト) の長さにすることができます。

BLOB、CLOB、および DBCLOB データ・タイプ

BLOB、CLOB、および DBCLOB の変数宣言は、PL/I SQL プリプロセッサによって変換されます。

例えば、次のように宣言したとします。

```
DCL my-identifier-name SQL TYPE IS lob-type-name (length);
```

SQL プリプロセッサは、LOB() プリコンパイラ・オプションの設定に応じて、この宣言を以下の構造体の 1 つに変換します。LOB(DB2) が指定され、LOB サイズが 32767 バイト以下である場合、生成される構造体は、以下のようになります。

```
DCL
/*$$$
SQL TYPE IS lob-type-name (length)
$$$*/
1 my-identifier-name,
3 my-identifier-name_LENGTH FIXED BIN(31),
3 my-identifier-name_DATA CHAR(size1);
```

LOB(DB2) が指定され、サイズが 32767 バイトより大きい場合、生成される構造体は、以下のようになります。

```
DCL
/*$$$
SQL TYPE IS lob-type-name (length)
$$$*/
1 my-identifier-name,
3 my-identifier-name_LENGTH FIXED BIN(31),
3 my-identifier-name_DATA,
4 my-identifier-name_DATA1(size1) CHAR(32767),
4 my-identifier-name_DATA2 CHAR(size2);
```

LOB(PLI) が指定され、LOB サイズが 32767 バイト以下である場合、生成される構造体は、以下のようになります。

```
DEFINE STRUCTURE
1 lob-type$$x,
2 my-identifier-name_LENGTH FIXED BIN(31),
2 my-identifier-name_DATA,
3 my-identifier-name_DATA1 CHAR(size1);
DCL my-identifier-name TYPE lob-type$$x;
```

LOB(PLI) が指定され、LOB サイズが 32767 バイトより大きい場合、生成される構造体は、以下のようになります。

```
DEFINE STRUCTURE
1 lob-type$$x,
2 my-identifier-name_LENGTH FIXED BIN(31),
2 my-identifier-name_DATA,
3 my-identifier-name_DATA1(size1) CHAR(32767),
3 my-identifier-name_DATA2 CHAR(size2),
DCL my-identifier-name TYPE lob-type$$x;
```

これらの構造体の中で、my-identifier-name は PL/I ホスト ID の名前、lob-type\$\$x はプリプロセッサが生成した名前です。size1 は、length/32767 の値を切り捨てた整数値です。size2 は、length/32767 の剰余です。

DBCLOB データ・タイプの場合、生成される構造体は少し異なります。2 バイト文字はそれぞれが 2 バイトなので、size1 は length/16383 の値を切り捨てた整数値であり、size2 は length/16383 の剰余です。

BLOB、CLOB、および DBCLOB LOCATOR データ・タイプ

BLOB、CLOB、および DBCLOB ロケータの変数宣言も、PL/I SQL プリプロセッサによって変換されます。

例えば、次のように宣言したとします。

```
DCL my-identifier-name SQL TYPE IS lob-type_LOCATOR;
```

SQL プリプロセッサ

SQL プリプロセッサは、LOB() プリコンパイラ・オプションの設定に応じて、この宣言を以下の構造体の 1 つに変換します。LOB(DB2) が指定されている場合、生成されるコードは次のようになります。

```
DCL
/****$
SQL TYPE IS lob-type_LOCATOR
$***$/
my-identifier-name FIXED BIN(31);
```

LOB(PLI) が指定されている場合、SQL プリプロセッサはこの宣言を次のコードに変換します。

```
DEFINE ALIAS lob-type_LOCATOR FIXED BIN(31);

Dcl my-identifier-name TYPE lob-type_LOCATOR;
```

この例の中で、my-identifier-name は PL/I ホスト ID、lob-type_LOCATOR は LOB のタイプとストリング LOCATOR からなる、プリプロセッサが生成した名前です。

LOB サポートのための PL/I 変数宣言

次の例は、サンプルの PL/I 変数宣言と、LOB サポートのための対応する変換を示しています。これらの例はすべて、デフォルトの LOB(DB2) プリコンパイラ・オプションを指定してコンパイルされています。

例 1:

```
DCL my_blob SQL TYPE IS BLOB(2000);
```

変換後:

```
DCL
/****$
SQL TYPE IS BLOB(2000)
$***$/
1 MY_BLOB,
3 MY_BLOB_LENGTH FIXED BIN(31),
3 MY_BLOB_DATA CHAR(2000);
```

例 2:

```
DCL my_dbclob SQL TYPE IS DBCLOB(4000K);
```

変換後:

```
DCL
/****$
SQL TYPE IS DBCLOB(4000K)
$***$/
1 MY_DBCLOB,
3 MY_DBCLOB_LENGTH FIXED BIN(31),
3 MY_DBCLOB_DATA,
4 MY_DBCLOB_DATA1(250) GRAPHIC(16383),
4 MY_DBCLOB_DATA2 GRAPHIC(250);
```

例 3:

```
DCL my_clob_locator SQL TYPE IS CLOB_LOCATOR;
```

変換後:

```
DEFINE ALIAS CLOB_LOCATOR FIXED BIN(31);
DCL my_clob_locator TYPE CLOB_LOCATOR;
```

SQL データ・タイプと PL/I データ・タイプの互換性の判別

SQL ステートメント内の PL/I ホスト変数は、ホスト変数を使用する列のタイプと互換性がなければなりません。

- 数値データ・タイプは、相互に互換性があります。SMALLINT、INTEGER、DECIMAL、または FLOAT の列は、BIN FIXED(15)、BIN FIXED(31)、DECIMAL(*p,s*)、BIN FLOAT(*n*) (ただし *n* は 22 から 53 まで)、または DEC FLOAT (*m*) (ただし *m* は 7 から 16 まで) の PL/I ホスト変数と互換性があります。
- 文字データ・タイプは、相互に互換性があります。CHAR または VARCHAR の列は、固定長または可変長の PL/I 文字ホスト変数と互換性があります。
- Datetime データ・タイプは、文字ホスト変数と互換性があります。DATE、TIME、または TIMESTAMP の列は、固定長または可変長の PL/I 文字ホスト変数と互換性があります。

必要に応じて、データベース・マネージャーは自動的に固定長文字ストリングを可変長ストリングに変換したり、可変長ストリングを固定長文字ストリングに変換したりします。

ホスト構造体の使用

構造体または共用体でないメンバーを持つ構造体の名前を、PL/I ホスト構造体の名前にすることができます。次に例を示します。

```

dcl 1 A,
    2 B,
    3 C1 char(...),
    3 C2 char(...);

```

この例で、B はスカラー C1 と C2 からなるホスト構造体の名前です。

ホスト構造体は 2 レベルに制限されます。ホスト構造体は、ホスト変数の名前付き集合と考えることができます。

宣言の終わりにセミコロンを入力することによって、ホスト構造体変数を区切る必要があります。次に例を示します。

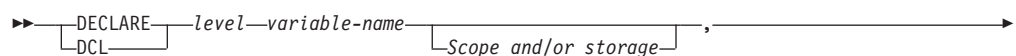
```

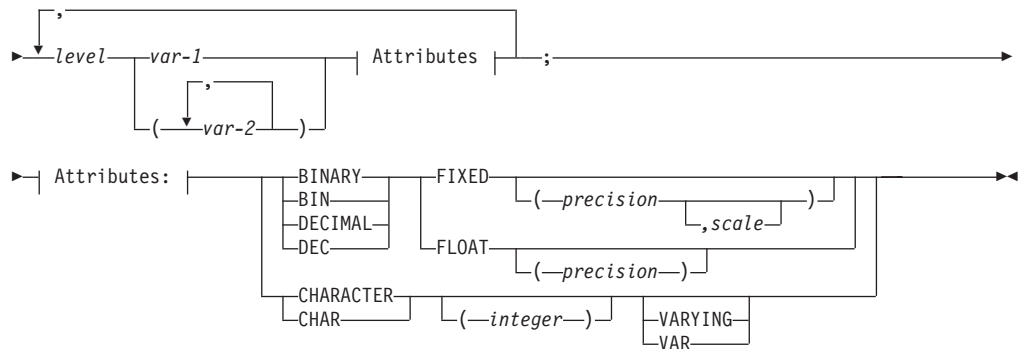
dcl 1 A,
    2 B char,
    2 (C, D) char;
dcl (E, F) char;

```

ホスト変数属性は、PL/I で許容できる任意の順序で指定できます。例えば、BIN FIXED(31)、BINARY FIXED(31)、BIN(31) FIXED、および FIXED BIN(31) はすべて許容できます。

次の図は、有効なホスト構造体の構文を示しています。





標識変数の使用

標識変数は 2 バイトの整数 (BIN FIXED(15)) です。検索時には、関連したホスト変数にヌル値が割り当てられているかどうかを示すために、標識変数が使用されます。列への割り当て時には、ヌル値を割り当てる必要があるかどうかを示すために、負の標識変数が使用されます。

標識変数はホスト変数と同じ方法で宣言され、両変数の宣言はプログラマーの裁量でどのように組み合わせることもできます。標識変数配列が使用される場合、その配列内のすべての標識変数は、最初のメンバーから開始して、順番に指定されていなければなりません。順番になっていない指標変数があると、エラーとしてフラグが立てられます。

次のステートメントがあるとして。

```
exec sql fetch Cls_Cursor into :Cls_Cd,
                                :Day :Day_Ind,
                                :Bgn :Bgn_Ind,
                                :End :End_Ind;
```

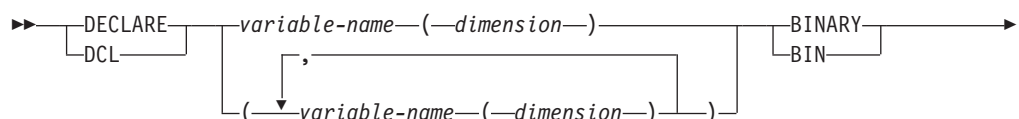
変数は次のように宣言できます。

```
exec sql begin declare section;
dcl Cls_Cd      char(7);
dcl Day         bin fixed(15);
dcl Bgn         char(8);
dcl End         char(8);
dcl (Day_Ind, Bgn_Ind, End_Ind) bin fixed(15);
exec sql end declare section;
```

次の図は、有効な標識変数の構文を示しています。



次の図は、有効な標識配列の構文を示しています。



▶FIXED(15)―▶

ホスト構造体の例

次の例では、ホスト構造体と標識配列の宣言を示し、その次に 2 つの同等な SQL ステートメントを示しています。これらのステートメントのどちらかを使用して、ホスト構造体にデータを取り込むことができます。

```

dcl 1 games,
    5 sunday,
        10 opponents char(30),
        10 gtime      char(10),
        10 tv         char(6),
        10 comments  char(120) var;
dcl indicator(4) fixed bin (15);

exec sql
    fetch cursor_a
    into :games.sunday.opponents:indicator(1),
        :games.sunday.gtime:indicator(2),
        :games.sunday.tv:indicator(3),
        :games.sunday.comments:indicator(4);

exec sql
    fetch cursor_a
    into :games.sunday:indicator;

```

最初の例で、その配列内のすべての標識変数は、最初のメンバーから開始して、順番に指定されることに注意してください。順番になっていない指標変数があると、エラーとしてフラグが立てられます。

コンパイラ・ユーザー出口 (IBMUEXIT) での SQL プリプロセッサの使用

IBM 提供のコンパイラ・ユーザー出口 (IBMUEXIT) を使用して、メッセージを抑止したり、メッセージの重大度を変更したりすることができます。SQL プリプロセッサでは、プリプロセッサ自体と DB2 コプロセッサの両方のメッセージを扱うので、次のことに注意する必要があります。

SQL プリプロセッサは IBM7021 から IBM7999 の範囲のメッセージを生成します。これらのメッセージの小サブセットである IBM7040、IBM7041、IBM7042、IBM7043、および IBM7044 は「特殊」メッセージで、DB2 コプロセッサから戻されたメッセージ情報を中継しています。これらの「特殊」メッセージには 2 つのメッセージ番号が含まれています。1 つは SQL プリプロセッサが割り当てた IBM704x (x は DB2 メッセージの重大度による) の番号で、もう 1 つは DB2 メッセージ・テキストを持つ DB2 メッセージです。特殊 SQL プリプロセッサ・メッセージのこの小さなサブセットを、ユーザー出口を使用して変更または抑止することはできません。これらの各メッセージは、さまざまな個別の DB2 コプロセッサ・メッセージに使用されるからです。ただし、その他の「通常」SQL プリプロセッサ・メッセージは、コンパイラ・ユーザー出口を使用して変更または抑止することができます。

SQL プリプロセッサ

ここで、これらの 2 つのタイプの SQL プリプロセッサ・メッセージの重大度を変更する方法の例を示します。

まず、「通常」SQL プリプロセッサ・メッセージを考えてみましょう。IBM7028 を例に取ります。

```
IBM7028I W Reference var-name is ambiguous.
```

IBM7028 のような「通常」SQL プリプロセッサ・メッセージの重大度を 4 (警告) から 12 (重大) に変更する場合は、SYSUEXIT DD カードを次のように変更します。

Fac Id	Msg No	Severity	Suppress	Comment
'SQL'	7028	12	0	Ambiguous reference

この例のフィールドと値の詳細については、459 ページの『第 20 章 ユーザー出口の用法』を参照してください。

次に、「特殊」SQL プリプロセッサ・メッセージの 1 つである IBM7041 を考えてみましょう。

```
IBM7041I W      DSNH527I DSNHOPTS  THE PRECOMPILER ATTEMPTED TO USE
                THE DB2-SUPPLIED DSNHDECP MODULE
```

これには 2 つのメッセージ番号が含まれていることに注目してください。

「IBM7041」は SQL プリプロセッサが割り当てたもので、「DSNH527」は DB2 コプロセッサが割り当てたものです。IBM7041 のような「特殊」SQL プリプロセッサ・メッセージの重大度を 4 (警告) から 12 (重大) に変更する場合は、SYSUEXIT DD カードを次のように変更します。

Fac Id	Msg No	Severity	Suppress	Comment
'SQL'	527	12	0	DB2 coprocessor message

両方のタイプの SQL プリプロセッサ・メッセージの機能 ID が同じであることに注目してください。ただし、「特殊」SQL プリプロセッサ・メッセージの場合は、DB2 コプロセッサが割り当てたメッセージ番号 (この場合は「527」) を、変更するメッセージ番号として使用することになります。

DECLARE STATEMENT ステートメント

プリプロセッサは、DECLARE STATEMENT ステートメントをすべて無視します。

CICS プリプロセッサ

CICS 環境でトランザクションとして実行される PL/I アプリケーション内では、EXEC CICS ステートメントを使用できます。

PP(CICS) オプションを指定しない場合、EXEC CICS ステートメントが構文解析され、ステートメント内の変数参照が検証されます。変数参照が正しい場合、NOCOMPILE オプションが有効であれば、メッセージは出されません。CICS 変換プログラムを呼び出さず、COMPILE オプションが有効になっていると、コンパイラは S レベル・メッセージを発行します。

PP オプションの CICS サブオプションを指定すると、コンパイラーは、CICS プリプロセッサを呼び出します。互換性のため、コンパイラーが、CICS、XOPTS、または XOPTS オプションのうちのどれかを見つけた場合も、CICS プリプロセッサを呼び出します。しかし、これらのオプションの 1 つを指定した上で、PP(CICS) オプションも指定してはいけません。

プログラミングとコンパイルに関する考慮事項

CICS の環境で実行するプログラムを開発する場合は、次の 2 つの方法のどちらかで EXEC CICS コマンドをすべて変換する必要があります。

- PL/I コンパイルの前のジョブ・ステップで、CICS 提供のコマンド言語変換プログラムを使用する
- PL/I コンパイル時に、PL/I CICS プリプロセッサを使用する (CICS TS 2.2 以降が必要)

CICS プリプロセッサを使用するには、さらに PP(CICS) コンパイル時オプションも指定する必要があります。

PP(CICS) オプションのサブオプションの 1 つとして CICS を指定しないかぎり、コンパイラーは、ソースの中に EXEC CICS ステートメントがあると、すべてフラグを立てます。同様に、EXEC CPSM または EXEC DLI ステートメントについても、PP(CICS) オプションのサブオプションとしてそれぞれ CPSM または DLI を指定しないと、これらのステートメントにフラグが立てられます。

CICS プログラムが MAIN プロシージャである場合、SYSTEM(CICS) オプションまたは SYSTEM(MVS) オプションも指定してコンパイルする必要があります。SYSTEM(MVS) を指定してコンパイルする場合は、ランタイム APAR PQ91318 に対応する PTF を適用する必要があります。このオプションでは NOEXECOPS が暗黙指定され、MAIN プロシージャに渡されるパラメーターはすべて POINTER でなければなりません。SYSTEM コンパイル時オプションに関する説明については、77 ページの『SYSTEM』を参照してください。

CICS プログラムを再入可能にしたい場合で、ご使用のプログラムが FILE または CONTROLLED 変数を使用している場合は、それも NOWRITABLE でコンパイルしなければなりません。

CICS プログラムで、EXEC CICS ステートメントが含まれたファイルがインクルードされているか、またはこのステートメントが含まれたマクロが使用されている場合は、コードを変換 (上記のいずれかの方法で) する前に MACRO プリプロセッサも実行する必要があります。CICS プリプロセッサを使用する場合、次の例に示すような PP オプション 1 つを使用してこのプリプロセッサを指定できます。

```
pp (macro(...) cics(...))
```

最後に、CICS プリプロセッサを使用するためには、PL/I コンパイラー用の STEPLIB DD に CICS SDFHLOAD データ・セットが含まれている必要があります。

CICS プリプロセッサのオプション

CICS 変換プログラムは、数多くのオプションをサポートしています。これらのオプションについては、「*CICS Transaction Server for z/OS CICS アプリケーション・プログラミング・ガイド*」を参照してください。

これらのオプションは引用符 (単一または二重で一致させる) で囲む必要がありますので注意してください。例えば、EDF オプションを指定して CICS プリプロセッサを呼び出すには、オプションを PP(CICS('EDF')) と指定します。

PL/I アプリケーション内での CICS ステートメントのコーディング

「オープン・システム CICS (AIX 版) アプリケーション・プログラミング・ガイド」に定義されている言語を使用して、PL/I アプリケーション内で CICS ステートメントをコーディングできます。CICS コード特有の要件について、以下に説明します。

CICS ステートメントの組み込み

組み込みプリプロセッサではなく、CICS 変換プログラムを使用したい場合は、ユーザーの PL/I プログラムの最初のステートメントは PROCEDURE ステートメントでなければなりません。実行可能ステートメントを置くことができる任意の場所で、プログラムに CICS ステートメントを追加できます。それぞれの CICS ステートメントは EXEC (または EXECUTE) CICS で始まり、セミコロン (;) で終わる必要があります。

例えば、GETMAIN ステートメントは次のようにコーディングされます。

```
EXEC CICS GETMAIN SET(BLK_PTR) LENGTH(STG(BLK));
```

コメント: CICS ステートメントのほかに、ブランクを入力できる場所では組み込み CICS ステートメントに PL/I コメントを組み込むことができます。

CICS ステートメントの継続: CICS ステートメントの行継続規則は、他の PL/I ステートメントと同じです。

コードの組み込み: 組み込むコードに EXEC CICS ステートメントが含まれる場合、または EXEC CICS ステートメントを生成する PL/I マクロをプログラムで使用する場合は、次のどちらかを使用する必要があります。

- MACRO コンパイル時オプション
- PP オプションの MACRO オプション (PP オプションの CICS オプションの前)

マージン: CICS ステートメントは、MARGINS コンパイル時オプションに指定された列の範囲内でコーディングする必要があります。

ステートメント・ラベル: EXEC CICS ステートメントには、PL/I ステートメントと同様にラベル接頭部を付けることができます。

PL/I を使用した CICS トランザクションの作成

PL/I を CICS 機能と組み合わせて使用して、CICS サブシステム用のアプリケーション・プログラム (トランザクション) を作成することができます。この場合、通常はオペレーティング・システムによって直接提供される機能が、CICS によって

PL/I プログラムに提供されます。これらの機能には、ほとんどのデータ管理機能や、ジョブとタスクの管理機能すべてが含まれます。

次の PL/I CICS プログラムの制限を順守する必要があります。

- マクロ・レベル CICS はサポートされません。
- 次のものを除く PL/I 入出力は使用できません。
 - PUT FILE(SYSPRINT)
 - DISPLAY
 - CALL PLIDUMP
- PLISRTx 組み込みサブルーチンは使用できません。
- PL/I 以外の言語で書かれたルーチンに EXEC CICS ステートメントが含まれている場合、そのルーチンを PL/I CICS プログラムから呼び出すことはできません。EXEC CICS ステートメントを含む非 PL/I プログラムとやり取りする場合は、EXEC CICS LINK または EXEC CICS XCTL を使用して行う必要があります。

CICS 環境で PUT FILE(SYSPRINT) を使用することは許可されていますが、パフォーマンスが低下するので通常は実動プログラムでは使用しないでください。

CICS EIB アドレスは、CICS 変換プログラムまたは、OPTIONS(MAIN) プログラムの PL/I CICS プリプロセッサのどちらかでしか生成されないので、OPTIONS(FETCHABLE) ルーチンの EIB へのアドレス可能性を確立する責任はユーザーにあります。

アドレス可能性を築くためには、このコマンドを使用するか、

```
EXEC CICS ADDRESS EIB(DFHEIPTR)
```

または、EIB アドレスを、外部プロシージャを呼び出す CALL ステートメントへの引数として引き渡します。

エラー処理

言語環境プログラムでは、PL/I ON ユニットまたは PL/I ON ユニットに呼び出されるコードで次の EXEC CICS コマンドを使用することが禁止されています。

- EXEC CICS ABEND
- EXEC CICS HANDLE AID
- EXEC CICS HANDLE ABEND
- EXEC CICS HANDLE CONDITION
- EXEC CICS IGNORE CONDITION
- EXEC CICS POP HANDLE
- EXEC CICS PUSH HANDLE

他のすべての EXEC CICS コマンドは ON ユニット内で許可されています。しかし、それらは NOHANDLE オプション、RESP オプション、または、RESP2 オプションを使用してコーディングされていなければなりません。

第 3 章 PL/I カタログ式プロシーチャーの用法

本章ではIBM Enterprise PL/I for z/OSコンパイラーで使用する IBM 提供の標準カタログ式プロシーチャーについて説明します。また、プロシーチャーの呼び出し方、一時的または永続的な変更方法についても説明します。この章で説明する任意のカタログ式プロシーチャーを使用するためには、言語環境プログラム SCEERUN データ・セットが STEPLIB に存在し、コンパイラーからアクセス可能になっていなければなりません。

カタログ式プロシーチャーは、ライブラリーに保管されたジョブ制御ステートメントのセットです。カタログ式プロシーチャーには、1 つ以上の EXEC ステートメントがあり、さらに 1 つ以上の DD ステートメントが続く場合があります。ステートメントを検索するには、入力ストリーム内の EXEC ステートメントの PROC パラメーターの中で、カタログ式プロシーチャーの名前を指定します。

カタログ式プロシーチャーを使うと、時間を節約し、ジョブ制御言語 (JCL) エラーを減らすことができます。カタログ式プロシーチャー内のステートメントがユーザー要件と正しく一致していなくても、ジョブが終わるまでの間、簡単にステートメントを変更したり、新たにステートメントを付け加えることができます。このプロシーチャーは見直しを行って変更を加え、使用できる各種機能を最も効率よく利用できるよう、また、ユーザー固有の規則を守れるようにしなければなりません。

IBM 提供のカタログ式プロシーチャー

Enterprise PL/I for z/OS で使用するために提供される PL/I カタログ式プロシーチャーは次のとおりです。

IBMZC

コンパイルのみ

IBMZCB

コンパイルおよびバインド

IBMZCPL

コンパイル、プリリンク、およびリンク・エディット

IBMZCBG

コンパイル、バインド、および実行

IBMZCPLG

コンパイル、プリリンク、リンク・エディット、および実行

IBMZCPG

コンパイル、プリリンク、ロード、および実行

カタログ式プロシーチャー IBMZCB および IBMZCBG では、言語環境プログラムで提供されるプリリンカーの代わりに、DFSMS/MVS® 1.4 で導入されたプログラム管理バインダーの機能を使用します。これらのプロシーチャーは、PDSE にプログラム・オブジェクトを作成します。

カタログ式プロシーチャー IBMZCPL、IBMZCPLG、および IBMZCPG では、言語環境プログラムで提供されるプリリンカーを使用して、PDS にロード・モジュール

を作成します。PDSE を使用したくない場合は、これらのプロシージャーを使用してください。このセクションでは、各種のカタログ式プロシージャーのプロシージャー・ステップを説明します。コンパイルおよびリンク・エディットのための個々のステートメントについては、160 ページの『JCL を使用した z/OS の下のコンパイラーの呼び出し』および「z/OS 言語環境プログラム プログラミング・ガイド」を参照してください。上記のカタログ式プロシージャーには、入力データ・セット用の DD ステートメントは入っていません。ユーザーが必ず提供しなければなりません。図 7 の例は、PL/I プログラムのコンパイル、バインド、および実行のためにカタログ式プロシージャー IBMZCBG を呼び出す時に使用する JCL ステートメントを示しています。

Enterprise PL/I は、最小 REGION サイズ 32M を必要とします。大きいプログラムにはより多くのストレージが必要です。実行しようとするカタログ式プロシージャーを呼び出す EXEC ステートメント上で REGION を指定しないと、コンパイラーはユーザーのサイト用にデフォルト REGION サイズを使用します。このデフォルト・サイズは、PL/I プログラムのサイズにより適切な場合と適切でない場合があります。

最適化をオンにしてプログラムをコンパイルする場合は、必要な REGION サイズ (および時間) がはるかに大きくなることがあります。

EXEC ステートメントでの REGION の指定方法の例は、図 7 を参照してください。

```
//COLEGO      JOB
//STEP1      EXEC IBMZCBG, REGION.PLI=32M
//PLI.SYSIN DD *
              .
              .
              .
              (insert PL/I program to be compiled here)
              .
              .
              .
/*
```

図 7. カタログ式プロシージャーの呼び出し

コンパイルのみ (IBMZC)

142 ページの図 8 に示すカタログ式プロシージャー IBMZC には、プロシージャー・ステップが 1 つだけあります。このステップでコンパイル用に指定されているオプションは OBJECT と OPTIONS です。(IBMZPLI は、コンパイラーのシンボル名です。) コンパイル・プロシージャー・ステップの入ったその他のカタログ式プロシージャーと同様に、IBMZC には入力データ・セット用の DD ステートメントは入っていません。ユーザーが修飾 dd 名 PLI.SYSIN を付けて適切なステートメントを必ず提供しなければなりません。

OBJECT コンパイル時オプションを指定すると、コンパイラーは、リンケージ・エディターへの入力に適した構文のオブジェクト・モジュールを、SYSLIN という名

前の DD ステートメントで定義された標準データ・セットに入れます。このステートメントは順次装置上で &&LOADSET という名前の一時データ・セットを定義します。ジョブ終了後にオブジェクト・モジュールを保持したい場合は、&&LOADSET を永続名 (すなわち、&& で始まらない名前) で置き換え、またそのデータ・セットを使用した最後のプロシージャ・ステップについては、適切な DISP パラメーターに KEEP を指定しなければなりません。そのためには、下記のように、ユーザー独自の SYSLIN DD ステートメントを提供します。このステートメント上のデータ・セット名と後処理パラメーターにより、IBMZC プロシージャの SYSLIN DD ステートメントの指定が変更されます。この例では、コンパイル・ステップが唯一のジョブ・ステップです。

```
//PLICOMP EXEC IBMZC
//PLI.SYSLIN DD DSN=MYPROG,DISP=(MOD,KEEP)
//PLI.SYSIN DD ...
```

142 ページの図 8 の DISP パラメーターにある MOD という項により、コンパイラーはデータ・セットに複数のオブジェクト・モジュールを入れることができます。また、PASS により、対応する DD ステートメントがある限り、以降のプロシージャ・ステップでもそのデータ・セットを使用できます。

SYSLIN SPACE パラメーターを指定すると、1 シリンダーの初期割り振りを行うことができ、必要であればさらに 15 回の割り振りを行うことができます (合計 16 シリンダー)。

```

//IBMZC  PROC LNGPRFX='IBMZ.V3R8M0',LIBPRFX='CEE',
//          SYSLBLK=3200
//*
//*****
//*
//* LICENSED MATERIALS - PROPERTY OF IBM
//*
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2008
//* ALL RIGHTS RESERVED.
//*
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
//* ADP SCHEDULE CONTRACT WITH IBM CORP.
//*
//*****
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 8 MODIFICATION 0
//*
//* COMPILE A PL/I PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX   IBMZ.V3R8M0    PREFIX FOR LANGUAGE DATA SET NAMES
//*  LIBPRFX   CEE            PREFIX FOR LIBRARY DATA SET NAMES
//*  SYSLBLK   3200           BLKSIZE FOR OBJECT DATA SET
//*
//* USER MUST SUPPLY //PLI.SYSIN DD STATEMENT THAT IDENTIFIES
//* LOCATION OF COMPILER INPUT
//*
//*****
//* COMPILE STEP
//*****
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB DD DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//          DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//          SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSALLDA,
//          SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024

```

図8. カタログ式プロシージャー IBMZC

コンパイルおよびバインド (IBMZCB)

143 ページの図9 の IBMZCB カタログ式プロシージャーには、2 つのプロシージャー・ステップがあります。1 つは PLI で、これはカタログ式プロシージャー IBMZC と同じです。もう 1 つは BIND で、これは最初のプロシージャー・ステップで作成されたオブジェクト・モジュールをバインドするためにプログラム管理バインダー (シンボル名 IEWBLINK) を呼び出します。

コンパイル・プロシージャー・ステップ用の入力データには、修飾 dd 名の PLI.SYSIN が必要です。EXEC ステートメント BIND の COND パラメーターは、コンパイラが生成した戻りコードが 8 より大きい場合 (つまり、コンパイル中に重大エラーまたは回復不能エラーが起こった場合) に、このプロシージャー・ステ

ップをバイパスするよう指定します。

```
//IBMZCB  PROC LNGPRFX='IBMZ.V3R8M0',LIBPRFX='CEE',
//          SYSLBLK=3200,GOPGM=GO
//*
//*****
//*
//* LICENSED MATERIALS - PROPERTY OF IBM
//*
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2008
//* ALL RIGHTS RESERVED.
//*
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
//* ADP SCHEDULE CONTRACT WITH IBM CORP.
//*
//*****
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 8 MODIFICATION 0
//*
//* COMPILE AND BIND A PL/I PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX  IBMZ.V3R8M0     PREFIX FOR LANGUAGE DATA SET NAMES
//*  LIBPRFX  CEE             PREFIX FOR LIBRARY DATA SET NAMES
//*  SYSLBLK  3200            BLKSIZE FOR OBJECT DATA SET
//*  GOPGM    GO              MEMBER NAME FOR PROGRAM OBJECT
//*
//*****
//* COMPILE STEP
//*****
//PLI        EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB    DD DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//           DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT   DD SYSOUT=*
//SYSOUT     DD SYSOUT=*
//SYSLIN     DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//           SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1     DD DSN=&&SYSUT1,UNIT=SYSALLDA,
//           SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*****
//* BIND STEP
//*****
//BIND       EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//           PARM='XREF,COMPAT=PM3'
//SYSLIB     DD DSN=&LIBPRFX..SCEELKD,DISP=SHR
//SYSPRINT   DD SYSOUT=*
//SYSLIN     DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//           DD DDNAME=SYSIN
//SYSLMOD    DD DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//           SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD   DD DUMMY
//SYSIN      DD DUMMY
```

図9. カタログ式プロシージャー IBMZCB

プログラム管理バインダーは、常に、SYSLMOD という名前の DD ステートメントで定義された標準データ・セットに、作成したプログラム・オブジェクトを入れます。カタログ式プロシージャーの中のこのステートメントは、新しい一時ライブラリー &&GOSET を指定します。プログラム・オブジェクトはこの一時ライブラリーに入れられ、GO というメンバー名を与えられます。一時ライブラリーを指定する

際は、カタログ式プロシージャはプログラム・オブジェクトが同じジョブ内で実行されることを想定します。プログラム・オブジェクトを保持したい場合は、`SYSLMOD` という名前の `DD` ステートメントをユーザー独自の名前に置き換えなければなりません。

コンパイル、バインド、および実行 (IBMZCBG)

145 ページの図 10 の IBMZCBG カatalog式プロシージャには、`PLI`、`BIND`、`GO` の 3 つのプロシージャ・ステップがあります。`PLI` と `BIND` は IBMZCB の 2 つのプロシージャ・ステップと同じです。`GO` は `BIND` ステップで作成されたプログラム・オブジェクトを実行します。`GO` ステップは、前のプロシージャ・ステップで重大エラーまたは回復不能エラーが起こらなかった場合だけ実行されます。

コンパイル・プロシージャ・ステップ用の入力データは、`PLI.SYSIN` という名前の `DD` ステートメントで指定しなければならず、また、`GO` ステップ用のものは、`GO.SYSIN` という名前の `DD` ステートメントで指定しなければなりません。

```

//IBMZCBG  PROC LNGPRFX='IBMZ.V3R8M0',LIBPRFX='CEE',
//          SYSLBLK=3200,GOPGM=GO
//*
//*****
//*
//* LICENSED MATERIALS - PROPERTY OF IBM
//*
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2008
//* ALL RIGHTS RESERVED.
//*
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
//* ADP SCHEDULE CONTRACT WITH IBM CORP.
//*
//*****
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 8 MODIFICATION 0
//*
//* COMPILE, BIND, AND RUN A PL/I PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX   IBMZ.V3R8M0    PREFIX FOR LANGUAGE DATA SET NAMES
//*  LIBPRFX   CEE            PREFIX FOR LIBRARY DATA SET NAMES
//*  SYSLBLK   3200           BLKSIZE FOR OBJECT DATA SET
//*  GOPGM     GO             MEMBER NAME FOR PROGRAM OBJECT
//*
//*****
//* COMPILE STEP
//*****
//PLI        EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB    DD  DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//           DD  DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT   DD  SYSOUT=*
//SYSOUT     DD  SYSOUT=*
//SYSLIN     DD  DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//           SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1     DD  DSN=&&SYSUT1,UNIT=SYSALLDA,
//           SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024

```

図 10. カタログ式プロシージャー IBMZCBG (1/2)

```

//*****
//* BIND STEP
//*****
//BIND      EXEC PGM=IEWBLINK,COND=(8,LT,PLI),
//          PARM='XREF,COMPAT=PM3'
//SYSLIB    DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//SYSMOD    DD DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//          SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY
//SYSDEFSD  DD DUMMY
//SYSIN     DD DUMMY
//*****
//* RUN STEP
//*****
//GO        EXEC PGM=*.BIND.SYSLMOD,COND=((8,LT,PLI),(8,LE,BIND))
//STEPLIB   DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//CEEDUMP   DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*

```

図 10. カタログ式プロシージャー IBMZCBG (2/2)

コンパイル、プリリンク、およびリンク・エディット (IBMZCPL)

147 ページの図 11 の IBMZCPL カタログ式プロシージャーには、3 つのプロシージャー・ステップがあります。すなわち、カタログ式プロシージャー IBMZC と同じものである PLI、言語環境プログラムのプリリンカーを呼び出す PLKED、および最初のプロシージャー・ステップで作成されたオブジェクト・モジュールをリンク・エディットするリンケージ・エディター (シンボル名 IEWL) を呼び出す LKED です。

コンパイル・プロシージャー・ステップ用の入力データには、修飾 dd 名の PLI.SYSIN が必要です。EXEC ステートメント LKED の COND パラメーターは、コンパイラが生成した戻りコードが 8 より大きい場合 (つまり、コンパイル中に重大エラーまたは回復不能エラーが起こった場合) に、このプロシージャー・ステップをバイパスするように指定します。

```

//IBMZCPL PROC LNGPRFX='IBMZ.V3R8M0',LIBPRFX='CEE',
//          SYSLBLK=3200,PLANG=EDCPMSGE,GOPGM=GO
//*
//*****
//*
//* LICENSED MATERIALS - PROPERTY OF IBM
//*
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2008
//* ALL RIGHTS RESERVED.
//*
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
//* ADP SCHEDULE CONTRACT WITH IBM CORP.
//*
//*****
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 8 MODIFICATION 0
//*
//* COMPILE, PRELINK, LINK-EDIT A PL/I PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX  IBMZ.V3R8M0     PREFIX FOR LANGUAGE DATA SET NAMES
//*  LIBPRFX  CEE             PREFIX FOR LIBRARY DATA SET NAMES
//*  SYSLBLK  3200            BLKSIZE FOR OBJECT DATA SET
//*  PLANG    EDCPMSGE        PRELINKER MESSAGES MEMBER NAME
//*  GOPGM    GO              MEMBER NAME FOR LOAD MODULE
//*
//*****
//* COMPILE STEP
//*****
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB DD DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//        DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//        SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSALLDA,
//        SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024

```

図 11. カタログ式プロシージャー IBMZCPL (1/2)

```

//*****
//* PRE-LINK-EDIT STEP
//*****
//PLKED EXEC PGM=EDCPRLK,COND=(8,LT,PLI)
//STEPLIB DD DSN=&LIBPRFX;..SCEERUN,DISP=SHR
//SYMSGS DD DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB DD DUMMY
//SYSMOD DD DSN=&PLNK,DISP=(,PASS),
// UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//*****
//* LINK-EDIT STEP
//*****
//LKED EXEC PGM=IEWL,PARM='XREF',COND=((8,LT,PLI),(8,LE,PLKED))
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLMOD DD DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
// SPACE=(1024,(50,20,1))
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSALLDA,SPACE=(1024,(200,20)),
// DCB=BLKSIZE=1024
//SYSIN DD DUMMY

```

図 11. カタログ式プロシージャー IBMZCPL (2/2)

リンケージ・エディターは、作成したロード・モジュールを常に、SYSLMOD という名前の DD ステートメントで定義された標準データ・セット内に入れます。カタログ式プロシージャーの中のこのステートメントは、新しい一時ライブラリー &&GOSET を指定します。ロード・モジュールはこの一時ライブラリーに入れられ、GO というメンバー名を与えられます。一時ライブラリーを指定する際は、カタログ式プロシージャーは同じジョブ内でロード・モジュールが実行されるものと想定します。このモジュールを保持したい場合は、SYSLMOD という名前の DD ステートメントをユーザー独自のステートメントで置き換えなければなりません。

147 ページの図 11 の SYSLIN DD ステートメントは、SYSIN という名前の DD ステートメントで定義されたデータ・セットを、リンケージ・エディターへの 1 次入力 (SYSLIN) に連結する方法を示しています。このような方法でリンケージ・エディター制御ステートメントを入力ストリームに入れることができます。これについては、「z/OS 言語環境プログラム プログラミング・ガイド」に説明があります。

コンパイル、プリリンク、リンク・エディット、および実行 (IBMZCPLG)

149 ページの図 12 の IBMZCPLG カタログ式プロシージャーには、PLI、PLKED、LKED、および GO の 4 つのプロシージャー・ステップがあります。PLI、PLKED、および LKED は IBMZCPL の 3 つのプロシージャー・ステップと同じです。GO は LKED ステップで作成されたロード・モジュールを実行します。GO ステップは、前のプロシージャー・ステップで重大エラーまたは回復不能エラーが起こらなかった場合だけ実行されます。

コンパイル・プロシージャー・ステップ用の入力データは、PLI.SYSIN という名前の DD ステートメントで指定しなければならず、また、GO ステップ用のものは、GO.SYSIN という名前の DD ステートメントで指定しなければなりません。

```
//IBMZCPLG PROC LNGPRFX='IBMZ.V3R8M0',LIBPRFX='CEE',
//          SYSLBLK=3200,PLANG=EDCPMSG, GOPGM=GO
//*
//*****
//*
//* LICENSED MATERIALS - PROPERTY OF IBM
//*
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2008
//* ALL RIGHTS RESERVED.
//*
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
//* ADP SCHEDULE CONTRACT WITH IBM CORP.
//*
//*****
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 8 MODIFICATION 0
//*
//* COMPILE, PRELINK, LINK-EDIT AND RUN A PL/I PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX  IBMZ.V3R8M0     PREFIX FOR LANGUAGE DATA SET NAMES
//*  LIBPRFX  CEE             PREFIX FOR LIBRARY DATA SET NAMES
//*  SYSLBLK  3200            BLKSIZE FOR OBJECT DATA SET
//*  PLANG    EDCPMSG         PRELINKER MESSAGES MEMBER NAME
//*  GOPGM    GO              MEMBER NAME FOR LOAD MODULE
//*
//*****
//* COMPILE STEP
//*****
//PLI      EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB DD DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//        DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//            SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSALLDA,
//            SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*****
//* PRE-LINK-EDIT STEP
//*****
//PLKED    EXEC PGM=EDCPRLK,COND=(8,LT,PLI)
//STEPLIB DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSMSG   DD DSN=&LIBPRFX..SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB   DD DUMMY
//SYSMOD   DD DSN=&&PLNK,DISP=(,PASS),UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN    DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
```

図 12. カタログ式プロシージャー IBMZCPLG (1/2)

```

//*****
//* LINK-EDIT STEP
//*****
//LKED      EXEC PGM=IEWL,PARM='XREF',COND=((8,LT,PLI),(8,LE,PLKED))
//SYSLIB    DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSLIN    DD DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//SYSLMOD   DD DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSALLDA,
//          SPACE=(1024,(50,20,1))
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSALLDA,SPACE=(1024,(200,20)),
//          DCB=BLKSIZE=1024
//SYSIN     DD DUMMY
//*****
//* RUN STEP
//*****
//GO        EXEC PGM=*.LKED.SYSLMOD,
//          COND=((8,LT,PLI),(8,LE,PLKED),(8,LE,LKED))
//STEPLIB   DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//CEEDUMP   DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*

```

図 12. カタログ式プロシージャー IBMZCPLG (2/2)

コンパイル、プリリンク、ロード、および実行 (IBMZCPG)

151 ページの図 13 の IBMZCPG カタログ式プロシージャーでは、IBMZCPLG と同じ結果が得られますが、リンケージ・エディターではなくローダーが使用されます。また、プロシージャ・ステップの数は 4 つ (コンパイル、プリリンク、リンク・エディット、実行) ではなく、3 つ (コンパイル、プリリンク、ロード/実行) しかありません。3 番目のプロシージャ・ステップはローダー・プログラムを実行します。ローダー・プログラムは、コンパイラが作成したオブジェクト・モジュールを処理し、結果の実行可能プログラムをすぐに実行します。コンパイル・ステップ用の入力データは、修飾 dd 名 PLI.SYSIN を提供してユーザーが提供する必要があります。

ローダーを使用すると、PL/I プログラムに対していくつかの制約が課せられます。このカタログ式プロシージャーを使用する前に、ローダーの使用方法を説明した「z/OS 言語環境プログラム プログラミング・ガイド」を参照してください。

```

//IBMZCPG PROC LNGPRFX='IBMZ.V3R8M0',LIBPRFX='CEE',
//          SYSLBLK=3200,PLANG=EDCPMSGE
//*
//*****
//*
//* LICENSED MATERIALS - PROPERTY OF IBM
//*
//* 5655-H31 (C) COPYRIGHT IBM CORP. 1999, 2008
//* ALL RIGHTS RESERVED.
//*
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
//* ADP SCHEDULE CONTRACT WITH IBM CORP.
//*
//*****
//*
//* IBM ENTERPRISE PL/I FOR Z/OS
//* VERSION 3 RELEASE 8 MODIFICATION 0
//*
//* COMPILE, PRELINK, LOAD AND RUN A PL/I PROGRAM
//*
//* PARAMETER  DEFAULT VALUE  USAGE
//*  LNGPRFX  IBMZ.V3R8M0     PREFIX FOR LANGUAGE DATA SET NAMES
//*  LIBPRFX  CEE             PREFIX FOR LIBRARY DATA SET NAMES
//*  SYSLBLK  3200            BLKSIZE FOR OBJECT DATA SET
//*  PLANG    EDCPMSGE        PRELINKER MESSAGES MEMBER NAME
//*
//*****
//* COMPILE STEP
//*****
//PLI        EXEC PGM=IBMZPLI,PARM='OBJECT,OPTIONS'
//STEPLIB    DD DSN=&LNGPRFX;.SIBMZCMP,DISP=SHR
//           DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSPRINT   DD SYSOUT=*
//SYSOUT     DD SYSOUT=*
//SYSLIN     DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSALLDA,
//           SPACE=(CYL,(1,1)),DCB=(LRECL=80,BLKSIZE=&SYSLBLK)
//SYSUT1     DD DSN=&&SYSUT1,UNIT=SYSALLDA,
//           SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024
//*****
//* PRE-LINK-EDIT STEP
//*****
//PLKED      EXEC PGM=EDCPRLK,COND=(8,LT,PLI)
//STEPLIB    DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYMSGS     DD DSN=&LIBPRFX;.SCEEMSGP(&PLANG),DISP=SHR
//SYSLIB     DD DUMMY
//SYSMOD     DD DSN=&&PLNK,DISP=(,PASS),
//           UNIT=SYSALLDA,SPACE=(CYL,(1,1)),
//           DCB=(RECFM=FB,LRECL=80,BLKSIZE=&SYSLBLK)
//SYSIN      DD DSN=*.PLI.SYSLIN,DISP=(OLD,DELETE)
//SYSPRINT   DD SYSOUT=*
//SYSOUT     DD SYSOUT=*

```

図 13. カタログ式プロシージャー IBMZCPG (1/2)

```

//*****
//* LOAD AND RUN STEP
//*****
//GO      EXEC PGM=LOADER,PARM='MAP,PRINT',
//          COND=((8,LT,PLI),(8,LE,PLKED))
//STEPLIB DD DSN=&LIBPRFX;.SCEERUN,DISP=SHR
//SYSLIB  DD DSN=&LIBPRFX..SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN  DD DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

図 13. カタログ式プロシージャー IBMZCPG (2/2)

他のカタログ式プロシージャーの詳細については、「*z/OS 言語環境プログラム プログラミング・ガイド*」を参照してください。

カタログ式プロシージャーの呼び出し

カタログ式プロシージャーを呼び出すには、EXEC ステートメントの PROC パラメーターにプロシージャ名を指定します。例えば、カタログ式プロシージャー IBMZC を使用するには、入力ストリーム内で他のジョブ制御ステートメント中の適切な場所に次のステートメントを入れます。

```
//stepname EXEC PROC=IBMZC
```

キーワード PROC のコーディングは必要ありません。EXEC ステートメントの第 1 オペランドが PGM= または PROC= で始まっていない場合は、ジョブ・スケジューラーは第 1 オペランドをカタログ式プロシージャーの名前であるものと解釈します。次のステートメントは上記のステートメントと同等です。

```
//stepname EXEC IBMZC
```

JOB ステートメントにパラメーター MSGLEVEL=1 を入れると、オペレーティング・システムはオリジナルの EXEC ステートメントをそのリストに組み込み、カタログ式プロシージャーのステートメントを追加します。このリストでは、カタログ式プロシージャーのステートメントは最初の 2 文字の XX あるいは X/ によって識別されます。X/ は、カタログ式プロシージャーの現在の呼び出しで変更されたステートメントを意味します。

カタログ式プロシージャーを呼び出したジョブ・ステップが終わるまでの間、カタログ式プロシージャーのステートメントを変更しておかなければならないことがあります。変更するには、DD ステートメントを付け加えるか、または EXEC ステートメントか DD ステートメント内の 1 つ以上のパラメーターを指定変更します。例えば、コンパイラーを呼び出すカタログ式プロシージャーでは、ソース・ステートメントが入っているデータ・セットを定義するために、SYSIN という名前の DD ステートメントを追加する必要があります。また、1 つのジョブで複数の標準リンク・エディットのプロシージャー・ステップを使用する場合、複数のロード・モジュールを実行したければ、呼び出すカタログ式プロシージャーのうち、最初のものを除くすべてのものを変更しなければなりません。

複数呼び出しの指定

同じジョブ内で、複数のカタログ式プロシージャを呼び出すことができます。また、同じジョブ内で同じカタログ式プロシージャを複数回呼び出すこともできます。これらのカタログ式プロシージャの 2 つ以上に同一のリンク・エディット・プロシージャ・ステップが関与しない限り、特別な問題が起こることはないと思われます。複数のカタログ式プロシージャが同一のリンク・エディット・プロシージャ・ステップに関与する場合には、すべてのロード・モジュールを実行できるようにするために、以下に示す予防策をとる必要があります。

リンケージ・エディターがロード・モジュールを作成する場合、リンケージ・エディターはロード・モジュールを `SYSLMOD` という名前の `DD` ステートメントで定義された標準データ・セットに入れます。バインダーがプログラム・オブジェクトを作成する場合、バインダーは、作成したプログラム・オブジェクトを、`SYSLMOD` という名前の `DD` ステートメントで定義された `PDSE` に入れます。リンケージ・エディターの `NAME` ステートメントがない場合、リンケージ・エディターまたはバインダーは `DSNAME` パラメーターで指定されたメンバー名をモジュール名として使用します。標準のカタログ式プロシージャでは、`SYSLMOD` という名前の `DD` ステートメントは常に、メンバー名 `GO` の付いた一時ライブラリー `&&GOSET` を指定します。

2 つの `PL/I` プログラムをコンパイル、バインド、および実行するために、同じジョブの中でカタログ式プロシージャ `IBMZCBG` を 2 回使用し、バインダーが作成する 2 つのプログラム・オブジェクトのそれぞれに名前を付けなかった場合は、最初のプログラム・オブジェクトが 2 回実行され、2 番目のプログラム・オブジェクトはまったく実行されません。

これを防ぐには、次に挙げる方法のうちどれか 1 つを使用します。

- `GO` ステップの終わりにあるライブラリー `&&GOSET` を削除します。`GO` ステップの終わりでの、カタログ式プロシージャの最初の呼び出しで、次の構文の `DD` ステートメントを追加します。

```
//GO.SYSLMOD DD DSN=&&GOSET,  
// DISP=(OLD,DELETE)
```

- カatalog式プロシージャの 2 番目以降の呼び出しで、`SYSLMOD` という名前の `DD` ステートメントを変更し、ロード・モジュールの名前が違うものになるようにします。次に例を示します。

```
//BIND.SYSLMOD DD DSN=&&GOSET(G01)
```

などとします。

- `NAME` リンケージ・エディター・オプションを使ってプログラム・オブジェクトごとに異なった名前を付け、各ジョブ・ステップの `EXEC` ステートメントを変更して、該当するジョブ・ステップ用の名前の付いたプログラム・オブジェクトの実行を指定します。

プログラム・オブジェクトにメンバー名を割り当てるには、`SYSLMOD DD` ステートメントに `DSNAME` パラメーターを指定して、リンケージ・エディターの `NAME` オプションを使用することができます。このプロシージャを使用するとき、プログラムを実行する `EXEC` ステートメントが、実行されるモジュール名を

SYSLMOD DD ステートメントで参照する場合は、メンバー名は NAME オプション上の名前と同じでなければなりません。

また、EXEC プロシージャ・ステートメントで GOPGM を使用して、プログラムごとに異なる名前を付けるという別の方法もあります。次に例を示します。

```
// EXEC IBMZCBG,GOPGM=G02
```

PL/I カタログ式プロシージャの変更

カタログ式プロシージャを呼び出す EXEC ステートメントにパラメーターを組み込んで、あるいは EXEC ステートメントの後に DD ステートメントを追加して、カタログ式プロシージャを一時的に変更することができます。一時変更は、プロシージャが呼び出されるジョブ・ステップの期間だけ適用されます。一時変更によって、プロシージャ・ライブラリー内のカタログ式プロシージャのマスタ・コピーが影響を受けることはありません。

一時変更は、カタログ式プロシージャの EXEC ステートメントまたは DD ステートメントに適用できます。EXEC ステートメントのパラメーターを変更するには、それに対応するパラメーターを、カタログ式プロシージャを呼び出す EXEC ステートメントに入れなければなりません。DD ステートメントの 1 つ以上のパラメーターを変更するには、カタログ式プロシージャを呼び出す EXEC ステートメントの後に、対応する DD ステートメントを組みこまなければなりません。カタログ式プロシージャに新しい EXEC ステートメントを追加することはできませんが、追加の DD ステートメントはいつでも組み込むことができます。

EXEC ステートメント

カタログ式プロシージャを呼び出す EXEC ステートメントのパラメーターに非修飾名がある場合、そのパラメーターはカタログ式プロシージャ内のすべての EXEC ステートメントに適用されます。カタログ式プロシージャに対する影響は、次のようにパラメーターによって異なります。

- PARM は最初のプロシージャ・ステップに適用され、他のすべての PARM パラメーターを無効にします。
- COND と ACCT はすべてのプロシージャ・ステップに適用されます。
- TIME と REGION はすべてのプロシージャ・ステップに適用され、既存の値を指定変更します。

次のステートメント

```
//stepname EXEC IBMZCBG,PARM='OFFSET',REGION=32M
```

- カタログ式プロシージャ IBMZCBG を呼び出します。
- プロシージャ・ステップ PLI の EXEC ステートメントの中の OBJECT および OPTIONS を OFFSET で置き換えます。
- プロシージャ・ステップ BIND の EXEC ステートメントの中の PARM パラメーターを無効にします。
- 3 つのプロシージャ・ステップすべてに、領域サイズ 32M を指定します。

カタログ式プロシージャの 1 つの EXEC ステートメントだけのパラメーターの値を変更したり、1 つの EXEC ステートメントに新しいパラメーターを追加したり

するには、パラメーターの名前をプロシージャ・ステップの名前で修飾して、その EXEC ステートメントを識別する必要があります。例えば、前の例でプロシージャ・ステップ PLI の領域サイズだけを変更するには、次のようにコーディングします。

```
//stepname EXEC PROC=IBMZCBG,PARM='OFFSET',REGION.PLI=90M
```

呼び出しを行う EXEC ステートメントに指定された新しいパラメーターは、プロシージャ EXEC ステートメントの対応パラメーターを指定変更します。

値を指定せずにキーワードと等号をコーディングすると、パラメーターに指定されたオプションをすべて無効にすることができます。例えば、カタログ式プロシージャ IBMZCBG を呼び出すときにリンケージ・エディター・リストを一括して抑止するには、次のようにコーディングします。

```
//stepname EXEC IBMZCBG,PARM.BIND=
```

DD ステートメント

カタログ式プロシージャに DD ステートメントを追加したり、既存 DD ステートメントの 1 つ以上のパラメーターを変更するには、入力ストリーム内の適切な場所に procstepname.ddname の形の DD ステートメントを入れなければなりません。ddname が procstepname として識別されているプロシージャ・ステップに既に存在する DD ステートメントの名前である場合に、新しい DD ステートメント内のパラメーターは既存の DD ステートメントの対応パラメーターをすべて指定変更します。そうでない場合は、新しい DD ステートメントがプロシージャ・ステップに追加されます。次のステートメント

```
//PLI.SYSIN DD *
```

は、カタログ式プロシージャ IBMZC のプロシージャ・ステップ PLI に DD ステートメントを追加しますが、次のステートメント

```
//PLI.SYSPRINT DD SYSOUT=C
```

は既存の DD ステートメント SYSPRINT を変更します (その結果、コンパイラー・リストはクラス C のシステム出力装置に送られます)。

指定変更を行う DD ステートメントは、プロシージャ呼び出しの後で、カタログ式プロシージャに出現するとおりの順序で出現しなければなりません。該当ステップに指定変更を行う DD ステートメントを指定した後に、DD ステートメントを追加することができます。

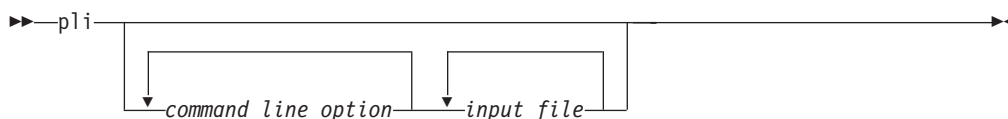
DD ステートメントのパラメーターを指定変更するには、そのパラメーターを改訂した形でコーディングするか、同様の機能を実行する置換パラメーターをコーディングします (例えば、SPLIT を SPACE に置き換えます)。パラメーターを無効にするには、値を指定せずにキーワードと等号をコーディングします。DCB サブパラメーターは、変更したいものだけをコーディングして指定変更できます。つまり、指定変更を行う DD ステートメントの DCB パラメーターは、カタログ式プロシージャ内の対応ステートメントの DCB パラメーター全体を指定変更するとは限りません。

第 4 章 プログラムのコンパイル

この章では、z/OS UNIX System Services (z/OS UNIX) の下でコンパイラーを呼び出す方法と、z/OS の下でコンパイルするのに使用するジョブ制御ステートメントについて説明します。プログラムをコンパイルするためには、言語環境プログラム SCEERUN データ・セットがコンパイラーからアクセス可能になっている必要があります。

z/OS UNIX の下でのコンパイラーの呼び出し

z/OS UNIX 環境下でプログラムをコンパイルするには、**pli** コマンドを使用します。



command_line_option

次のようにして `command_line_option` を指定できます。

- **-qoption**
- オプション・フラグ (通常は前に - を付けた単一文字)

コマンド行でコンパイル時オプションを指定する場合は、`%PROCESS` ステートメントを使用してソース・ファイルに設定する場合と異なるフォーマットを使うことになります。158 ページの『z/OS UNIX の下でのコンパイル時オプションの指定』を参照。

input_file

プログラム・ファイル用の z/OS UNIX ファイル指定。ファイル指定から拡張子を省略すると、コンパイラーは拡張子 `.pli` を想定します。絶対パスを省略すると、現行ディレクトリーが想定されます。

入力ファイル

pli コマンドは、PL/I ソース・ファイルをコンパイルし、結果のオブジェクト・ファイルを、コマンド行で指定した任意のオブジェクト・ファイルおよびライブラリーに指定順にリンクし、単一の実行可能ファイルを作成します。

pli コマンドが受け入れるファイルのタイプは次のとおりです。

ソース・ファイルー.pli

`.pli` ファイルはすべてコンパイル用のソース・ファイルです。**pli** コマンドは、リストされた順にソース・ファイルをコンパイラーに送ります。指定されたソース・ファイルが見つからない場合、コンパイラーはエラー・メッセージを生成し、**pli** コマンドは、次のファイルがあればそれ进行处理します。

すべての HFS ソース・ファイルは、行区切りで、かつ EBCDIC でエンコードされている必要があります。

オブジェクト・ファイルー.o

.o ファイルはすべてオブジェクト・ファイルです。**-c** オプションを指定しない限り、**pli** コマンドはリンク・エディット時にすべてのオブジェクト・ファイルおよびライブラリー・ファイルをリンケージ・エディターに送ります。すべてのソース・ファイルをコンパイルした後、コンパイラーはリンケージ・エディターを呼び出して、結果のオブジェクト・ファイルを、入力ファイル・リストに指定されているすべてのオブジェクト・ファイルにリンク・エディットし、単一の実行可能出力ファイルを作成します。

ライブラリー・ファイルー.a

pli コマンドは、リンク・エディット時にすべてのライブラリー・ファイル (.a ファイル) をリンケージ・エディターに送ります。

z/OS UNIX の下でのコンパイル時オプションの指定

Enterprise PL/I は、コンパイラーのデフォルト設定を変更するためのコンパイル時オプションを備えています。コマンド行でオプションを指定することができます。指定したオプションは、ソース・プログラムの中の **%PROCESS** ステートメントにより指定変更されない限り、ファイルのすべてのコンパイル単位に有効です。

これらのオプションについては、5 ページの『コンパイル時オプションの説明』を参照してください。

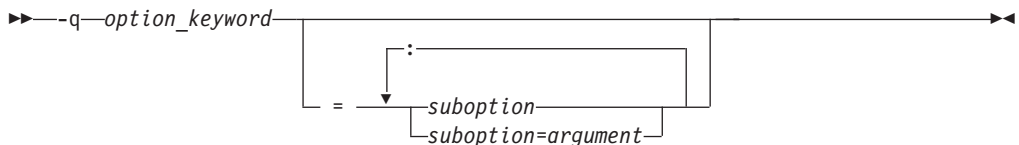
コマンド行で指定したオプションは、オプションのデフォルト設定を指定変更します。指定したオプションは、ソース・ファイルで設定されているオプションにより指定変更されます。

コマンド行でコンパイル時オプションを指定するには、次の 3 つの方法があります。

- **-qoption_keyword** (コンパイラー特有)
- 単一フラグおよび複数文字フラグ
- **-q+/u/myopts.txt**

-qoption_keyword

-qoption のフォーマットを使用して、コマンド行でオプションを指定できます。



同じコマンド行で複数の **-qoption** を指定できます。ただし、オプションをブランクで区切る必要があります。オプション・キーワードは大文字または小文字のどちらでも指定できます。ただし、**-q** は小文字で指定しなければなりません。

一部のコンパイル時オプションではサブオプションを指定できます。これらのサブオプションは、コマンド行では等号の後に **-qoption_keyword** を付けて示されます。複数のサブオプションは、ブランクを入れずにコロン (:) で区切らなければなりません。

複数のサブオプションを持つオプションの例として RULES があります (66 ページの『RULES』)。コマンド行で RULES(LAXDCL) を指定するには、次のように入力します。

```
-qrules=ibm:laxdcl
```

LIMITS オプション (44 ページの『LIMITS』) の場合は、そのサブオプションがそれぞれ引数をとるため、もう少し複雑になります。コマンド行では、LIMITS(EXTNAME(31),FIXEDDEC(15)) を次の例のように指定します。

```
-qlimits=extname=31:fixeddec=15
```

単一フラグおよび複数文字フラグ

z/OS UNIX ファミリーのコンパイラーでは、共通の標準フラグをいくつも使用します。各言語には、独自の追加フラグのセットがあります。

いくつかのフラグ・オプションは、フラグの一部となる引数をとります。例えば、

```
pli samp.pli -I/home/test3/include
```

この場合、/home/test3/include は、INCLUDE ファイルが検索される組み込みディレクトリーです。

それぞれのフラグ・オプションは、別々の引数として指定してください。

表 11. z/OS UNIX の下で Enterprise PL/I によりサポートされるコンパイル時オプション・フラグ

オプション	説明
-c	コンパイルのみ。
-e	フェッチ可能なロード・モジュールに名前とエントリーを作成します。
-I<dir>*	INCLUDE ファイルを検索するディレクトリーにパス <dir> を追加します。-I の後にはパス名が必要です。1 つの -I オプションには 1 つのパスしか指定できません。複数のパスを追加するには、複数の -I オプションを使用します。-I とパス名の間にスペースは入れないでください。
-0, -02	生成されるコードを最適化します。このオプションは -qOPT=2 と同等です。
-q<option>*	コンパイラーに渡します。<option> はコンパイル時オプションです。各オプションはコンマで区切り、各サブオプションは等号またはコロンで区切ってください。-q と <option> の間にスペースは入れないでください。
-v	コンパイルおよびリンクのステップを表示し、それらのステップを実行します。
-#	コンパイルおよびリンクのステップを表示しますが、それらのステップを実行しません。

注: * 指示のある個所に引数を指定しなければなりません。そうしないと、予測できない結果となります。

JCL を使用した z/OS の下でのコンパイラーの呼び出し

ユーザーはおそらく、コンパイラーを呼び出すジョブ・ステップに必要なすべての JCL ステートメントを自分で提供するのではなく、カタログ式プロシージャを使用することになりますが、コンパイラーを最も効果的に使用し、必要であればカタログ式プロシージャのステートメントを指定変更することもできるようにするために、これらのステートメントに慣れておくべきです。

複数のオブジェクト・デックを一度のコンパイルで生成する、いわゆる「バッチ・コンパイル」はサポートされません。

また、BPXBATCH によるコンパイラーの呼び出しもサポートされていません。

以下のセクションでは、コンパイルに必要な JCL について説明します。IBM 提供のカタログ式プロシージャ（139 ページの『IBM 提供のカタログ式プロシージャ』）には、これらのステートメントが入っています。カタログ式プロシージャを使用しない場合だけ、これらのステートメントを自分でコーディングする必要があります。

EXEC ステートメント

基本 EXEC ステートメントは次のとおりです。

```
//stepname EXEC PGM
```

このステートメントの REGION パラメーターには、512K が必要です。

最適化をオンにしてプログラムをコンパイルする場合は、必要な REGION サイズ（および時間）がはるかに大きくなることがあります。

EXEC ステートメントの PARM パラメーターを使用すれば、コンパイラーに備わったオプション機能のうちの 1 つ以上のものを指定することができます。これらの機能は 164 ページの『EXEC ステートメントでのオプションの指定』で説明されています。各種オプションの説明については、5 ページの『第 1 章 コンパイラー・オプションと機能の使用』を参照してください。

標準データ・セット用の DD ステートメント

コンパイラーにはいくつかの標準データ・セットが必要です。データ・セットの数は指定したオプション機能により異なります。これらのデータ・セットは、標準 DD 名を持つ DD ステートメントに定義する必要があります。標準 DD 名は、他のデータ・セット特性と共に、161 ページの表 12 に示されています。DD ステートメント SYSIN、SYSUT1、および SYSPRINT は常に必要です。

標準データ・セットはどれも直接アクセス装置に保管できますが、DD ステートメントに SPACE パラメーターを組み込む必要があります。このパラメーターは、必要な補助記憶域の量を指定するためにデータ・セットを定義します。IBM 提供のカタログ式プロシージャに割り振られる補助記憶域の量は、大部分のアプリケーションにとって十分と思われます。

表 12. コンパイラーの標準データ・セット

標準 DDNAME	データ・セットの内容	可能な 装置 クラス ¹	レコード のフォー マット (RECFM)	レコード のサイズ (LRECL)
SYSDEBUG	TEST(SEPARATE) 出力	SYSDA	F,FB	>=80 および <=1024
SYSDEFSD	XINFO(DEF) 出力	SYSDA	F,FB	128
SYSIN	コンパイラーへの入力	SYSSQ	F,FB,U VB,V	<101(100) <105(104)
SYSLIB	INCLUDE ファイル用ソー ス・ステートメント	SYSDA	F,FB,U V,VB	<101 <105
SYSLIN	オブジェクト・モジュール	SYSSQ	FB	80
SYSPRINT	リスト (メッセージを含む)	SYSSQ	VBA	137
SYSPUNCH	ブリプロセッサ出力、コ ンパイラー出力	SYSSQ SYSCP	FB	80 または MARGINS() 値
SYSUT1	一時作業ファイル	SYSDA	F	4051
SYSXMI	XINFO(XMI) 出力	SYSDA	VB	16383
SYSXMLSD	XINFO(XML) 出力	SYSDA	VB	16383
SYSADATA	XINFO(MSG) 出力	SYSDA	U	1024

注:

指定変更できるコンパイル時 SYSPRINT の値は BLKSIZE だけです。

1. 可能な装置クラスは次のとおりです。

SYSSQ 順次装置

SYSDA 直接アクセス装置

SYSUT1 以外はブロック・サイズを指定できます。SYSUT1 のブロック・サイズと論理レコード長はコンパイラーが選択します。

入力 (SYSIN)

コンパイラーへの入力は、SYSIN という名前の DD ステートメントによって定義されたデータ・セットでなければなりません。このデータ・セットは CONSECUTIVE 編成でなければなりません。入力は 1 つ以上の外部 PL/I プロシージャでなければなりません。単一のジョブまたはジョブ・ステップで複数の外部プロシージャをコンパイルしたい場合は、各プロシージャの前に %PROCESS ステートメントを置きます (最初のプロシージャの前には不要な場合があります)。

PL/I ソース・プログラムの入力メディアとして、80 バイト・レコードが一般に使用されます。入力データ・セットは、直接アクセス装置または他の一部の順次メディアに置くことができます。入力データ・セットには、固定長レコード (ブロック化または非ブロック化のもの)、可変長レコード (コード化または非コード化のもの)、または不定長レコードのいずれでも入れることができます。最大レコード・サイズは 100 バイトです。

コンパイル時オプションの指定

入力ファイルの最大行数は 999,999 です。

コンパイラーへの入力のためにデータ・セットを連結する場合は、連結されたデータ・セットは同じような特性 (例えば、ブロック・サイズやレコード・フォーマット) を持っていなければなりません。

出力 (SYSLIN、SYSPUNCH)

コンパイラーからの 1 つ以上のオブジェクト・モジュール形式の出力は、コンパイル時オプション **OBJECT** を指定した場合には、データ・セット **SYSLIN** に保管することができます。このデータ・セットは **DD** ステートメントで定義されます。

オブジェクト・モジュールは、常に、ブロック化または非ブロック化の 80 バイトの固定長レコードの形式になります。**BLKSIZE** が **SYSLIN** に指定され、**BLKSIZE** の値が 80 以外であれば、**LRECL** を 80 として指定しなければなりません。

SYSLIN DD は、一時データ・セットまたは永続データ・セットのどちらかを指定する必要があります。データ・セットのタイプにかかわらず、連結したデータ・セットを指定することはできません。

SYSLIN DD は、**PDS** や **PDSE** ではなく、順次データ・セットを指定しなければなりません。

MDECK コンパイル時オプションを指定した場合、**SYSPUNCH** という名前の **DD** ステートメントで定義されるデータ・セットは、プリプロセッサからの出力の保管にも使用されます。

一時作業ファイル (SYSUT1)

コンパイラーは、一時作業ファイルとして使用するためのデータ・セットを必要とします。このデータ・セットは、**SYSUT1** という名前の **DD** ステートメントで定義され、予備ファイルと呼ばれます。これは直接アクセス装置上にある必要があり、マルチボリューム・データ・セットとして割り振ってはなりません。

予備ファイルは、主記憶域の論理拡張部分として使用され、テキストと辞書情報を入れるためにコンパイラーとプリプロセッサによって使用されます。**SYSUT1** の **LRECL** および **BLKSIZE** は、予備ファイル・ページに使用できるストレージの量に基づいて、コンパイラーが選択します。

本書で記述される **DD** ステートメントおよび **SYSUT1** 用のカタログ式プロシージャの中の **DD** ステートメントは、1024 バイト・ブロック単位のスペース割り振りを要求します。これは直接アクセス・ストレージ・スペースの適切な 2 次割り振りが獲得されるようにするためです。

リスト (SYSPRINT)

コンパイラーは、処理したすべてのソース・ステートメント、オブジェクト・モジュールに関連した情報、および、必要に応じてメッセージの入ったリストを生成します。リストに入れられる情報の大半はオプション情報であり、適切なコンパイル時オプションを入れれば、ユーザーが必要とする情報部分を指定することができます。表示されることがある情報、および関連したコンパイル時オプションについては、94 ページの『コンパイラー・リストの使用』で説明しています。

コンパイラーにそのリストを保管させるデータ・セットを、SYSPRINT という名前の DD ステートメントで指定する必要があります。このデータ・セットは CONSECUTIVE 編成でなければなりません。リストは通常印刷されますが、任意の順次装置または直接アクセス装置に保管することもできます。ご使用のシステムで出力クラス A がプリンターを指すきまりになっている場合は、印刷出力には次のステートメントで十分です。

```
//SYSPRINT DD SYSOUT=A
```

ソース・ステートメント・ライブラリー (SYSLIB)

%INCLUDE ステートメントを使用してライブラリーから PL/I プログラムヘソース・ステートメントを取り込む場合は、SYSLIB という名前の DD ステートメントにライブラリーを定義できます。あるいは、独自の DD 名 (複数可) を選択して、各 %INCLUDE ステートメントに DD 名を指定することもできます。

DD は、PDS または PDSE を指定しますが、実メンバーではありません。例えば、データ・セット INCLUDE.PLI を使用して、ファイル HEADER をライブラリー SYSLIB からインクルードするときの %INCLUDE ステートメントは、次のようになります。

```
%INCLUDE HEADER;
```

または

```
%INCLUDE SYSLIB( HEADER );
```

DD ステートメントは次のようになります。

```
SYSLIB DD DISP=SHR,DSN=INCLUDE.PLI
```

しかし、次のようにはなりません。

```
SYSLIB DD DISP=SHR,DSN=INCLUDE.PLI(HEADER)
```

%INCLUDE ファイルはすべて、SYSIN ソース・ファイルのように、同じレコード・フォーマット (固定、可変、未定義)、同じ論理レコード長を持ち、左右のマージンが一致するフォーマットのものでなければなりません。

ライブラリーの BLOCKSIZE は 32,760 バイト以下でなければなりません。

どのインクルード・ファイルであれ、1 ファイル当たりの最大行数は 999,999 です。

オプションの指定

それぞれのコンパイルごとに、%PROCESS ステートメントで、あるいは EXEC ステートメントの PARM パラメーターでオプションを指定して、デフォルトを指定変更した場合を除き、コンパイル時オプションとして IBM 提供のデフォルトまたはご使用のシステムのデフォルトが適用されます。

PARM パラメーターに指定したオプションはデフォルト値を指定変更します。また、%PROCESS ステートメントに指定したオプションは、PARM パラメーターで指定した値とデフォルト値の両方を指定変更します。

注: 矛盾する属性が他のオプションの指定により明示的または暗黙的に指定された場合は、最新の暗黙的または明示的なオプションが受け入れられます。このように指定変更されるオプションについては、診断メッセージは出されません。

EXEC ステートメントでのオプションの指定

EXEC ステートメントでオプションを指定するには、次の例のとおり、`PARM=` をコーディングし、その後に任意の順のオプションのリストを続け、おのおののオプションをコンマで区切り、リストを単一引用符で囲みます。

```
//STEP1 EXEC PGM=IBMZPLI,PARM='OBJECT,LIST'
```

`MARGINI('c')` のように引用符があるオプションでは、引用符を二重にする必要があります。オプション・リストの長さは、分離文字コンマを含めて 100 文字を超えてはなりません。ただし、多くのオプションには省略構文があり、それを使用してスペースを節約できます。ステートメントが次行へ続くときには、オプション・リストを括弧 (引用符ではなく) で囲み、各行内のオプション・リストを引用符で囲み、そして、最終行を除く各行の最後のコンマが引用符の外側にくるようにしなければなりません。上述のことがらをすべて含んだ例を次に示します。

```
//STEP1 EXEC PGM=IBMZPLI,PARM=('AG,A',  
//      'C,F(I)'),  
// 'M,MI('X'),NEST,STG,X')
```

カタログ式プロシージャーを使用していて、オプションを明示的に指定したい場合は、そのオプションを呼び出す EXEC ステートメントに `PARM` パラメーターを組み込み、コンパイラーを呼び出すプロシージャー・ステップの名前でキーワード `PARM` を修飾する必要があります。次に例を示します。

```
//STEP1 EXEC nnnnnnn,PARM.PLI='A,LIST'
```

オプション・ファイルを使用した EXEC ステートメントでのオプションの指定

EXEC ステートメントでオプションを指定するもう 1 つの方法は、オプション・ファイルの中ですべてのオプションを宣言し、次のようにコーディングするというものです。

```
//STEP1 EXEC PGM=IBMZPLI,PARM='+DD:OPTIONS'
```

この方法を使うと、よく使用するオプションの整合性のあるセットを作成できます。これは他のプログラマーに共通のオプション・セットを使ってもらいたい場合に特に効果的です。また、100 文字の制限もなくなります。

`MARGINS` オプションは、オプション・ファイルには適用されません。1 桁目のデータは、オプションの一部として読み取られます。また、ファイルが `F` フォーマットの場合、72 桁目より後のデータは無視されます。

`parm` ストリングには「通常」のオプションを使用でき、複数のオプション・ファイルを指定することもできます。例えば、オプション `LIST` を指定し、さらに `GROUP DD` と `PROJECT DD` 内の両方のファイルからオプションを指定する場合は、次のように指定できます。

```
PARM='LIST +DD:GROUP +DD:PROJECT'
```

PROJECT ファイルのオプションは、GROUP ファイルのオプションより優先されます。

またこの例では、どちらかのオプション・ファイルに NOLIST オプションを指定することによって、LIST オプションをオフにすることもできます。LIST が必ずオンになるようにするには、次のように指定します。

```
PARM='+DD:GROUP +DD:PROJECT LIST'
```

オプション・ファイルは、z/OS UNIX 環境でも使用できます。例えば、z/OS UNIX で、ファイル /u/pli/group.opt のオプションを使用して sample.pli をコンパイルするには、次のように指定します。

```
pli -q+/u/pli/group.opt sample.pli
```

コンパイラーの旧リリースでは、オプション・ファイルの指定の前に付けるトリガー文字として、文字 '@' を使用していました。この文字は EBCDIC コード・ポイントのインバリアント・セットに含まれないため、文字 '+' (インバリアント) の使用をお勧めします。ただし、16 進値 '7C'x を使用して指定すれば、'@' 文字も引き続き使用できます。

第 5 章 リンク・エディットと実行

コンパイルが終わったプログラムは、相互に対する未解決の参照ならびに、言語環境プログラムランタイム・ライブラリーに対する参照の入った 1 つ以上のオブジェクト・モジュールで構成されています。これらの参照は、リンク・エディット時 (静的) または実行時 (動的) に解決されます。静的にリンク・エディットを行うには、次の 2 つの方法があります。

1. 従来のリンク・ステップの前にプリリンカーを使用する。
2. プリリンカーを使用しないでリンクを行う。これは PL/I for MVS & VM によるリンクと似ていますが、使用するコンパイル時オプションによっては、作成されるロード・モジュールを保持するために PDSE を使用する必要が生じることがあります。

したがって、PL/I プログラムのコンパイルが終わった後で次にとるべきステップは、そのプログラムをテスト・データを用いてリンクして実行し、予想どおりの結果が出るかどうか確認することです。Enterprise PL/I を使用するときは、(上の項目 2 で説明したように) プリリンカーを使用しないリンク方法を選択することをお勧めします。

言語環境プログラムは、ユーザーがプログラムを実行するのに必要なランタイム環境とサービスを提供します。PL/I および他のすべての言語環境プログラムに準拠した言語プログラムのリンクと実行については、「z/OS 言語環境プログラム プログラミングの手引き」を参照してください。既存の PL/I プログラムから言語環境プログラムへの移行については、「Enterprise PL/I for z/OS コンパイラーおよびランタイム 移行ガイド」を参照してください。

リンク・エディットに関する考慮事項

オプション RENT、または $n > 8$ を指定したオプション LIMITS(EXTNAME(n)) を使用してコンパイルする場合は、プリリンカーを使用するか、リンカー出力用に PDSE を使用する必要があります。

バインダーの使用

バインダー出力は PDSE 内に配置する必要があります。

DLL をリンクする際には、必要な定義サイド・デックをバインド・ステップの中で指定する必要があります。

バインダーは、プリリンカーやリンケージ・エディターの代わりに使用できますが、次の例外があります。

- CICS Transaction Server 1.3 より前にリリースされた CICS は、PDSE をサポートしていません。CICS Transaction Server 1.3 以降では、CICS に PDSE のサポートが備わっています。前提条件 APAR フィックスについては、「CICS Transaction Server for z/OS リリース・ガイド」(GC88-8662) を参照してください。

- MTF は PDSE をサポートしません。

プリリンカーの使用

プリリンカーを使用する場合、入力オブジェクト・デック内の外部参照を定義するオブジェクト・デックすべてを、1 つのジョブ・ステップ内で相互にプリリンクする必要があります。

例えば、A と B という別個にコンパイルされたプログラムがある場合、A が B を静的に呼び出す場合、A と B を別個にプリリンクしてから、後で相互にリンクさせることはできません。その代わりに、A と B を一度のプリリンク処理で相互にリンクする必要があります。

ENTRY カードの使用

フェッチされ、そのエントリー・ポイントとして Enterprise PL/I ルーチンを持っているモジュールをビルドしている場合には、ENTRY カードでその PL/I エントリー・ポイントの名前を指定する必要があります。モジュールが Enterprise PL/I からフェッチされる場合、強くはお勧めしませんが、ENTRY カードに CEESTART を指定することができます。ただし、モジュールが COBOL またはアセンブラーからフェッチされる場合は、ENTRY カードには、モジュール内の PL/I エントリー・ポイントの名前を必ず指定するようにし、CEESTART を指定してはなりません。

実行時の考慮事項

プログラム初期化ルーチンに渡されるパラメーターとしてランタイム・オプションを指定できます。また、PLIXOPT 変数の中でランタイム・オプションを指定できます。PLIXOPT 変数を使用してランタイム・オプションを指定し、既存のプログラムを変更してから再コンパイルするのも、パフォーマンスの観点から有利な場合があります。PLIXOPT の使用については、「z/OS 言語環境プログラム プログラミング・ガイド」を参照してください。

端末の入出力を簡素化するため、端末に割り当てられるストリーム・ファイルには各種規則が取り入れられています。次の 3 つの領域が影響を受けます。

1. PRINT ファイルのフォーマット設定
2. 自動プロンプト機能
3. 入力のスペーシングと句読法の規則

注: 端末でのレコード入出力にはプロンプトその他の機能は提供されていません。したがって、端末から、または端末への伝送にはストリーム入出力を使用することを強くお勧めします。

PRINT ファイルのフォーマット設定に関する規則

端末に PRINT ファイルが割り当てられる場合、そのファイルは印刷されるとおりに読み込まれると想定されています。したがって、印刷時間を短縮するために、スペーシングは最小限に行われます。PAGE、SKIP、および ENDPAGE キーワードには、次の規則が適用されます。

- PAGE オプションまたはフォーマット項目は、3 行スキップする原因となります。

- SKIP (2) より大きい SKIP オプションやフォーマット項目は、3 行スキップする原因となります。SKIP (2) 以下は、通常の方法で処置されます。
- ENDPAGE 条件が発生することはありません。

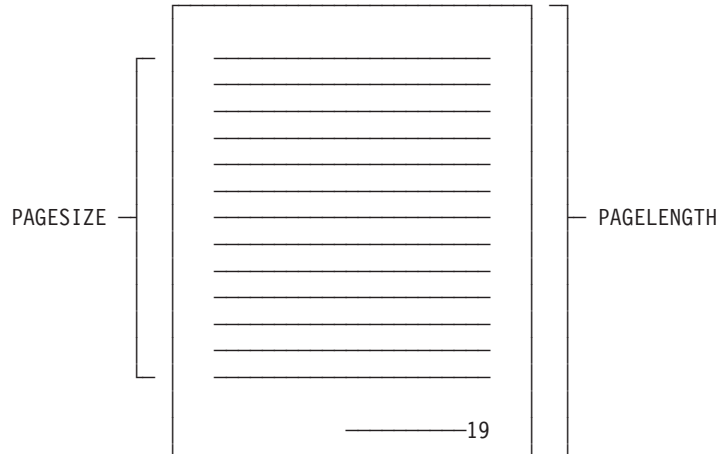
PRINT ファイルでのフォーマットの変更

端末で PRINT ファイルからの出力に通常のスペーシングを適用したい場合は、PL/I 用のタブ・テーブルを独自に提供する必要があります。そのためには、メインプログラム内またはメインプログラムとリンクされるプログラム内で PLITABS という外部構造体を宣言し、エレメント PAGELENGTH をページに収まる行数に初期化します。この値は PAGESIZE とは異なります。PAGESIZE は、ENDPAGE になる前にページに印刷したい行数を定義します (170 ページの図 15 を参照)。64 行の PAGELENGTH が必要な場合は、図 14 に示すように PLITABS を宣言します。タブ・テーブルの指定変更の詳細については、238 ページの『タブ制御テーブルの指定変更』を参照してください。

ご使用のコードが PLITABS の宣言を含む場合は、ページ・サイズ、行のサイズ、およびその他の値が有効であるだけでなく、PLITABS 構造体の最初のフィールドも有効でなければなりません。このフィールドには、構造体が設定されるタブの数を指定しているフィールドへのオフセットが入っている必要があります。そうでない場合、Enterprise PL/I ライブラリー・コードは正しく機能しません。

```
DCL 1 PLITABS STATIC EXTERNAL,
  ( 2  OFFSET INIT (14),
    2  PAGESIZE INIT (60),
    2  LINESIZE INIT (120),
    2  PAGELENGTH INIT (64),
    2  FILL1 INIT (0),
    2  FILL2 INIT (0),
    2  FILL3 INIT (0),
    2  NUMBER_OF_TABS INIT (5),
    2  TAB1 INIT (25),
    2  TAB2 INIT (49),
    2  TAB3 INIT (73),
    2  TAB4 INIT (97),
    2  TAB5 INIT (121)) FIXED BIN (15,0);
```

図 14. PLITABS の宣言： これは標準ページ・サイズ、行サイズ、およびタブ位置を決める宣言です。



PAGELENGTH: 1 ページに印刷可能な行数

PAGESIZE: ENDPAGE 条件が生じるまでに
1 ページに印刷される行数

図 15. *PAGELENGTH* および *PAGESIZE* : *PAGELENGTH* は用紙のサイズを定義し、*PAGESIZE* はメイン印刷域の行数を定義します。

自動プロンプト

プログラムは、端末に関連したファイルからの入力が必要になると、プロンプトを出します。このプロンプトは、次の行にコロンを印刷してから、コロンの次の行の 1 桁目にスキップするという形をとります。これで、次のように、ユーザーが入力するのに 1 行全部を使用することができます。

```
:
```

(データ入力用のスペース)

この種のプロンプトを 1 次プロンプトと呼びます。

自動プロンプトの指定変更

1 次プロンプトを指定変更するには、データ要求の最後の項目をコロンにします。2 次プロンプトは指定変更できません。例えば、次の 2 つの PL/I ステートメントがあるとして。

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);  
GET EDIT (PERITIME) (A(10));
```

これによって、端末は次のものを表示します。

```
ENTER TIME OF PERIHELION  
:          (automatic prompt)  
(space for entry of data)
```

ただし次のように、最初のステートメントで、出力の終わりにコロンがある場合、

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

順序は次のようになります。

ENTER TIME OF PERIHELION: (space for entry of data)

注: 指定変更は 1 つのプロンプトにのみ有効です。自動プロンプトを指定変更しない限り、次の項目に関するプロンプトが自動的に出されます。

長い入力行の句読法

行継続文字

端末で 2 行以上のスペースを必要とするデータを 1 つのデータ項目として送るには、最後の行を除く各行の最後に SBCS ハイフンをタイプします。例えば、“this data must be transmitted as one unit.” という文を送るには、次のように入力します。

```
: 'this data must be transmitted -  
+: as one unit.'
```

“unit.” の後で ENTER を押すまで、送信は行われません。ハイフンは除去されます。送信された項目を「論理行」と呼びます。

注: 最後のデータ文字がハイフンまたは PL/I 負符号である行を送るには、行の終わりに 2 つのハイフンを入力し、次の行をヌル行にします。次に例を示します。

```
xyz--  
(press ENTER only, on this line)
```

GET LIST ステートメントと GET DATA ステートメントの句読法

GET LIST ステートメントと GET DATA ステートメントの場合、プログラマーがコンマを省略すると、端末から送信される各論理行の終わりに、コンマが追加されます。したがって、項目を別々の論理行に入力する場合は、項目を区切るためのブラークまたはコンマを入力する必要はありません。PL/I ステートメント GET LIST(A,B,C) に関しては、端末において以下のように入力することができます。

```
:1  
+:2  
+:3
```

この規則は、文字ストリング・データの入力にも適用されます。したがって、1 つの文字ストリングは 1 つの論理行として送信する必要があります。そうしないと、ブレークポイントにコンマが置かれます。例えば、次のように入力したとします。

```
: 'COMMAS SHOULD NOT BREAK  
+: UP A CLAUSE.'
```

入力結果のストリングは、“COMMAS SHOULD NOT BREAK, UP A CLAUSE.” となります。行継続文字としてハイフンを用いれば、コンマは追加されません。

GET EDIT での自動埋め込み

GET EDIT ステートメントに関しては、行の終わりにブラークを入力する必要はありません。データは指定された長さまで埋め込まれます。例えば次の PL/I ステートメントの場合、

```
GET EDIT (NAME) (A(15));
```

SMITH という 5 文字を入力できます。データは 10 個のブランクで埋め込まれ、プログラムは次のように 15 文字を受け取ります。

```
'SMITH          '
```

注: 単一のデータ項目は 1 つの論理行として送信する必要があります。そうしないと、送信される最初の行に必要なブランクが埋め込まれ、完了したデータ項目と見なされます。

端末入力での SKIP の使用: 入力での SKIP の使用は、すべてファイルが端末に割り振られるときに SKIP(1) として解釈されます。SKIP(1) は、現在使用可能な論理行にある未使用データをすべて無視するという命令として扱われます。

ENDFILE

端末で、2 つの文字 “/*” からなる論理行をキー入力して、ファイルの終わりを入力できます。以後、クローズせずにファイルを使用しようとする、ENDFILE 条件になります。

SYSPRINT の考慮事項

PL/I 標準 SYSPRINT ファイルは、アプリケーション内で複数のエンクレーブによって共用されます。同じあるいは異なるエンクレーブから、例えば STREAM PUT などの入出力要求を出すことができます。これらの要求は標準 PL/I SYSPRINT ファイルを、全アプリケーション共通のファイルとして使用して処理されます。SYSPRINT ファイルは、エンクレーブの終了時ではなく、アプリケーションが終了するときのみ暗黙的に閉じられます。

標準 PL/I SYSPRINT ファイルには、STREAM PUT などのユーザー開始出力のみが入っています。ランタイム・ライブラリー・メッセージおよび他の類似診断出力は、言語環境プログラム MSGFILE へ向けられます。SYSPRINT ファイル出力を言語環境プログラム MSGFILE にリダイレクトする処理の詳細については、「z/OS 言語環境プログラム プログラミング・ガイド」を参照してください。

アプリケーション内で複数のエンクレーブによって共用されるためには、PL/I SYSPRINT ファイルは SYSPRINT のファイル名で EXTERNAL FILE 定数として宣言されなければならない、また属性 STREAM および OUTPUT ならびに暗黙の PRINT(OPEN 処理されるとき) も持たなくてはなりません。これはコンパイラによってデフォルトとされる標準 SYSPRINT ファイルです。

アプリケーション内にはただ 1 つの標準 PL/I SYSPRINT FILE が存在し、このファイルはそのアプリケーション内の全エンクレーブによって共用されます。例えば、SYSPRINT ファイルはアプリケーション内の多重ネスト・エンクレーブによる共用が可能であり、また、言語環境プログラム事前初期化機能によってアプリケーション内で作成され終了される一連のエンクレーブによる共用が可能です。アプリケーション内でエンクレーブによって共用されるためには、PL/I SYSPRINT ファイルはそのエンクレーブ内で宣言されなくてはなりません。標準 SYSPRINT ファイルは、エンクレーブ間でファイル引数としてそれを渡すことによって共用することはできません。標準 SYSPRINT ファイルの宣言済み属性は、アプリケーション内では EXTERNAL として宣言された定数によるのと同様、同じでなくてはなりません。PL/I はこの規則を強制しません。TITLE オプションと MSGFILE(SYSPRINT) オプ

ションは、どちらも SYSPRINT を別のデータ・セットへ経路指定しようとしします。したがって、この 2 つのオプションを一緒に使用すると、矛盾が起こり、TITLE オプションは無視されます。

共通 SYSPRINT ファイルをアプリケーション内で持つことは、互いに緊密に結ばれたエンクレープを利用するアプリケーションにとっては利点となります。しかし、アプリケーション内のすべてのエンクレープは、同じ共用データ・セットに書き込みを行うので、各エンクレープ間での何らかの調整が必要となります。

SYSPRINT ファイルは、アプリケーションのエンクレープ内で最初に参照が行われた時にオープンされます (暗黙的にあるいは明示的に)。SYSPRINT ファイルが CLOSE 処理されると、ファイル・リソースは解放され (あたかもファイルがオープンされていなかったかのように)、また全エンクレープは閉じられた状況を反映するように更新されます。

SYSPRINT が複数のエンクレープ・アプリケーション内で利用される場合は、LINENO 組み込み関数はエンクレープ内の最初の PUT あるいは OPEN が出されるまで現在の行番号を返すだけです。これは旧プログラムとの完全な互換性を維持するために必要です。

COUNT 組み込み関数はエンクレープ・レベルにおいて維持されます。これは常に、エンクレープ内の最初の PUT が出されるまでゼロの値を返します。ネスト化された子エンクレープが親エンクレープから呼び出される場合、COUNT 組み込み関数の値は、親エンクレープが子エンクレープから制御を取り戻したときは未定義です。

TITLE オプションを使用して、標準 SYSPRINT ファイルを異なるオペレーティング・システム・データ・セットと関連付けることができますが、特定のオープン関連付けは、別のものをオープンする前にクローズする必要があることに留意してください。この関連付けはオープン状態が続く間は各エンクレープ間で保持されます。

標準 PL/I SYSPRINT ファイルと関連する PL/I 条件処理は、その現行のセマンティクスと有効範囲を保持します。例えば、子エンクレープ内で生じた ENDPAGE 条件は、その子エンクレープ内で、設定された ON ユニットを呼び出すだけです。これは親エンクレープ内の ON ユニットの呼び出しは行いません。

標準 PL/I SYSPRINT ファイルのタブは、エンクレープがユーザー PLITABS テーブルを含む場合、PUT が異なるエンクレープから実行されたときは変わる可能性があります。

PL/I SYSPRINT ファイルが RECORD ファイルあるいは STREAM INPUT ファイルとして利用される場合、PL/I は個々のエンクレープあるいはタスク・レベルでそれをサポートしますが、エンクレープ間の共用可能ファイルとしてはサポートしません。同じアプリケーションの異なるエンクレープ内の異なるファイル属性 (例えば RECORD および STREAM) において PL/I SYSPRINT ファイルがオープンされると、結果は予測不能のものとなります。

SYSPRINT は、Enterprise PL/I コンパイラーによってコンパイルされたコードと以前の PL/I コンパイラーによってコンパイルされたコードの間で共有することもできます。ただし、そのためには以下の要件をすべて適用する必要があります。

- SYSPRINT は STREAM OUTPUT として宣言されなければならない。
- アプリケーションを TSO 環境で実行することはできない。
- ランタイム・オプション MSGFILE(SYSPRINT) が有効になっている場合、アプリケーション内に、事前初期設定されたプログラムおよびストアード・プロシージャが存在してはいけません。

ルーチン内での FETCH の使用

Enterprise PL/I では、PL/I、C、COBOL またはアセンブラーによってコンパイルされたルーチンをフェッチできます。

Enterprise PL/I ルーチンのフェッチ

旧 PL/I コンパイラによって課せられた、フェッチされたモジュールの制限は、ほぼすべて除去されました。したがって、FETCH されたモジュールは次のことが行えます。

- 他のモジュールをフェッチする。
- 任意の PL/I ファイルに入出力操作を実行する。ファイルは、フェッチされたモジュール、メイン・モジュール、または他のフェッチされたモジュールによりオープンできます。
- 独自の CONTROLLED 変数の ALLOCATE と FREE を実行する。

しかし、フェッチの対象である Enterprise PL/I モジュールに対するいくつかの制限があります。それらは次のとおりです。

1. OPTIONS(FETCHABLE) は、フェッチされる側のモジュール内のエントリー・ポイントを提供するルーチンの PROCEDURE ステートメントに指定する必要があります。あるいは DLLINIT コンパイラ・オプションを使用して、そのプロシージャに OPTIONS(FETCHABLE) を適用する必要があります。
2. ENTRY カードには、PL/I エントリー・ポイントの名前を指定する必要があります。
 - モジュールが Enterprise PL/I からフェッチされる場合、強くはお勧めしませんが、ENTRY カードに CEESTART を指定することができます。
 - ただし、モジュールが COBOL またはアセンブラーからフェッチされる場合は、ENTRY カードには、モジュール内の PL/I エントリー・ポイントの名前を必ず指定するようにし、CEESTART を指定してはなりません。
3. フェッチされる側のコードのいずれかをコンパイルするために RENT コンパイラ・オプションが使用された場合、そのモジュールは DLL としてリンクする必要があります。
4. フェッチする側のコードのコンパイルに NORENT コンパイラ・オプションが使用された場合、フェッチされる側のすべてのモジュールが NORENT コードのみで構成されていなければなりません。
5. フェッチする側のコードのコンパイルに RENT コンパイラ・オプションが使用された場合、FETCH される側の ENTRY が、フェッチする側のモジュールに OPTIONS(COBOL) または OPTIONS(ASM) として宣言されていないと見なされます。この状態で記述子を渡すのを避けたい場合には、ENTRY 宣言に OPTIONS(NODESCRIPTOR) 属性を指定する必要があります。

NORENT WRITABLE コードは逐次使用可能です。そのため、FFETCHABLE 定数を示すために使用されるポインターは、すべての NORENT WRITABLE ルーチンのプロローグ・コードで、ゼロにリセットされます。これによって、コードが正しい PL/I セマンティクスも提供しながら逐次再使用可能になることが保証されますが、NORENT WRITABLE コードにおいて TITLE を指定した FETCH の使用に制限が課されます。この制限によって、FETCH A TITLE('B') を行ったルーチンでは、終了して再入した場合、CALL A ステートメントを実行する前に FETCH A TITLE('B') を再実行する必要があります (そうしなければ、CALL の実行前に暗黙の (TITLE のない) A の FETCH が実行されます)。

これらの制限を説明するために、コンパイラー・ユーザー出口を考えてみます。EXIT コンパイル時オプションを指定すると、コンパイラーは IBMUEEXIT という名前の Enterprise PL/I モジュールをフェッチして呼び出します。

まず、RENT オプションを指定してコンパイラー・ユーザー出口をコンパイルする必要があるので注意してください。コンパイラーは、このユーザー出口が DLL であることを前提としているからです。

上の項目 1 により、このルーチンに関するコンパイラーの PROCEDURE ステートメントは次のようになります。

```
ibmuexit:
  proc ( addr_Userexit_Interface_Block,
         addr_Request_Area )
    options( fetchable );

    dcl addr_Userexit_Interface_Block pointer byvalue;

    dcl addr_Request_Area                pointer byvalue;
```

上の項目 3 により、ユーザー出口を DLL にリンクするときに、リンカー・オプション DYNAM=DLL を指定する必要があります。DLL は、PDSE または一時データ・セットのどちらかにリンクする必要があります (一次データ・セットにリンクする場合は、DSNTYPE=LIBRARY を SYSLMOD DD ステートメントに指定する必要があります)。

ユーザー出口のコンパイル、リンク、および呼び出しを行うための JCL ステートメントはすべて、176 ページの図 16 の JCL に示されています。下記のサンプル・コードでは、フェッチされたユーザー出口は構造体を指す 2 つの BYVALUE ポインターを受け取らず、代わりに 2 つの構造体 BYADDR を受け取ります。これが上のコード抜粋との大きな相違です。この変更を有効にするために、コードではその PROCEDURE ステートメントのそれぞれに OPTIONS(NODESCRIPTOR) を指定しています。

```

/**
/*****
/** compile the user exit
/*****
//PLIEXIT EXEC PGM=IBMZPLI,
//      REGION=256K
//STEPLIB DD DSN=IBMZ.V3R8M0.SIBMZCMP,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSSQ,
//      SPACE=(CYL,(3,1))
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,
//      SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN DD *
*Process or('|') not('!');
*Process limits(extname(31));

/*****
/*
/* NAME - IBMUEXIT.PLI
/*
/*
/* DESCRIPTION
/* User-exit sample program.
/*
/*
/* Licensed Materials - Property of IBM
/* 5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,2008.
/* All Rights Reserved.
/* US Government Users Restricted Rights-- Use, duplication or
/* disclosure restricted by GSA ADP Schedule Contract with
/* IBM Corp.
/*
/*
/* DISCLAIMER OF WARRANTIES
/* The following "enclosed" code is sample code created by IBM
/* Corporation. This sample code is not part of any standard
/* IBM product and is provided to you solely for the purpose of
/* assisting you in the development of your applications. The
/* code is provided "AS IS", without warranty of any kind.
/* IBM shall not be liable for any damages arising out of your
/* use of the sample code, even if IBM has been advised of the
/* possibility of such damages.
/*
/*
/*****

/*****
/*
/* During initialization, IBMUEXIT is called. It reads
/* information about the messages being screened from a text
/* file and stores the information in a hash table. IBMUEXIT
/* also sets up the entry points for the message filter service
/* and termination service.
/*
/*
/* For each message generated by the compiler, the compiler
/* calls the message filter registered by IBMUEXIT. The filter
/* looks the message up in the hash table previously created.
/*
/*
/* The termination service is called at the end of the compile
/* but does nothing. It could be enhanced to generates reports
/* or do other cleanup work.
/*
/*
/*****

```

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (1/7)

```

pack: package exports(*);

Dcl
  1 Uex_UIB          native Based( null() ),
  2 Uex_UIB_Length   fixed bin(31),

  2 Uex_UIB_Exit_token  pointer,          /* for user exit's use*/

  2 Uex_UIB_User_char_str pointer,          /* to exit option str */
  2 Uex_UIB_User_char_len fixed bin(31),

  2 Uex_UIB_Filename_str pointer,          /* to source filename */
  2 Uex_UIB_Filename_len fixed bin(31),

  2 Uex_UIB_return_code fixed bin(31),     /* set by exit procs */
  2 Uex_UIB_reason_code fixed bin(31),     /* set by exit procs */

  2 Uex_UIB_Exit_Routs,                    /* exit entries setat
                                           initialization */

  3 ( Uex_UIB_Termination,
      Uex_UIB_Message_Filter,              /* call for each msg */
      *, *, *, * )
      limited entry (
        *,                                /* to Uex_UIB */
        *,                                /* to a request area */
      );

  /******
  /*
  /* Request Area for Initialization exit
  /*
  /******

Dcl 1 Uex_ISA native based( null() ),
  2 Uex_ISA_Length fixed bin(31);

  /******
  /*
  /* Request Area for Message_Filter exit
  /*
  /******

Dcl 1 Uex_MFA native based( null() ),
  2 Uex_MFA_Length   fixed bin(31),
  2 Uex_MFA_Facility_Id char(3),
  2 *                char(1),
  2 Uex_MFA_Message_no fixed bin(31),
  2 Uex_MFA_Severity   fixed bin(15),
  2 Uex_MFA_New_Severity fixed bin(15); /* set by exit proc */

  /******
  /*
  /* Request Area for Terminate exit
  /*
  /******

Dcl 1 Uex_TSA native based( null() ),
  2 Uex_TSA_Length fixed bin(31);

```

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (2/7)

```

/*****
/*
/*   Severity Codes
/*
/*
/*****

dc1 uex_Severity_Normal          fixed bin(15) value(0);
dc1 uex_Severity_Warning        fixed bin(15) value(4);
dc1 uex_Severity_Error          fixed bin(15) value(8);
dc1 uex_Severity_Severe         fixed bin(15) value(12);
dc1 uex_Severity_Unrecoverable  fixed bin(15) value(16);

/*****
/*
/*   Return Codes
/*
/*
/*****

dc1 uex_Return_Normal          fixed bin(15) value(0);
dc1 uex_Return_Warning        fixed bin(15) value(4);
dc1 uex_Return_Error          fixed bin(15) value(8);
dc1 uex_Return_Severe         fixed bin(15) value(12);
dc1 uex_Return_Unrecoverable  fixed bin(15) value(16);

/*****
/*
/*   Reason Codes
/*
/*
/*****

dc1 uex_Reason_Output          fixed bin(15) value(0);
dc1 uex_Reason_Suppress       fixed bin(15) value(1);

dc1 hashsize fixed bin(15) value(97);
dc1 hashtable(0:hashsize-1) ptr init((hashsize) null());

dc1 1 message_item native based,
    2 message_Info,
    3 facid char(3),
    3 msgno fixed bin(31),
    3 newsev fixed bin(15),
    3 reason fixed bin(31),
    2 link pointer;

```

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (3/7)

```

ibmuexit: proc ( ue, ia )
    options( fetchable nodescriptor );

    dcl 1 ue like uex_Uib byaddr;
    dcl 1 ia like uex_Isa byaddr;

    dcl sysuexit    file stream input env(recsize(80));
    dcl p           pointer;
    dcl bucket      fixed bin(31);
    dcl based_Chars char(8) based;
    dcl title_Str   char(8) var;

    ue.uex_Uib_Message_Filter = message_Filter;
    ue.uex_Uib_Termination = exitterm;

    on undefinedfile(sysuexit)
    begin;
        put edit ( '** User exit unable to open exit file ' )
            (A) skip;
        put skip;
        signal error;
    end;

    if ue.uex_Uib_User_Char_Len = 0 then
        do;
            open file(sysuexit);
        end;
    else
        do;
            title_Str
                = substr( ue.uex_Uib_User_Char_Str->based_Chars,
                           1, ue.uex_Uib_User_Char_Len );
            open file(sysuexit) title(title_Str);
        end;

    on error, endfile(sysuexit)
        goto done;

    allocate message_item set(p);

    /*****
    /*
    /* Skip header lines and read first data line
    /*
    /*
    /*****/

    get file(sysuexit) list(p->message_info) skip(3);

```

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (4/7)

```

do loop;

    /******
    /*
    /*  Put message information in hash table
    /*
    /*
    /******

    bucket = mod(p->msgno, hashsize);
    p->link = hashtable(bucket);
    hashtable(bucket) = p;

    /******
    /*
    /*  Read next data line
    /*
    /*
    /******

    allocate message_item set(p);
    get file(sysuexit) skip;
    get file(sysuexit) list(p->message_info);

end;

/******
/*
/*  Clean up
/*
/*
/******

done:

free p->message_Item;
close file(sysuexit);

end;

message_Filter:
proc ( ue, mf )
options( nodedescriptor );

dcl 1 ue like uex_Uib byaddr;
dcl 1 mf like uex_Mfa byaddr;

dcl p pointer;
dcl bucket fixed bin(15);

on error snap system;

ue.uex_Uib_Reason_Code = uex_Reason_Output;
ue.uex_Uib_Return_Code = 0;

mf.uex_Mfa_New_Severity = mf.uex_Mfa_Severity;

/******
/*
/*  Calculate bucket for error message
/*
/*
/******

bucket = mod(mf.uex_Mfa_Message_No, hashsize);

```

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (5/7)

```

/*****
/*
/* Search bucket for error message
/*
/*
*****/

do p = hashtable(bucket) repeat (p->link) while(p!=null())
  until (p->msgno = mf.uex_Mfa_Message_No &
        p->facid = mf.Uex_Mfa_Facility_Id);
end;

if p = null() then;
else
do;

/*****
/*
/* Filter error based on information in has table
/*
/*
*****/

ue.uex_Uib_Reason_Code = p->reason;
if p->newsev < 0 then;
else
mf.uex_Mfa_New_Severity = p->newsev;
end;
end;

exitterm:
proc ( ue, ta )
options( nodedescriptor );

dcl 1 ue like uex_Uib byaddr;
dcl 1 ta like uex_Tsa byaddr;

ue.uex_Uib_return_Code = 0;
ue.uex_Uib_reason_Code = 0;

end;

end pack;

/*****
/* link the user exit
*****/
//LKEDEXIT EXEC PGM=IEWL,PARM='XREF,LIST,LET,DYNAM=DLL',
// COND=(9,LT,PLIEXIT),REGION=5000K
//SYSLIB DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD DD DSN=&&EXITLIB(IBMUEXIT),DISP=(NEW,PASS),UNIT=SYSDA,
// SPACE=(TRK,(7,1,1)),DSNTYPE=LIBRARY
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(3,1)),
// DCB=BLKSIZE=1024
//SYSPRINT DD SYSOUT=X
//SYSDEFSD DD DUMMY
//SYSLIN DD DSN=&&LOADSET,DISP=SHR
// DD DDNAME=SYSIN
//LKED.SYSIN DD *
ENTRY IBMUEXIT

```

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (6/7)

```

//*****
//* compile main
//*****
//PLI EXEC PGM=IBMZPLI,PARM='F(1),EXIT',
//      REGION=256K
//STEPLIB DD DSN=*&EXITLIB,DISP=SHR
//      DD DSN=IBMZ.V3R8M0.SIBMZCMP,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=*&LOADSET2,DISP=(MOD,PASS),UNIT=SYSSQ,
//      SPACE=(CYL,(3,1))
//SYSUT1 DD DSN=*&SYSUT1,UNIT=SYSDA,
//      SPACE=(1024,(200,50),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN DD *
*process;
MainFet: Proc Options(Main);
/* the exit will suppress the message for the next dcl */
dcl one_byte_integer fixed bin(7);
End ;
/*
//SYSUEXIT DD *

```

Fac Id	Msg No	Severity	Suppress	Comment
'IBM'	1042	-1	1	String spans multiple lines
'IBM'	1044	-1	1	FIXED BIN 7 mapped to 1 byte

図 16. ユーザー出口のコンパイル、リンク、および呼び出しのためのサンプル JCL (7/7)

z/OS C ルーチンのフェッチ

NORENT オプションが指定されている場合を除き、z/OS C ルーチンをフェッチするルーチン内の ENTRY 宣言では、OPTIONS(COBOL) または OPTIONS(ASM) は指定できません。これらは DLL としてリンクされていない COBOL ルーチンまたは ASM ルーチンだけに指定してください。

z/OS C DLL のコンパイルとリンクの方法についての指示は、z/OS C 資料にあります。

アセンブラー・ルーチンのフェッチ

NORENT オプションが指定されている場合を除き、アセンブラー・ルーチンをフェッチするルーチン内の ENTRY 宣言は、OPTIONS(ASM) を指定する必要があります。

z/OS UNIX を指定した場合の MAIN の呼び出し

z/OS UNIX 環境では、SYSTEM(MVS) オプションを指定して MAIN プログラムをコンパイルすると、通常どおり、プログラムが呼び出されたときに指定されたパラメーターを含む 1 つの CHARACTER VARYING スtringがプログラムに渡されます。

しかし z/OS UNIX 環境で SYSTEM(OS) オプションを指定して MAIN プログラムをコンパイルすると、z/OS UNIX のマニュアルに記述されている 7 つのパラメーターがプログラムに渡されます。これらの 7 パラメーターは、次を含んでいます。

- 引数カウンタ (第 1 引数として実行可能な名前を含む)
- 複数の引数長さのアドレスを含む、単一の配列のアドレス
- ヌル終了文字ストリングである複数の引数のアドレスを含む、単一の配列のアドレス
- 環境変数セットのカウント
- 複数の環境変数の長さのアドレスを含む、単一の配列のアドレス
- ヌル終了文字ストリングである複数の環境変数のアドレスを含む、単一の配列のアドレス

図 17 内のプログラムは、SYSTEM(OS) インターフェースを使用して、個々の引数と環境変数を参照して表示します。

```

*process display(std) system(os);

sayargs:
  proc(argc, pArgLen, pArgStr, envc, pEnvLen, pEnvStr, pParmSelf)
    options( main, noexecops );

    decl argc          fixed bin(31) nonasgn byaddr;
    decl pArgLen        pointer nonasgn byvalue;
    decl pArgStr        pointer nonasgn byvalue;
    decl envc          fixed bin(31) nonasgn byaddr;
    decl pEnvLen        pointer nonasgn byvalue;
    decl pEnvStr        pointer nonasgn byvalue;
    decl pParmSelf      pointer nonasgn byvalue;

    decl q(4095)        pointer based;
    decl bxb            fixed bin(31) based;
    decl bcz            char(31) varz based;

    display( 'argc = ' || argc );
    do jx = 1 to argc;
      display( 'pargStr(jx) = ' || pArgStr->q(jx)->bcz );
    end;
    display( 'envc = ' || envc );
    do jx = 1 to envc;
      display( 'pEnvStr(jx) = ' || pEnvStr->q(jx)->bcz );
    end;

  end;

```

図 17. z/OS UNIX 引数と環境変数を表示するサンプル・プログラム

第 2 部 入出力機能の使用

第 6 章 データ・セットとファイルの使用	187
z/OS でのデータ・セットとファイルの関連付け	187
複数のファイルと 1 つのデータ・セットの関連付け	189
複数のデータ・セットと 1 つのファイルの関連付け	190
複数のデータ・セットの連結	190
z/OS での HFS ファイルへのアクセス	190
z/OS UNIX でのデータ・セットとファイルの関連付け	191
環境変数の使用	191
OPEN ステートメントの TITLE オプションの使用	192
データ・セットに関連付けられていないファイルの使用の試み	193
PL/I によるデータ・セットの検索方法	194
DD_DDNAME 環境変数の使用による特性の指定	194
APPEND	194
BUFSIZE	195
レコード入出力用の CHARSET	195
ストリーム入出力用の CHARSET	195
DELAY	196
DELIMIT	196
LRECL	196
LRMSKIP	196
PROMPT	197
PUTPAGE	197
RECCOUNT	197
RECSIZE	197
SAMELINE	198
SKIP0	198
TYPE	199
データ・セット特性の設定	200
ブロックおよびレコード	200
情報交換コード	201
レコード・フォーマット	201
固定長レコード	201
可変長レコード	202
不定長レコード	203
データ・セットの編成	203
ラベル	204
データ定義 (DD) ステートメント	204
条件付きサブパラメーターの用法	205
データ・セット特性	205
OPEN ステートメントの TITLE オプションの使用	206
PL/I ファイルとデータ・セットの関連付け	207
ファイルのオープン	207
ENVIRONMENT 属性での特性の指定	208
ENVIRONMENT 属性	208

レコード単位データ伝送のレコード・フォーマット	210
ストリーム指向データ伝送のレコード・フォーマット	211
RECSIZE オプション	211
BLKSIZE オプション	212
レコード・フォーマット、BLKSIZE、および RECSIZE のデフォルト	214
GENKEY オプション - キーの分類	214
SCALARVARYING オプション - 可変長ストリング	216
KEYLENGTH オプション	217
ORGANIZATION オプション	217
PL/I レコード入出力で使用されるデータ・セットのタイプ	217
環境変数の設定	218
PL/I 標準ファイル (SYSPRINT と SYSIN)	219
標準入力、標準出力、および標準エラー装置のリダイレクト	219
第 7 章 ライブラリーの使用	221
ライブラリーのタイプ	221
ライブラリーの用法	222
ライブラリーの作成	222
SPACE パラメーター	222
ライブラリー・メンバーの作成と更新	223
例	224
ライブラリー・ディレクトリーにある情報の取り出し	226
第 8 章 連続データ・セットの定義と使用	227
ストリーム指向データ伝送の使用	227
ストリーム入出力の使用によるファイルの定義	228
ENVIRONMENT オプションの指定	228
CONSECUTIVE	228
レコード・フォーマット・オプション	228
RECSIZE	229
レコード・フォーマット、BLKSIZE および RECSIZE のデフォルト値	229
GRAPHIC オプション	230
ストリーム入出力によるデータ・セットの作成	230
必須情報	230
例	232
ストリーム入出力によるデータ・セットへのアクセス	233
必須情報	234
レコード・フォーマット	234
例	235
ストリーム入出力による PRINT ファイルの使用	235
印刷する行の長さの制御	236
タブ制御テーブルの指定変更	238

SYSIN ファイルおよび SYSPRINT ファイルの 使用方法	240	ENVIRONMENT オプションの指定	278
端末からの入力の制御	241	BKWD オプション	279
データのフォーマット	241	BUFND オプション	279
ストリーム・ファイルおよびレコード・ファイル	242	BUFNI オプション	279
大文字と小文字	243	BUFSP オプション	280
ファイルの終わり	243	GENKEY オプション	280
GET ステートメントの COPY オプション	243	PASSWORD オプション	280
端末への出力の制御	243	REUSE オプション	280
PRINT ファイルのフォーマット	243	SKIP オプション	281
ストリーム・ファイルおよびレコード・ファイル	244	VSAM オプション	281
大文字と小文字	244	パフォーマンス・オプション	281
PUT EDIT コマンドの出力	244	代替索引パス用のファイルの定義	281
レコード単位データ伝送の使用	244	VSAM データ・セットの定義	282
レコード・フォーマットの指定	246	入力順データ・セット	283
レコード入出力の使用によるファイルの定義	246	ESDS のロード	284
ENVIRONMENT オプションの指定	246	SEQUENTIAL ファイルの使用による ESDS へ のアクセス	284
CONSECUTIVE	246	ESDS の定義とロード	284
ORGANIZATION(CONSECUTIVE)	247	ESDS の更新	285
CTLASAICTL360	247	キー順および索引付き入力順データ・セット	286
LEAVEIREREAD	248	KSDS または索引付き ESDS の代替索引	292
レコード入出力によるデータ・セットの作成	249	相対レコード・データ・セット	299
必須情報	250	非 VSAM データ・セット用に定義されたファイル の使用	306
レコード入出力によるデータ・セットへのアクセ スと更新	250	共用データ・セットの使用	306
必須情報	251		
連続データ・セットの例	252		
第 9 章 領域データ・セットの定義と使用	257		
領域データ・セット用のファイルの定義	259		
ENVIRONMENT オプションの指定	260		
REGIONAL オプション	260		
REGIONAL データ・セットでのキーの使用	260		
REGIONAL(1) データ・セットの使用	261		
ダミー・レコード	261		
REGIONAL(1) データ・セットの作成	261		
例	262		
REGIONAL(1) データ・セットへのアクセスと更 新	263		
順次アクセス	264		
直接アクセス	264		
例	264		
領域データ・セットの作成時、および領域データ・ セットへのアクセス時の必須情報	267		
第 10 章 VSAM データ・セットの定義と使用	271		
VSAM データ・セットの使用	271		
VSAM データ・セットでのプログラムの実行	271		
代替索引パスとファイルのベア化	271		
VSAM 編成	272		
VSAM データ・セットのキー	274		
索引付き VSAM データ・セットのキー	275		
相対バイト・アドレス (RBA)	275		
相対レコード番号	275		
データ・セット・タイプの選択	275		
VSAM データ・セットのファイルの定義	277		

第 6 章 データ・セットとファイルの使用

PL/I プログラムは、レコードと呼ばれる情報単位の処理と送信を行います。レコードの集まりをデータ・セットと呼びます。データ・セットは、PL/I プログラムの外部にある情報の物理的な集まりです。つまり、データ・セットは、PL/I またはその他の言語で書かれたプログラムや、オペレーティング・システムのユーティリティー・プログラムによって、作成、アクセス、または変更することができます。

PL/I プログラムは、ファイルと呼ばれるデータ・セットのシンボルによる表現または論理表現を使って、データ・セット内の情報を認識したり処理したりします。本章では、使用しているプログラム内で既知のファイルにデータ・セットを関連付ける方法を説明します。また、データ・セットの主要な 5 つのタイプ、データ・セットの編成およびアクセス方法、および指定方法を理解しておく必要があるファイルやデータ・セットの特性についても説明します。

注: INDEXED は VSAM を暗黙指定し、バッチ環境下でのみサポートされます。

z/OS でのデータ・セットとファイルの関連付け

PL/I プログラムで使用するファイルには、PL/I ファイル名が付いています。プログラムの外部で存在する物理データ・セットの名前は、オペレーティング・システムが認識できる名前、すなわち、データ・セット名 または *dsname* をとります。名前のないデータ・セットもあります。その場合、データ・セットはそのデータ・セットが設定されている装置によってシステムに認識されます。

オペレーティング・システムには、使用しているプログラムが参照する物理データ・セットを識別する手段が必要なので、PL/I ファイル名を *dsname* に関連付ける、プログラムにとっては外部のデータ定義 つまり *DD* ステートメントを記述する必要があります。例えば、次のようなファイル宣言が使用しているプログラムに設定されているとします。

```
DCL STOCK FILE STREAM INPUT;
```

この PL/I ファイル名に対応するデータ定義名 (*ddname*) を指定して *DD* ステートメントを作成しなければなりません。つまり、*DD* ステートメントにより物理データ・セット名 (*dsname*) が指定され、その特性が指定されます。

```
//GO.STOCK DD DSN=PARTS.INSTOCK, . . .
```

DD ステートメントの書き方については本書にも記述されていますが、詳細は使用しているシステムのジョブ制御言語 (JCL) 解説書を参照してください。

データ・セットを PL/I ファイルに関連付ける方法は、いくつかあります。データ・セットを PL/I ファイルに関連付けるには、データ・セットを定義する *DD* ステートメントの *dd* 名を、次のいずれかと同じになるように指定します。

- 宣言された PL/I ファイル名
- 関連する *OPEN* ステートメントの *TITLE* オプションに指定されている式の文字ストリング値

また、対応する dd 名が次の条件を満たすように、PL/I ファイル名を選択する必要があります。

- ファイルを暗黙的にオープンする場合、あるいは、ファイルを明示的にオープンする OPEN ステートメントに TITLE オプションが指定されていない場合は、dd 名はデフォルトでそのファイル名になります。ファイル名が 8 文字を超える場合、デフォルトの dd 名はそのファイル名の最初の 8 文字から構成されます。
- JCL の文字セットには、区切り文字 (,) は含まれません。したがって、区切り文字は dd 名の中に出てくることはできません。有効な dd 名を式として使用する TITLE オプションによりファイルをオープンする場合を除き、ファイル名の最初の 8 文字に区切り文字は使用しないでください。英字の拡張文字 \$、@、および # も dd 名に使用することができますが、dd 名の最初の文字には A から Z までの英字でなければなりません。

外部名は 7 文字までに限定されているため、7 文字を超える外部ファイル名は、そのファイル名の最初の 4 文字と最後の 3 文字を連結したものに短縮されます。しかしこのような短縮名は、関連 DD ステートメント内で dd 名として使用される名前ではありません。

次の 3 つのステートメントを見てみましょう。

1. OPEN FILE(MASTER);
2. OPEN FILE(OLDMASTER);
3. READ FILE(DETAIL) ...;

ステートメント番号 1 を実行すると、ファイル名 MASTER は現行ジョブ・ステップの DD ステートメントの dd 名と同じであると見なされます。ステートメント番号 2 を実行すると、ファイル名 OLDMASTER は現行ジョブ・ステップの DD ステートメントの dd 名と同じであると見なされます。(ファイル名の最初の 8 文字が dd 名になります。例えば、OLDMASTER が外部名であるとする、プログラム内ではコンパイラによって短縮された OLDMASTER という名前が使われます。) ステートメント番号 3 を使ってファイル DETAIL を暗黙的にオープンする場合、ファイル名 DETAIL は現行ジョブ・ステップの DD ステートメントの dd 名と同じであると見なされます。

上記の場合はいずれも、対応する DD ステートメントがジョブ・ストリーム内に存在しなければなりません。それが存在しないと、UNDEFINEDFILE 条件が発生します。上記 3 つの DD ステートメントは、次のように始まります。

1. //MASTER DD ...
2. //OLDMASTER DD ...
3. //DETAIL DD ...

ファイルを明示的あるいは暗黙的にオープンするステートメント内のファイル参照がファイル定数でない場合は、その DD ステートメント名はファイル参照の値と同じでなければなりません。次の例は、DD ステートメントをどのようにファイル変数の値と関連付けるかを示します。

```
DCL PRICES FILE VARIABLE,  
    RPRICE FILE;  
    PRICES = RPRICE;  
    OPEN FILE(PRICES);
```

上記の DD ステートメントは、データ・セットをファイル定数 RPRICE と関連付けるはずですが、このファイル定数は、ファイル変数 PRICES の値です。したがって、次のようになります。

```
//RPRICE DD DSNAME=...
```

また、ファイル変数を使えば、1 つのステートメントで、いくつものファイルを何回も処理することができます。次に例を示します。

```
DECLARE F FILE VARIABLE,  
        A FILE,  
        B FILE,  
        C FILE;  
        .  
        .  
        .  
DO F=A,B,C;  
  READ FILE (F) ...;  
  .  
  .  
  .  
END;
```

上記の場合、READ ステートメントにより 3 つのファイル A、B、C が読み取られ、各ファイルをそれぞれ異なるデータ・セットに関連付けることができます。ファイル A、B、C は、それぞれの場合にこの READ ステートメントが実行されたあとも、オープンの状態に保たれます。

次の OPEN ステートメントは、TITLE オプションの使用を示したものです。

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

このステートメントを正常に実行するには、現行ジョブ・ステップ内に dd 名が DETAIL1 の DD ステートメントがなければなりません。このステートメントは、次のように始まります。

```
//DETAIL1 DD DSNAME=DETAILA,...
```

このように、dd 名 DETAIL1 を使ってデータ・セット DETAILA をファイル DETAIL に関連付けます。

複数のファイルと 1 つのデータ・セットの関連付け

同じ TITLE 名に対する 2 番目のファイル関連付けをオープンする前に最初のファイル関連付けをクローズしていれば、TITLE オプションを使用して、複数の PL/I ファイルを同じ外部データ・セットに関連付けることができます。以下の例で、INVNTRY は 2 つのファイルを関連付けるデータ・セットを定義する DD ステートメントの名前です。

```
OPEN FILE (FILE1) TITLE('INVNTRY');  
.....  
CLOSE FILE (FILE1);  
.....  
OPEN FILE (FILE2) TITLE('INVNTRY');
```

2 番目のファイルのオープンが行われる前に最初のファイルがクローズされていないと、subcode1 値が 59 の UNDEFINEDFILE 条件が発生し、既にオープンしているファイルに対してオープンが試みられたことを示します。

複数のデータ・セットと 1 つのファイルの関連付け

ファイル名は、時点が異なれば、まったく別のデータ・セットを表すことができます。前述の OPEN ステートメントの例では、ファイル DETAIL1 が DD ステートメント DETAIL1 の DSNNAME パラメーターで指定されているデータ・セットと関連付けられます。このファイルをクローズしてもう一度オープンした場合、TITLE オプションに別の dd 名を指定し、そのファイルを別のデータ・セットに関連付けることが可能です。

つまり、TITLE オプションを使用すると、ファイル・オープン時に、複数のデータ・セットの中から任意の 1 つのデータ・セットを動的に選択して、特定のファイル名に関連付けることができます。次の例を考えてみてください。

```
DO IDENT='A','B','C';
  OPEN FILE(MASTER)
    TITLE('MASTER1' || IDENT);
  .
  .
  .
  CLOSE FILE(MASTER);
END;
```

この例では、DO グループの最初の反復処理時に MASTER がオープンされるときに、関連 dd 名は MASTER1A になります。処理が終了すると、そのファイルはクローズされ、ファイル名と dd 名との関連付けも解除されます。DO グループの 2 回目の反復処理時に、MASTER がもう一度オープンされます。このとき、MASTER は dd 名 MASTER1B と関連付けられます。同様に、DO グループの最後の反復処理時では、MASTER が dd 名 MASTER1C に関連付けられます。

複数のデータ・セットの連結

入力の場合にだけ、連結データ・セットを記述する最初の DD ステートメントだけに dd 名を指定し、その他の DD ステートメントで dd 名を省略すれば、複数の順次データ・セットや領域データ・セットを連結する (すなわち、複数のデータ・セットをリンクして、1 つの連続するデータ・セットとして処理されるようにする) ことができます。例えば、次の DD ステートメントを定義すると、そのステートメントが出てくるジョブ・ステップの実行の間中、データ・セット LIST1、LIST2、LIST3 は 1 つのデータ・セットとして処理されます。

```
//GO.LIST DD DSN=LIST1,DISP=OLD
//          DD DSN=LIST2,DISP=OLD
//          DD DSN=LIST3,DISP=OLD
```

PL/I プログラムからの読み込みの場合は、連結データ・セットは同じボリューム上にある必要はありません。連結データ・セットを逆方向に処理することはできません。

z/OS での HFS ファイルへのアクセス

バッチ・プログラムから HFS ファイルにアクセスするには、DD ステートメント内、または OPEN ステートメントの TITLE オプション内に HFS ファイル名を指定します。

例えば、DD HFS を介して HFS ファイル /u/USER/sample.txt にアクセスするには、次のように DD ステートメントをコーディングします。

```
//HFS DD PATH='/u/USER/sample.txt',PATHOPTS=ORDONLY,DSNTYPE=HFS
```

OPEN ステートメントの TITLE オプションを使用して同じファイルにアクセスするには、次のようにコーディングします。

```
OPEN FILE(HFS) TITLE('/u/USER/sample.txt');
```

TITLE オプションにある 2 つのスラッシュに注意してください。最初のスラッシュはファイル名 (DD 名でなく) が後に続くことを示し、2 番目は完全修飾 HFS ファイル名の先頭を示します (また、バッチ環境で HFS ファイルを参照する場合、ファイル指定を完結するのに使用できる「現行ディレクトリー」がないので、完全修飾名を使用する必要があります)。

バッチ環境では、PL/I は HFS ファイルの扱いを以下の順序で決定します。

- ファイル宣言で ENV(F) が指定されていたら、固定長レコードで構成されるファイルと見なします。
- ファイル宣言で ENV(V) が指定されていたら、If 区切りレコードで構成されるファイルと見なします。
- ファイルの DD ステートメントで FILEDATA=BINARY が指定されていたら、固定長レコードで構成されるファイルと見なします。
- その他の場合は、If 区切りレコードで構成されるファイルと見なします。

z/OS UNIX でのデータ・セットとファイルの関連付け

PL/I プログラム内で使用されるファイルには、PL/I ファイル名が付きます。また、データ・セットには、オペレーティング・システムにより識別される名前が付きます。

PL/I には、使用しているプログラムの PL/I ファイルが参照するデータ・セットを認識する手段が必要なので、使用するデータ・セットには識別名を設定する必要があります。識別名が設定されていないデータ・セットには、PL/I プログラムによってデフォルトの識別名が設定されます。

環境変数、または OPEN ステートメントの TITLE オプションを使用することにより、明示的にデータ・セットを識別することができます。

環境変数の使用

export コマンドを使用すると、PL/I ファイルに関連付けるデータ・セットを識別する環境変数を設定することができます。また、任意でデータ・セットの特性を指定することもできます。環境変数から得られる情報を、データ定義 (または、DD) 情報と呼びます。

環境変数名の形は DD_DDNAME です。この場合、DDNAME は、PL/I ファイル定数 (または、代替 DDNAME (後述)) の名前です。ファイル名が HFS ファイルを参照する場合は、ファイル名を適切に修飾する必要があります。そうしないと PL/I ライブラリーは、MVS データ・セットを参照するファイル名と見なします。

次に例を示します。

```
declare MyFile stream output;  
export DD_MYFILE=/datapath/mydata.dat
```

`/datapath/mydata.dat` は HFS ファイルを参照します。ファイル名は完全修飾されています。

```
export DD_MYFILE=./mydata.dat
```

`./mydata.dat` は、現行ディレクトリーにある HFS ファイルを参照します。

```
export DD_MYFILE=mydata.dat
```

`mydata.dat` は MVS データ・セットを参照します。

IBM のメインフレーム環境に詳しい方は、この環境変数は以下のものと同様と考えることができます。

z/OS での DD ステートメント

TSO の ALLOCATE ステートメント

DD_DDNAME 環境変数を併用した構文およびオプションの詳細については、194 ページの『DD_DDNAME 環境変数の使用による特性の指定』を参照してください。

z/OS UNIX 環境では、バッチ環境よりも多くの可変長 HFS ファイル・タイプがサポートされます。この環境では、PL/I は HFS ファイルの扱いを以下のように決定します。

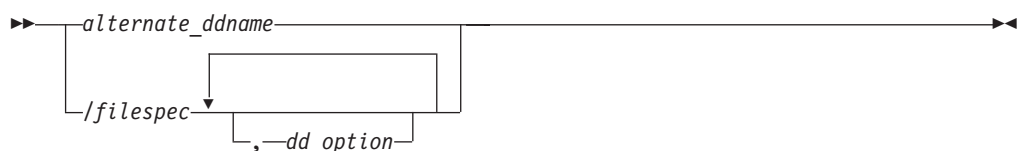
- ファイル宣言で ENV(F) が指定されていたら、固定長レコードで構成されるファイルと見なします。
- ファイルの EXPORT ステートメントで TYPE が指定されていたら、そのタイプのレコードで構成されるファイルと見なします。
- その他の場合は、If 区切りレコードで構成されるファイルと見なします。

OPEN ステートメントの TITLE オプションの使用

OPEN ステートメントの TITLE オプションを使用すると、PL/I ファイルに関連付けるデータ・セットを識別することができます。また、オプションでデータ・セットの特性も設定することができます。

▶▶—TITLE—(*expression*)——▶▶

expression は、次の構文をとる文字ストリングを与えられなければなりません。



alternate_ddname

代替 DD_DDNAME 環境変数の名前。代替 DD_DDNAME 環境変数には、ファイル定数と同じ名前を指定することはできません。例えば、プログラムに INVENTORY という名前のファイルがあり、最初の名前が INVENTORY、2 番目の名前が PARTS という 2 つの DD_DDNAME 環境変数を設定した場合、次のステートメントを使って、INVENTORY ファイルを 2 番目の環境変数に関連付けることができます。

```
open file(Inventory) title('PARTS');
```

filespec

使用しているシステムでの任意の有効なファイル指定。filespec の最大長は、1023 文字です。

dd_option

DD_DDNAME 環境変数で許可されている 1 つ以上のオプション。

DD_DDNAME 環境変数のオプションの詳細については、194 ページの『DD_DDNAME 環境変数の使用による特性の指定』を参照してください。

次に、z/OS DSN を指定して上記の方法で OPEN ステートメントを使用している例を示します。

```
open file(Payroll) title('/June.Dat,append(n),reclsize(52)');
```

TITLE オプションにある必須の先行スラッシュに注意してください。この先行スラッシュは、ファイル名 (DD 名ではなく) が後に続くことを示しています。この場合、June.Dat は MVS データ・セットを参照します。

June.Dat が HFS ファイルである場合、この例は次のようになります。

```
open file(Payroll) title('/u/USER/June.Dat,append(n),reclsize(52)');
```

TITLE オプションにある 2 つのスラッシュに注意してください。最初のスラッシュはファイル名 (DD 名でなく) が後に続くことを示し、2 番目は完全修飾 HFS ファイル名の先頭を示します。

完全修飾名の代わりに、相対 HFS ファイル名を指定することもできます。次に例を示します。

```
open file(Payroll) title('./June.Dat,append(n),reclsize(52)');
```

データ・セット名 *June.Dat* の接頭部に、現行 z/OS UNIX ディレクトリーのパス名が付けられることになります。

この形式の場合、PL/I プログラムはすべての DD 情報を TITLE 式またはファイル宣言の ENVIRONMENT 属性から入手します (DD_DDNAME 環境変数は参照されません)。

データ・セットに関連付けられていないファイルの使用の試み

データ・セットに関連付けられていないファイルを (OPEN ステートメントの TITLE オプションを使って、あるいは、DD_DDNAME 環境変数を設定して) 使用しようとする、UNDEFINEDFILE 条件が発生します。SYSIN ファイルと SYSPRINT ファイルだけは例外です。つまり、この 2 つのファイルはデフォルトでそれぞれ stdin および stdout になります。

PL/I によるデータ・セットの検索方法

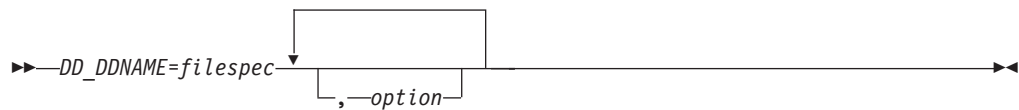
PL/I により、新規データ・セットを作成するためのパス、または既存データ・セットへアクセスするためのパスが、次のいずれかの方法で設定されます。

- 現行ディレクトリー
- export DD_DDNAME 環境変数により定義されているパス

DD_DDNAME 環境変数の使用による特性の指定

export コマンドを使用すると、PL/I ファイルに関連付けられるデータ・セットを識別する環境変数を設定することができます。また、オプションでデータ・セットの特性も指定することができます。環境変数から得られるこのような情報を、データ定義 (または、dd) 情報と呼びます。

DD_DDNAME 環境変数の構文は、次のとおりです。



この構文ではブランクを使用することができます。また、このステートメントの構文は、コマンド入力時にはチェックされません。データ・セットのオープン時に、このステートメントの構文が検証されます。構文に誤りがあれば、オン・コード 96 により UNDEFINEDFILE 条件が発生します。

DD_DDNAME

環境変数の名前を指定します。DDNAME は、大文字でなければなりません。さらに、OPEN ステートメントの TITLE オプションに指定したファイル定数の名前または代替 DDNAME のいずれかになります。TITLE オプションの詳細は、192 ページの『OPEN ステートメントの TITLE オプションの使用』を参照してください。

代替 DDNAME を使用する場合に、その長さが 31 文字を超えるときは、最初の 31 文字だけが環境変数名に指定されます。

filespec

PL/I ファイルに関連付けるファイルまたは装置名を指定します。

option

DD 情報として指定するオプションです。

DD 情報として指定できるオプションは、『APPEND』から 199 ページの『TYPE』で説明します。

APPEND

APPEND オプションにより、既存データ・セットが拡張されるのか、再作成されるのか指定されます。



- Y** 新規レコードを、順次データ・セットの終わりに追加する、あるいは相対データ・セットまたは索引付きデータ・セットに挿入することを指定します。
- N** ファイルが存在する場合、そのファイルを再作成することを指定します。

APPEND オプションを適用できるのは、OUTPUT ファイルだけです。したがって、次の場合、APPEND オプションは無視されます。

- 指定ファイルが存在しない場合
- 指定ファイルに OUTPUT 属性がない場合
- 指定ファイルの編成が REGIONAL(1) の場合

BUFSIZE

BUFSIZE オプションは、バッファのバイト数を指定します。

▶▶—BUFSIZE—(n)—▶▶

RECORD 出力はデフォルト設定でバッファに入り、BUFSIZE のデフォルト値 64k です。STREAM 出力もバッファに入りますが、デフォルトによるものではありません。また、この場合、BUFSIZE のデフォルト値はゼロです。

BUFSIZE の値にゼロが指定されている場合は、バッファのバイト数は、RECSIZE オプションまたは LRECL オプションで指定されている値と同じです。

BUFSIZE オプションが有効なのは、連続バイナリー・ファイルだけです。ファイルが端末入力に使用されている場合は、効率を上げるために、BUFSIZE に値をゼロを割り当てるべきです。

レコード入出力用の CHARSET

CHARSET オプションのこのバージョンは、レコード入出力を使用する連続ファイルにだけ適用されます。このオプションにより、ユーザーは ASCII データ・ファイルを入力ファイルとして使用したり、出力ファイルの文字セットを指定することができます。

▶▶—CHARSET—(—

ASIS
EBCDIC
ASCII

—)—▶▶

入力ファイルの形式、または出力ファイルにとらせたい形式に基づいて、CHARSET のサブオプションを選択してください。

ストリーム入出力用の CHARSET

CHARSET オプションのこのバージョンは、ストリーム入力ファイルおよびストリーム出力ファイルにだけ適用されます。このオプションにより、ユーザーは ASCII データ・ファイルを入力ファイルとして使用したり、出力ファイルの文字セットを指定することができます。ストリーム入出力を使用しているときに ASIS を指定しようとしても、エラーは出されず、文字セットは EBCDIC として扱われます。

▶▶—CHARSET—(—

EBCDIC
ASCII

—)—▶▶

入力ファイルの形式、または出力ファイルにとらせたい形式に基づいて、CHARSET のサブオプションを選択してください。

DELAY

DELAY オプションは、システムがファイル・ロックやレコード・ロックを入手できない場合に失敗した操作を再試行するまでの遅延の時間をミリ秒単位で指定します。

►►—DELAY—($\overbrace{\hspace{1cm}}^0_n$)—►►

このオプションを適用できるのは、VSAM ファイルだけです。

DELIMIT

DELIMIT オプションは、入力ファイルにフィールド区切り文字が含まれているかどうかを指定します。フィールド区切り文字は、レコードのフィールドを分離するブランクかまたはユーザー定義文字です。このオプションを適用できるのは、ソート入力ファイルだけです。

►►—DELIMIT—($\overbrace{\hspace{1cm}}^N_Y$)—►►

ソート・ユーティリティー・プログラムは、フィールド区切り文字の有無により、テキスト・ファイルとバイナリー・ファイルを区別します。フィールド区切り文字が含まれている入力ファイルは、テキスト・ファイルとして処理され、区切り文字がない入力ファイルはバイナリー・ファイルと見なされます。この情報は、ライブラリーが正しいパラメーターをソート・ユーティリティー・プログラムに渡すために必要です。

LRECL

LRECL オプションは RECSIZE オプションと同じです。

►►—LRECL—(n)—►►

LRECL が指定されておらず LINESIZE 値による暗黙指定もされていない場合 (ただし TYPE(FIXED) ファイルを除く)、デフォルトは 1024 です。

LRMSKIP

LRMSKIP オプションを使用すると、ファイルがオープンされてから最初の SKIP フォーマット項目が実行されるように、1 ページ目の n 行目 (n は PUT ステートメントまたは GET ステートメントの SKIP オプションで指定されている値) で出力が開始されるようにすることができます。

►►—LRMSKIP—($\overbrace{\hspace{1cm}}^N_Y$)—►►

n がゼロまたは 1 の場合は、1 ページ目の 1 行目で出力が開始されます。

PROMPT

PROMPT オプションは、コロンを端末からのストリーム入力のプロンプトとして表示するかどうかを指定します。

▶▶ PROMPT—()—▶▶

PUTPAGE

PUTPAGE オプションは、用紙送り文字の後ろに復帰文字を入れるかどうかを指定します。このオプションが適用されるのは、プリンター向けファイルだけです。プリンター向けファイルは、PRINT 属性を指定して宣言されたストリーム出力ファイル、あるいは、CTLASA 環境オプションを指定して宣言されたレコード出力ファイルです。

▶▶ PUTPAGE—()—▶▶

NOCR

用紙送り文字 ('0C'x) の後ろに復帰文字 ('0D'x) を入れないことを指します。

CR

用紙送り文字の後ろに復帰文字を追加することを指します。このオプションは、出力が IBM 以外のプリンターに送られる場合に指定する必要があります。

RECCOUNT

RECCOUNT オプションは、PL/I ファイルのオープン・プロセス中に作成される、相対データ・セットまたは領域データ・セットにロードできる最大のレコード数を指定します。

▶▶ RECCOUNT—(n)—▶▶

PL/I がデータ・セットを作成も再作成もしない場合、RECCOUNT オプションは無視されます。

RECCOUNT オプションのデフォルトは 50 です。

注: z/OS の場合、REGIONAL(1) データ・セットの機能性とパフォーマンスを向上させるには、/filespec パラメーターと RECCOUNT パラメーターの両方を指定した TITLE オプションを省略することをお勧めします。この場合、ファイルにロードされるレコードの数は、データ・セットの最初のエクステンツに割り振られたスペースによって決まります。詳しくは、257 ページの『第 9 章 領域データ・セットの定義と使用』を参照してください。

RECSIZE

RECSIZE オプションは、データ・セット内のレコードの長さ *n* を指定します。

▶▶ RECSIZE—()—▶▶

領域データ・セットおよび固定長データ・セットの場合は、RECSIZE はデータ・セットの各レコードの長さを指定します。その他のデータ・セット・タイプの場合は、RECSIZE はレコードがとれる最大の長さを指定します。

SAMELINE

SAMELINE オプションは、入力を求めるプロンプトのステートメントと同じ行で、システム・プロンプトを行わせるかどうかを指定します。

▶▶—SAMELINE—()—▶▶

以下の例は、PROMPT オプションと SAMELINE オプションのいくつかの組み合わせの結果を示しています。

例 1

PUT SKIP LIST('ENTER:'); というステートメントが与えられると、その出力結果は次のようになります。

```
prompt(y), sameline(y)prompt(n),          ENTER: (cursor)ENTER: (cursor)ENTER:
sameline(y)prompt(y), sameline(n) prompt(n), (cursor)ENTER:(cursor)
sameline(n)
```

例 2

PUT SKIP LIST('ENTER'); というステートメントが与えられると、その出力結果は次のようになります。

```
prompt(y), sameline(y)prompt(n),          ENTER: (cursor)ENTER (cursor)ENTER:
sameline(y)prompt(y), sameline(n) prompt(n), (cursor)ENTER(cursor)
sameline(n)
```

SKIP0

SKIP0 オプションは、ソース・プログラムに SKIP(0) ステートメントがコーディングされた場合、ライン・カーソルをどこに移動するかを指定します。SKIP0 オプションは、PM アプリケーションとしてリンクされていない端末ファイルに適用されます。

▶▶—SKIP0—()—▶▶

SKIP0(N)

カーソルを次の行の先頭に移動することを指定します。

SKIP0(Y)

カーソルを現在行の先頭に移動することを指定します。

次の例は、現在の出力行の先頭にカーソルが移動するように、出力を端末スキップ・ゼロ行で行う方法を示しています。

```
export DD_SYSPRINT='stdout:.,SKIP0(Y)'
```

TYPE

TYPE オプションは、ネイティブ・ファイル内のレコードのフォーマットを指定します。



CRLF

レコードを文字の組み合わせ CR - LF で区切ることを指定します。('CR' と 'LF' は、それぞれ復帰と改行の ASCII 値である '0D'x と '0A'x を表します。) 出力ファイルの場合、PL/I は各レコードの終わりにこれらの文字を挿入します。入力ファイルの場合、PL/I はこれらの文字を破棄します。入力、出力のいずれの場合も、これらの文字は RECSIZE の考慮事項には入りません。

データ・セットのレコード長として決められている値よりも長いレコードをデータ・セットに入れることはできません。

LF

レコードが LF 文字組み合わせで区切られることを指定します。('LF' は、ASCII コードの用紙送り、つまり '0A'x を表します。) 出力ファイルの場合、PL/I は各レコードの終わりにこれらの文字を挿入します。入力ファイルの場合、PL/I はこれらの文字を破棄します。入力、出力のいずれの場合も、これらの文字は RECSIZE の考慮事項には入りません。

データ・セットのレコード長として決められている値よりも長いレコードをデータ・セットに入れることはできません。

TEXT

前述の LF と同じです。

FIXED

データ・セット内の各レコードの長さが同じであることを指定します。データ・セット内のレコード長として指定されている値は、レコードの境界を認識する場合に使用されます。

TYPE(FIXED) ファイル内のすべての文字は、制御文字も含め (ある場合)、データと見なされます。指定したレコード長が、存在している文字を反映していること、あるいは、指定したレコード長がレコード内の全文字を扱えることを確認してください。

CRLFEOF

出力ファイルを除けば、このサブオプションは CRLF オプションと同じ情報を指定します。ファイルの 1 つが出力についてクローズされるときに、ファイルの終わりマーカが最後のレコードに追加されます。

U レコードが不定形式であることを表します。これらの不定形式ファイルは、OPEN および CLOSE を除いて、どのレコード入出力またはストリーム入出力のステートメントでも使用できません。TYPE(U) ファイルからの読み取りは、

FILEREAD 組み込み関数を使用することのみにより行うことができます。また、TYPE(U) ファイルへの書き込みは、FILEWRITE 組み込み関数を使用することのみにより行うことができます。

ASA(N) オプションを指定したプリンター向けファイルの場合このオプションが無視されるということを除き、TYPE オプションを適用できるのは CONSECUTIVE ファイルだけです。

使用しているプログラムが TYPE(FIXED) が有効である既存のデータ・セットにアクセスしようとしており、かつそのデータ・セット長がユーザーが指定した複数の論理レコード長の倍数でない場合は、PL/I は UNDEFINEDFILE 条件を発生させます。

TYPE(FIXED) 属性を指定した非印刷ファイルを使用している場合は、SKIP が行の終わりまで末尾ブランクに置き換えられます。TYPE(LF) が使用されている場合、SKIP は末尾ブランクなしに、LF で置き換えられます。

データ・セット特性の設定

データ・セットは、オペレーティング・システムのデータ管理ルーチンが理解できる特定のフォーマットで保管されているレコードから構成されています。ユーザーのプログラムでファイルの宣言またはオープンを行うときに、ユーザーはそのファイルに入っているレコードの特性を PL/I およびオペレーティング・システムに記述します。また、JCL または OPEN ステートメントの TITLE オプションの式を使って、データ・セット内またはデータ・セットに関連付けられている PL/I ファイル内のデータ特性をオペレーティング・システムに記述することもできます。

必ずしも、プログラムの内と外の両方で、自分のデータを記述する必要はありません。多くの場合、1 回の記述が、データ・セットとその関連 PL/I ファイルの両方に役立ちます。實際上、データの特性は一か所だけに記述した方が有利です。このことについては、本章および後続の章で説明します。

使用しているプログラム・データおよびデータ・セットを効率よく記述するには、オペレーティング・システムによるデータの移動および保管の方法をある程度理解する必要があります。

ブロックおよびレコード

データ・セット内のデータ項目は、ブロック間ギャップ (IBG) で区切られているブロックに配置されます。(これをレコード間ギャップと呼んでいるマニュアルもあります。)

ブロック は、データ・セットに送られてくる、あるいはデータ・セットから送り出されるデータの単位です。各ブロックには、1 つのレコード、レコードの一部分、または複数のレコードが入っています。ブロック・サイズは、DD ステートメントの BLKSIZE パラメーター内、あるいは ENVIRONMENT 属性の BLKSIZE オプション内で指定することができます。

レコード は、プログラムに送られてくる、またはプログラムから送り出されるデータの単位です。レコード長は、DD ステートメントの LRECL パラメーター内、

OPEN ステートメントの TITLE オプション内、または ENVIRONMENT 属性の RECSIZE オプション内で指定することができます。

PL/I プログラムを作成する場合は、読み取りあるいは書き込みを行うレコードだけを考慮すれば済みます。しかし、自分のプログラムが作成あるいはアクセスするデータ・セットを記述する場合は、ブロックおよびレコード間の関係を知っている必要があります。

ブロック化によって、磁気ストレージ・ボリューム内のストレージ・スペースを節約することができます。ブロック化が、ブロック間ギャップ数を減らし、データ・セットを処理するのに必要な入出力操作回数を減らすことにより効率を上げるためです。また、レコードは、データ管理ルーチンにより、ブロック化およびブロック化解除されます。

情報交換コード

データが記録される通常のコードは拡張 2 進化 10 進コード (EBCDIC) です。

ASCII コードの各文字は 7 ビット・パターンで表されるので、このようなパターンが 128 通りあります。また、ASCII セットには、有効な ASCII コードがない EBCDIC 文字を表すために使用される置換文字 (SUB 制御文字) があります。ASCII 置換文字は、00111111 というビット・パターンを持つ EBCDIC SUB 文字に変換されます。

レコード・フォーマット

データ・セット内のレコードは、次のいずれかのフォーマットを持っています。

- 固定長
- 可変長
- 不定長

レコードは、必要に応じて、ブロック化することができます。オペレーティング・システムは、固定長レコードと可変長レコードを非ブロック化しますが、不定長レコードを非ブロック化するには、ユーザーのプログラム内にコードを提供する必要があります。

レコード・フォーマットは、DD ステートメントの RECFM パラメーター内、OPEN ステートメントの TITLE オプション内、または ENVIRONMENT 属性のオプションとして指定します。

固定長レコード

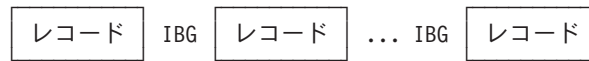
固定長レコードには、次に挙げるフォーマットを指定することができます。

- F 固定長、非ブロック化
- FB 固定長、ブロック化
- FS 固定長、非ブロック化、標準
- FBS 固定長、ブロック化、標準

固定長レコードを持っているデータ・セットの場合は、202 ページの図 18 に示すように、すべてのレコードの長さが等しくなります。レコードがブロック化されると、通常、各ブロックには同じ数の固定長レコードが入っています (ただし、ブロックは切り捨てられる場合もあります)。レコードがブロック化されていない場合

は、各レコードがブロックを構成します。

非ブロック化レコード (F フォーマット):



ブロック化レコード (FB フォーマット):

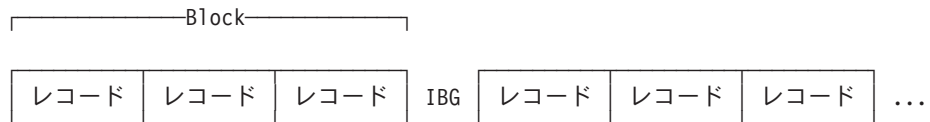


図 18. 固定長レコード

オペレーティング・システムでは、一定のレコード長に基づいてブロック化、非ブロック化が行われるため、可変長レコードより固定長レコードの方が速く処理されます。

可変長レコード

指定できる可変長レコードのフォーマットは、次のとおりです。

- V 可変長、非ブロック化
- VB 可変長、ブロック化
- VS 可変長、非ブロック化、スパン
- VBS 可変長、ブロック化、スパン

V フォーマットでは、可変長レコードと可変長ブロックの両方を使用することができます。各レコードの 4 バイトの接頭部と、各ブロックの最初の 4 バイトには、オペレーティング・システムが使用する場合の制御情報が入ります (レコードまたはブロックのバイト単位の長さを含む)。このような制御フィールドのため、可変長レコードは逆方向に読み込むことができません。

V フォーマットは、非ブロック化可変長レコードを表します。各レコードは、レコードが 1 つだけを持つブロックとして扱われます。ブロックの最初の 4 バイトにはブロック制御情報が入り、次の 4 バイトにはレコード制御情報が入ります。

VB フォーマットは、ブロック化可変長レコードを表します。各ブロックには、そのブロックに収容可能なレコード数と同じ数のレコードが入ります。ブロックの最初の 4 バイトにはブロック制御情報が入り、各レコードの 4 バイト接頭部にはレコード制御情報が入ります。

スパン・レコード: スパン・レコードは可変長レコードで、レコードの長さがブロック・サイズを超えることもできます。超えた場合は、レコード・フォーマットを VS または VBS のいずれかで指定して、レコードを複数のセグメントに分割し、2 つ以上の連続ブロックに置きます。セグメンテーションおよび組み立ては、オペレーティング・システムによって処理されます。スパン・レコードを使用すると、レコードの長さを問わず、ブロック・サイズが選択でき、補助記憶域を最大限に活用し、伝送の効果を最大限に高めます。

VS フォーマットは、V フォーマットに似ています。それぞれのブロックは、1 つのレコードまたはレコードのセグメントのみを含みます。ブロックの最初の 4 バイトにはブロック制御情報が入り、次の 4 バイトにはレコードまたはセグメント制御情報 (レコードが単体のものか、または最初、中間、最後のセグメントであるかということを示す情報を含む) が入ります。

VBS フォーマットでは、各ブロックが、単体のレコードまたはセグメントをできるだけ多く保持することができるという点において、VS フォーマットと異なっています。したがって、各ブロックのサイズがほとんど同じになります (ただし、各セグメントは最低 1 バイトのデータを含まなければならないため、最大で 4 バイトの変化幅があります)。

不定長レコード

U フォーマットでは、F フォーマットにも V フォーマットにも当てはまらないレコードを処理することができます。オペレーティング・システムおよびコンパイラは各ブロックをレコードとして扱います。ユーザー・プログラムで必要に応じてブロック化あるいは非ブロック化を行わなくてはなりません。

データ・セットの編成

オペレーティング・システムのデータ管理ルーチンは、データ・セット内でデータが保管される方法およびデータへのアクセスに使用できる方法により異なるいくつかの種類のデータ・セットを扱うことができます。VSAM 以外のデータ・セットの主な 3 つのタイプおよび、それらに該当する PL/I 編成¹ を記述するキーワードは、次のとおりです。

データ・セットの PL/I 編成

タイプ

順次	CONSECUTIVE または ORGANIZATION (連続)
索引	INDEXED または ORGANIZATION (索引)
直接	REGIONAL または ORGANIZATION (相対)

データ・セットの 4 つ目のタイプ 区分 には、対応する PL/I 編成はありません。

また、PL/I は 3 つのタイプの VSAM データ編成、*ESDS*、*KSDS*、*RRDS* もサポートしています。VSAM データ・セットの詳細については、271 ページの『第 10 章 VSAM データ・セットの定義と使用』を参照してください。

順次 (つまり CONSECUTIVE) データ・セット内では、各レコードは物理的順序で設定されます。あるレコードが与えられた場合、そのレコードの次のレコード位置は、そのレコードが物理的にデータ・セットのどこにあるかによって決まります。直接アクセス装置に対して順次編成を選択することができます。

索引順次 (または INDEXED) データ・セットは、直接アクセス・ボリューム上になければなりません。オペレーティング・システムが維持する索引や索引のセットにより、特定の基本レコードの位置が与えられます。これによって、順次処理だけでなく、レコードの直接検索、置換、追加、および削除を行うことができます。

1. 『順次』 および 『直接』 という用語を、PL/I ファイル属性 SEQUENTIAL および DIRECT と混同しないでください。この属性は、ファイルの処理方法を指すための属性であり、対応するデータ・セットの編成方法を指す属性ではありません。

直接 (または REGIONAL) データ・セットは、直接アクセス・ボリューム上になければなりません。データ・セットは複数の領域に分割され、各領域には 1 つ以上のレコードが入っています。領域番号を指定するキーを使えば、任意のレコードに直接アクセスすることができます。順次処理を行うことも可能です。

区分 データ・セットでは、順次編成されるデータから成り、それぞれがメンバーと呼ばれる独立したグループは、直接アクセス・データ・セット内に存在します。このタイプのデータ・セットには、各メンバーの位置をリストするディレクトリーが 1 つ含まれています。区分データ・セットはしばしば ライブラリー と呼ばれます。このコンパイラーには、区分データ・セットを作成したり、区分データ・セットにアクセスするための特殊機能はありません。各メンバーは、CONSECUTIVE データ・セットとして PL/I プログラムが処理することができます。区分データ・セットをライブラリーとして使用する方法は、221 ページの『第 7 章 ライブラリーの使用』で説明します。

ラベル

オペレーティング・システムは内部ラベルを使って、直接アクセス・ボリュームを識別したり、データ・セット属性 (例えば、レコード長やブロック・サイズ) を保管します。これらの属性情報は、最初は DD ステートメントまたはユーザー・プログラムから入手する必要があります。

IBM 標準ラベルには 2 つの部分、すなわち、初期ボリューム・ラベルとヘッダー・ラベルがあります。初期ボリューム・ラベルは、特定のボリュームとその所有者を識別します。一方、ヘッダー・ラベルはボリューム上の各データ・セットの前後にあります。ヘッダー・ラベルには、システム情報、装置依存情報 (例えば、記録手法)、およびデータ・セット特性が入っています。

直接アクセス・ボリュームには、IBM 標準ラベルが付きます。ボリューム・ラベルで各ボリュームが識別されます。このラベルには、ボリューム通し番号とボリューム目録 (VTOC) のアドレスが入っています。この目録には、ボリュームに保管されているデータ・セットごとのデータ・セット制御ブロック (DSCB) と呼ばれるラベルが入っています。

データ定義 (DD) ステートメント

データ定義 (DD) ステートメントは、オペレーティング・システムに対してデータ・セットを定義するためのジョブ制御ステートメントであり、入出力リソースの割り振りをオペレーティング・システムに要求するものです。データ・セットを動的に割り振らない場合は、各ジョブ・ステップに、そのジョブ・ステップで処理するデータ・セットの DD ステートメントをすべて組み込んでおく必要があります。

ジョブ制御ステートメントの構文については、「*z/OS JCL User's Guide*」を参照してください。DD ステートメントのオペランド・フィールドには、データ・セットの位置 (例えば、ボリューム通し番号や、ボリュームをマウントする装置の識別名) を記述したキーワード・パラメーター、およびデータそのものの属性 (例えば、レコード・フォーマット) を記述したキーワード・パラメーターを入れることができます。

DD ステートメントを使用すると、使用するデータ・セットおよび入出力装置から独立した PL/I ソース・プログラムを作成することができます。また、ユーザー・プ

ログラムを再コンパイルしなくても、データ・セットのパラメーターを変更したり、さまざまなデータ・セットを処理したりすることができます。

次の項に、DD ステートメントのオペランドと、ユーザーの PL/I プログラムとの関係について説明します。

ENVIRONMENT 属性の LEAVE および REREAD オプションを使用すると、磁気テープ・ボリュームの終わりに達したとき、または磁気テープのデータ・セットが閉じたときにとるアクションを制御する DISP パラメーターを使用できます。

LEAVE および REREAD オプションの説明は、第 8 章の『LEAVE/REREAD』にあります。

PL/I バージョン 1 の標準機能であった書き込み妥当性検査は、本バージョンでは実行されません。書き込み妥当性検査を実行するには、JCL DD ステートメントの DCB パラメーターの OPTCD サブパラメーターを使って要求できます。詳細については、「*OS/VS2 Job Control Language*」を参照してください。

条件付きサブパラメーターの用法

PL/I プログラムにより処理されるデータ・セットの DISP パラメーターの条件付きサブパラメーターを使用する場合は、ステップ異常終了機能を使う必要があります。ステップ異常終了機能は、次のようなにして入手します。

1. アプリケーションの条件付きサブパラメーターを適用する必要がある障害が発生したためにプログラムの実行を終了する場合は常に、ERROR 条件を発生させるか、または ERROR 信号を出す。
2. PL/I のユーザー出口を ABEND 要求に変更する必要がある。

データ・セット特性

DD ステートメントの DCB (データ制御ブロック) パラメーターを使用すると、データ・セット内のデータの特性を記述したり、実行時のデータの処理方法を記述することができます。DD ステートメントの他のパラメーターは、主としてデータ・セットの識別、位置、処置を扱うのに対し、DCB パラメーターはレコード自体の処理に必要な情報を指定します。DCB パラメーターのサブパラメーターの詳細については、「*z/OS JCL User's Guide*」を参照してください。

DCB パラメーターには、次の事項を記述するサブパラメーターが入っています。

- データ・セットの編成とそのアクセス方法 (サブパラメーター CYLOFL、DSORG、LIMCT、NTM、および OPTCD)
- プリンターの行送りなどの装置依存情報 (サブパラメーター CODE、FUNC、MODE、OPTCD=J、PRTSP、STACK、および STACK)
- レコード・フォーマット (サブパラメーター BLKSIZE、KEYLEN、LRECL、および RECFM)
- 各レコードの最初のバイト (RECFM サブパラメーター) に挿入される ASA 制御文字 (ある場合)

BLKSIZE、LRECL、KEYLEN、および RECFM (またはそれらと同等のもの) は、DCB パラメーターではなく、ユーザーの PL/I プログラムのファイル宣言の ENVIRONMENT 属性を使って指定することができます。

DCB パラメーターを使って、PL/I プログラム内でデータ・セット用に (宣言されたファイル属性と、その属性で暗黙指定されたその他の属性を使って) 既に設定されている情報を指定変更することはできません。既に提供されている情報を変更しようとする DCB サブパラメーターは、無視されます。

新規データ・セットの場合、DD ステートメントと矛盾していれば、プログラムで定義されたファイルの属性が使用されます。

PDS ファイルをクローズするときに、RC=4 を伴うメッセージ IEC225I が出される場合があります。このメッセージは安全であり、無視できます。

DCB パラメーターの例

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

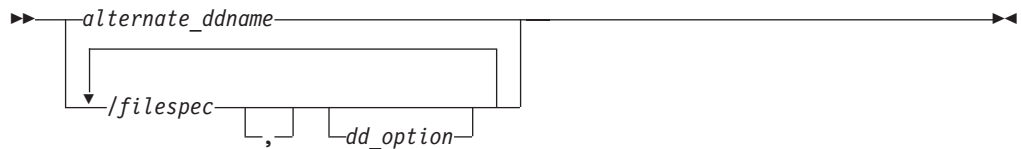
上記の例では、長さ 40 バイトの固定長レコードが、400 バイトの長さのブロックにグループ化されます。

OPEN ステートメントの TITLE オプションの使用

OPEN ステートメントの TITLE オプションを使用すると、PL/I ファイルに関連付けられるデータ・セットを識別することができます。また、オプションで追加のデータ・セット特性も与えることができます。

▶▶—TITLE—(—*expression*—)————▶▶

expression は、次の構文をとる文字ストリングを与えられなければなりません。



alternate_ddname

代替 DD_DDNAME 環境変数の名前。代替 DD_DDNAME 環境変数には、ファイル定数と同じ名前を指定することはできません。例えば、プログラムに INVENTORY という名前のファイルがあり、最初の名前が INVENTORY、2 番目の名前が PARTS という 2 つの DD_DDNAME 環境変数を設定した場合、次のステートメントを使って、INVENTORY ファイルを 2 番目の環境変数に関連付けることができます。

```
open file(Inventry) title('PARTS');
```

filespec

有効な z/OS UNIX ファイル指定または z/OS DSN ファイル指定。

dd_option

DD_DDNAME 環境変数で許可されている 1 つ以上のオプション。

DD_DDNAME 環境変数の詳細については、194 ページの『DD_DDNAME 環境変数の使用による特性の指定』を参照してください。

次に、上記の方法で OPEN ステートメントを使用している例を示します。

```
open file(Payroll) title('/June.Dat, append(n),recsize(52)');
```

上記の形式の場合、PL/I は、すべての DD 情報を TITLE 式から、あるいはファイル宣言の ENVIRONMENT 属性から入手します。DD_DDNAME 環境変数は参照されません。

PL/I ファイルとデータ・セットの関連付け

ファイルのオープン

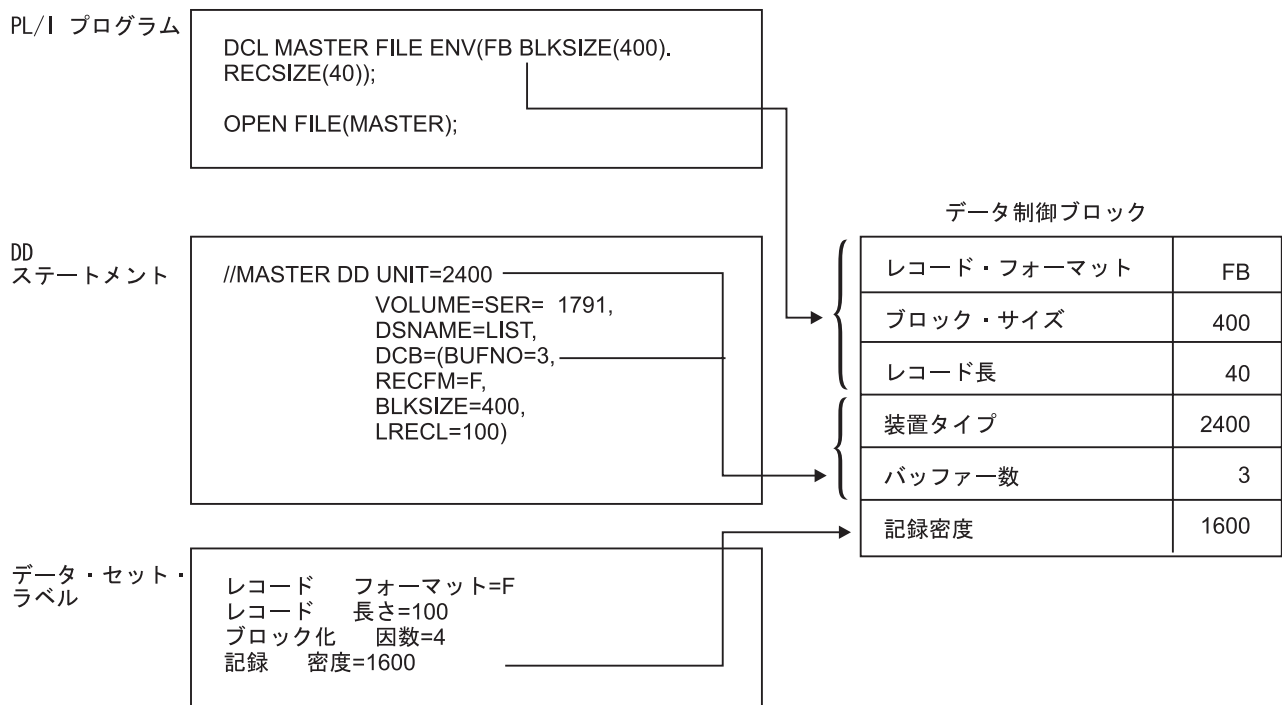
PL/I OPEN ステートメントを実行することにより、ファイルとデータ・セットを関連付けることができます。これを行うには、ファイルを記述している情報とデータ・セットを記述している情報をマージする必要があります。ファイルの属性とデータ・セットの特性の間に矛盾が検出されると、UNDEFINEDFILE 条件が発生します。

PL/I ライブラリーのサブルーチンは、データ・セットの骨組みデータ制御ブロックが作成されます。これらのサブルーチンは、DECLARE ステートメントおよび OPEN ステートメントから得られるファイル属性と、宣言された属性が暗黙設定するあらゆる属性を使って、可能な限りデータ制御ブロックを完成させます。(208 ページの図 19 参照を参照してください。) そのあと、これらのサブルーチンは、OPEN マクロ命令を出します。このマクロ命令は、データ管理ルーチンと呼び出して、正しいボリュームがマウントされているかどうかをチェックし、データ制御ブロックを完成させます。

データ管理ルーチンは、さらに必要な情報がないか調べるためにデータ制御ブロックを検査し、次に、まず DD ステートメント内からその情報を検索し、最後に、データ・セットが存在するかどうか、かつデータ・セット・ラベル内に標準ラベルが付いているかどうかを調べます。新規データ・セットの場合は、データ管理ルーチンは、ラベル (必要な場合) を作成し、そのラベルにデータ制御ブロックからの情報を入れます。

INPUT データ・セットの場合、属性が矛盾しない限り、PL/I プログラムは DCB 属性を指定変更できます。OUTPUT データ・セットの場合、PL/I プログラムは DCB 属性を指定変更できません。ただし、データ・セットのオープン時に欠落 DCB 属性があれば、その属性は PL/I プログラムから取得されます (プログラムでその属性が指定されている場合)。

DCB フィールドがこれらのソースの情報で埋められると、制御は PL/I ライブラリー・サブルーチンに戻ります。依然として値が入力されていないフィールドには、PL/I OPEN サブルーチンがデフォルトの情報を提供します。例えば、LRECL が指定されていない場合は、BLKSIZE に与えられた値が提供されます。



注: PL/I プログラムの情報は、DD ステートメントおよびデータ・セット・ラベルの情報を指定変更します。
DD ステートメントの情報は、データ・セット・ラベルの情報を指定変更します。

図 19. オペレーティング・システムによる DCB への情報の組み込みの方法

ファイルのクローズ: PL/I CLOSE ステートメントを実行することにより、関連付けられていたデータ・セットからファイルが切り離されます。PL/I ライブラリー・サブルーチンは、まず CLOSE マクロ命令を出し、データ管理ルーチンから制御が戻ると、ファイルのオープン時に作成されたデータ制御ブロックを解放します。データ管理ルーチンは、新規データ・セット用のラベルが書き込みを完了し、既存データ・セットのラベルを更新します。

ENVIRONMENT 属性での特性の指定

ENVIRONMENT 属性では、さまざまなオプションを使うことができます。以下に示すように、おのこのタイプのファイルごとにさまざまな属性および環境オプションがあります。

ENVIRONMENT 属性

PL/I のファイル宣言ファイルの ENVIRONMENT 属性を使用して、ファイルに関連付けられているデータ・セットの物理編成についての情報やその他の関連情報を指定することができます。この情報のフォーマットは、括弧で囲んだオプション・リストでなければなりません。

►►—ENVIRONMENT—(—option-list—)——►►

省略形: ENV

ブランクまたはコンマで区切ったオプションを、任意の順序で指定することができます。

次の例は、完全なファイル宣言のコンテキスト中にある ENVIRONMENT 属性の構文を説明しています。(指定されているオプションは VSAM 用のもので、271 ページの『第 10 章 VSAM データ・セットの定義と使用』に説明があります。)

```
DCL FILENAME FILE RECORD SEQUENTIAL
  INPUT ENV(VSAM GENKEY);
```

表 13 は、ENVIRONMENT オプションとファイル属性を要約したものです。使用に関する特定の制限については、表内の注および注釈を参照してください。複数のデータ・セット編成に適用できるオプションは、本章の後半で説明します。また、各オプションについては、オプションが適用される各データ・セット編成とともに後続の章で説明します。

表 13. PL/I ファイル宣言の属性

データ・セット・ タイプ	S t r e a m	レコード									凡例: C VSAM では検査 D デフォルト I 指定または暗黙指定が必要 N VSAM では無視 O オプション S 指定が必要 - 無効	
		順次					直接					
		連続		U n b b e r e d	R e g i o n a l	T e l e p r o c e s s i n g	I n d e x e d	V S A M	R e g i o n a l	I n d e x e d		V S A M
		B u f f e r e d	B u f f e r e d									
ファイル のタイプ	C o n s e c u t i v e	B u f f e r e d	U n b b e r e d	R e g i o n a l	T e l e p r o c e s s i n g	I n d e x e d	V S A M	R e g i o n a l	I n d e x e d	V S A M	暗黙指定属性	
ファイル属性 ¹												

ファイル	I	I	I	I	I	I	I	I	I	I	
入力 ¹	D	D	D	D	D	D	D	D	D	D	ファイル
出力	O	O	O	O	O	O	O	O	O	O	ファイル
環境	I	I	I	S	S	S	S	S	S	S	ファイル
ストリーム	D	-	-	-	-	-	-	-	-	-	ファイル
印刷 ¹	O	-	-	-	-	-	-	-	-	-	ファイル・ストリーム出力
レコード	-	I	I	I	I	I	I	I	I	I	ファイル
更新	-	O	O	O	-	O	O	O	O	O	ファイル・レコード
順次	-	D	D	D	-	D	D	-	-	D	ファイル・レコード
バッファ付き	-	D	-	-	I	D	D	-	-	S	ファイル・レコード
キー付 ²	-	-	-	O	I	O	O	I	I	O	ファイル・レコード
直接	-	-	-	-	-	-	S	S	S	S	キー付きファイル・レコード

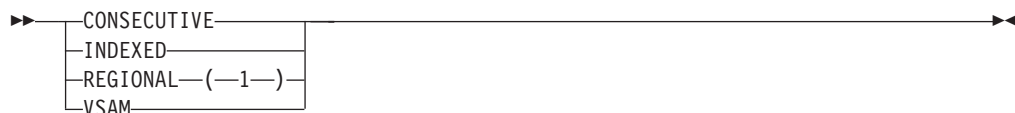
ENVIRONMENT オプション											コメント
FIFBIFSIFBSIVI VBIVSIVBSIIU	I	S	S	-	-	-	N	-	-	N	VS と VBS は ストリームでは無効
FIFBIU	S	S	-	-	-	-	N	-	-	N	ASCII データ・セットのみ
FIVIU	-	-	-	S	-	-	N	S	-	N	REGIONAL(1) の場合は F のみ
FIFBIVIB	-	-	-	-	-	S	N	-	S	N	

RECSIZE(n)	I	I	I	I	S	I	C	I	I	C	RECSIZE か BLKSIZE (または両方) を指定する 必要がある (連続ファイル、 索引付きファイル、 領域ファイルの場合)
BLKSIZE(n)	I	I	I	I	-	I	N	I	I	N	ASCII データ・セットの場合は無効 VSAM ESDS の場合は指定可能
SCALARVARYING	-	O	O	O	-	O	O	O	O	O	ASCII データ・セットの場合は無効
CONSECUTIVE	D	D	D	-	-	-	O	-	-	O	VSAM ESDS の場合は指定可能
LEAVEIREREAD	O	O	O	-	-	-	-	-	-	-	
CTLASAICTL360	-	O	O	-	-	-	-	-	-	-	ASCII データ・セットの場合は無効
GRAPHIC	O	-	-	-	-	-	-	-	-	-	
INDEXED	-	-	-	-	-	S	O	-	S	O	VSAM ESDS の場合は指定可能
KEYLOC(n)	-	-	-	-	-	O	-	-	O	-	
ORGANIZATION	D	-	-	-	-	-	-	-	-	-	
GENKEY	-	-	-	-	-	O	O	-	O	O	INPUT または UPDATE ファイル のみ (KEYED が必須)
REGIONAL(1)	-	-	-	S	-	-	-	S	-	-	
VSAM	-	-	-	-	-	-	S	-	-	S	
BKWD	-	-	-	-	-	-	O	-	-	O	
REUSE	-	-	-	-	-	-	O	-	-	O	OUTPUT ファイルのみ

注:

1. INPUT 属性が指定されているファイルは PRINT 属性をとることはできません。
2. INDEXED 出力および REGIONAL 出力にはキーが必要です。

データ・セット編成オプション: データ・セットの編成を指定するオプションは、次のとおりです。



各オプションについては、適用されるデータ・セット編成の項目で解説されています。

その他の ENVIRONMENT オプション: ブロック・サイズやレコード長など、整数引数が必要な ENVIRONMENT オプションは、定数や変数と併用することができます。変数には、添え字や修飾を付けたりすることはできません。変数には、属性 FIXED BINARY(31,0) および STATIC が必要です。

ENVIRONMENT オプションと、それと同等の DCB パラメーターは、以下のとおりです。

ENVIRONMENT オプション	DCB サブパラメーター
レコード・フォーマット	RECFM ¹
RECSIZE	LRECL
BLKSIZE	BLKSIZE
CTLASAICTL360	RECFM
KEYLENGTH	KEYLEN

注: ¹VS は、DCB 中ではなく、ENVIRONMENT オプションとして指定されなければなりません。

レコード単位データ伝送のレコード・フォーマット

サポートされるレコード・フォーマットは、データ・セット編成により異なります。



レコードのフォーマットは、次のいずれかです。

固定長	F	非ブロック化
	FB	ブロック化
	FS	非ブロック化、標準
	FBS	ブロック化、標準
可変長	V	非ブロック化
	VB	ブロック化
	VS	スパン
	VBS	ブロック化、スパン
不定長	U	(ブロック化できない)

U フォーマット・レコードが可変長ストリングに読み込まれる場合は、PL/I はストリングの長さを検索されたデータのブロック長に設定します。

上記のレコード・フォーマット・オプションは、VSAM データ・セットには適用されません。VSAM データ・セットに関連付けられているファイルにレコード・フォーマット・オプションを指定すると、そのオプションは無視されます。

VS フォーマットのレコードは、連続編成のデータ・セットにのみ指定できます。

ストリーム指向データ伝送のレコード・フォーマット

ストリーム指向データ伝送のレコード・フォーマット・オプションについては、227 ページの『ストリーム指向データ伝送の使用』で説明します。

RECSIZE オプション

RECSIZE オプションは、レコード長を指定します。

>>—RECSIZE—(—record-length—)<-----<

VSAM データ・セットに関連付けられているファイルの場合、**record-length** (レコード長) は、次の値の合計です。

1. データに必要な長さ。可変長レコードおよび不定長レコードの場合は、この長さが最大長になります。
2. 必要な制御バイト。可変長レコードには 4 バイトが必要です (レコード長の接頭部用)。固定長レコードおよび不定長レコードでは必要ありません。

VSAM データ・セットの場合、データ・セットが定義されるときに、レコードの最大長と平均長がアクセス方式サービス・プログラム・ユーティリティに指定され

ます。検査の目的で RECSIZE オプションをファイル宣言に組み込む場合は、レコードの最大長を指定する必要があります。指定した RECSIZE がデータ・セットに定義されている値と矛盾する場合は、UNDEFINEDFILE 条件が発生します。

record-length (レコード長) は、整数または属性 FIXED BINARY(31,0) STATIC を持つ変数を指定することができます。

レコード長の値には、次のような規則があります。

最大長:

固定長、および不定長 (ASCII データ・セットを除く): 32760

UPDATE ファイルを持つ V フォーマット、ならびに VS および VBS フォーマット: 32756

入力ファイルおよび出力ファイルを持つ VS および VBS フォーマット: 16777215

ASCII データ・セット : 9999

VSAM データ・セット: 32761

注: 32,756 バイト以上の VS フォーマットおよび VBS フォーマット・レコードの場合は、ENVIRONMENT の RECSIZE オプションの長さを指定し、DD ステートメントの DCB サブパラメーターに LRECL=X を指定しなければなりません。RECSIZE が INPUT または OUTPUT の最大長を超える場合は、レコード条件が発生するか、またはレコードが切り捨てられます。

ゼロ値:

有効な値の検索が、次のように行われます。

- 最初に、ファイルと関連したデータ・セット用の DD ステートメント内
- 次に、データ・セット・ラベル内

上記のいずれかで値を入手できない場合は、デフォルト・アクションが行われます (214 ページの『レコード・フォーマット、BLKSIZE、および RECSIZE のデフォルト』参照)。

負の値:

UNDEFINEDFILE 条件が発生します。

BLKSIZE オプション

BLKSIZE オプションは、データ・セット上の最大ブロック・サイズを指定します。

►►BLKSIZE—(—*block-size*—)◄◄

block-size (ブロック・サイズ) は、次の値の合計です。

1. 次のいずれかの全長

- 1 つのレコード
- 1 つのレコードと、1 つまたは 2 つのレコード・セグメント
- 複数のレコード
- 複数のレコードと、1 つまたは 2 つのレコード・セグメント
- 2 つのレコード・セグメント
- 1 つのレコード・セグメント

可変長レコードの場合、各レコードの長さまたは各レコード・セグメントの長さには、レコードやレコード・セグメント用の 4 制御バイトが含まれています。

上記のリストは、レコード・オプションおよびレコード・セグメント・オプション、すなわち固定長ブロック化、可変長ブロック化、非ブロック化、またはスパン、非スパンの考えられるすべての組み合わせをまとめたものです。スパン・レコードのブロック・サイズを指定する場合、各レコードおよび各レコード・セグメントにはレコード長のための 4 制御バイトが必要であり、これらの数量は各ブロックに必要な 4 制御バイトに加算されるものであることに注意してください。

2. 必要なその他の制御バイト

- 可変長ブロック・レコードには、4 バイトが必要です (ブロック・サイズの場合)。
- 固定長レコードおよび不定長レコードには、追加の制御バイトは必要ありません。

3. 必要なブロック接頭部バイト (ASCII データ・セットのみ)

block-size (ブロック・サイズ) は、整数または属性 **FIXED BINARY(31,0) STATIC** を持つ変数として指定することができます。

レコード長の値には、次のような規則があります。

最大長:

32760

ゼロ値:

z/OS 上で **BLKSIZE** を 0 (ゼロ) に設定すると、データ機能プロダクトがブロック・サイズを設定します。このトピックの詳細については、214 ページの『レコード・フォーマット、**BLKSIZE**、および **RECSIZE** のデフォルト』を参照してください。

負の値:

UNDEFINEDFILE 条件が発生します。

次のように、ブロック・サイズとレコード長の関係は、レコードのフォーマットによって異なります。

FB フォーマットまたは **FBS** フォーマット

このブロック・サイズは、レコード長の倍数でなければなりません。

VB フォーマット:

このブロック・サイズは、次の値の合計値以上でなければなりません。

1. 任意のレコードの最大長
2. 4 制御バイト

VS フォーマットまたは **VBS** フォーマット:

ブロック・サイズは、レコード長より小、等しい、またはより大きくすることができます。

注:

- 非ブロック化 (**F** フォーマットまたは **V** フォーマット) レコードで **BLKSIZE** オプションを使用するには、以下のいずれか方法を使用します。

- BLKSIZE オプションを指定し、RECSIZE オプションを指定しない。レコード長をブロック・サイズ (制御バイトまたは接頭語バイトを差しく) と同じ値に設定し、レコード・フォーマットを未変更のままにする。
- BLKSIZE と RECSIZE の両方を指定し、2 つの値の関係と使用するレコード・フォーマットのブロック化が矛盾しないことを確認する。レコード・フォーマットを FB、VB に設定する (どちらでも該当する方)。
- FB フォーマットまたは FBS フォーマット・レコードでブロック・サイズがレコード長と等しい場合は、レコード・フォーマットは F に設定されます。
- BLKSIZE オプションは VSAM データ・セットには適用されません。そのため、指定しても無視されます。

レコード・フォーマット、BLKSIZE、および RECSIZE のデフォルト

VSAM 以外のデータ・セットにレコード・フォーマット、ブロック・サイズ、またはレコード長を指定しない場合は、次のようなデフォルト・アクションがとられます。

レコード・フォーマット:

関連付けられている DD ステートメントまたはデータ・セット・ラベル内で検索が行われます。値が検出されない場合は、UNDEFINEDFILE 条件が発生します。ただし、ダミー・データ・セットまたはフォアグラウンド端末に関連付けられているファイルの場合を除きます。その場合は、レコード・フォーマットが U に設定されます。

ブロック・サイズあるいはレコード長:

ブロック・サイズまたはレコード長のいずれかが指定されている場合は、関連付けられている DD ステートメントまたはデータ・セット・ラベル内で、もう一方の検索が行われます。その検索により値が検出されても、値が指定オプションの値と矛盾している場合は、UNDEFINEDFILE 条件が発生します。検索が成功しなかった場合は、指定オプションから値が導き出されます (制御バイトまたは接頭部バイトを追加あるいは除去して)。

ブロック・サイズもレコード長も指定されていない場合は、UNDEFINEDFILE 条件が発生します。ただし、ダミー・データ・セットと関連付けられているファイルの場合を除きます。その場合、BLKSIZE は 121 (F フォーマット・レコードや U フォーマット・レコードの場合)、または 129 (V フォーマット・レコードの場合) に設定されます。フォアグラウンド端末に関連付けられているファイルの場合は、RECSIZE は 120 に設定されます。

z/OS 上でデータ機能プロダクト (DFP) のシステム決定ブロック・サイズ機能を使用している場合は、割り当てられている装置タイプに最適なブロック・サイズが DFP により算出されます。DD 割り当てまたは ENVIRONMENT ステートメント内で BLKSIZE(0) を指定する場合は、レコード長、レコード・フォーマット、および装置タイプを使って BLKSIZE が DFP により算出されます。

GENKEY オプション - キーの分類

GENKEY (総称キー) オプションは、INDEXED キー順データ・セットおよび VSAM キー順データ・セットにのみ適用されます。このオプションを使用すると、

データ・セット内に記録されているキーを分類したり、キー・クラスに従ってレコードにアクセスするために SEQUENTIAL KEYED INPUT ファイルまたは SEQUENTIAL KEYED UPDATE ファイルを使用することができます。

▶—GENKEY—◀

総称キーはキーのクラスを識別する文字ストリングです。このストリングで始まるすべてのキーはそのクラスのメンバーです。例えば、記録済みキー「ABCD」、「ABCE」、および「ABDF」はすべて、総称キー「A」および「AB」で識別されるクラスのメンバーであり、最初の 2 つのキーは、クラス「ABC」のメンバーでもあります。そして 3 つの記録済みキーはそれぞれ「ABCD」、「ABCE」、および「ABDF」の各クラスの固有のメンバーと考えることができます。

GENKEY オプションを使用すると、特定クラスのキーを持つ最初のレコードから、VSAM データ・セットを順次に読み取ることも更新することもできます。また、INDEXED データ・セットの場合は、このオプションを使用すると、特定のクラスのキーを持つ最初の非ダミー・レコードを順次に読み取ることも更新することもできます。READ ステートメントの KEY オプションに総称キーを入れることにより、クラスを識別することができます。KEY オプションを指定せずに、READ ステートメントで後続のレコードを読みとることができます。キー・クラスの終わりに到達したときに、その旨の指摘は行われません。

KEY オプションを指定した READ ステートメントを使用することによって、特定クラスのキーを持つ最初のレコードを検索することができますが、KEYTO オプションは KEY オプションと同じステートメントで使用することができないため、レコードに組み込みキーがない限り、実際のキーを得ることはできません。

次の例では、3 バイトを超えているキーの長さが想定されます。

```
DCL IND FILE RECORD SEQUENTIAL KEYED
  UPDATE ENV (GENKEY);
  .
  .
  .
  READ FILE(IND) INTO(INFIELD)
    KEY ('ABC');
  .
  .
  .
NEXT: READ FILE (IND) INTO (INFIELD);
  .
  .
  .
  GO TO NEXT;
```

最初の READ ステートメントによって、「ABC」で始まるキーを持ったデータ・セット内の最初の非ダミー・レコードが、INFIELD に読み込まれることになります。第二の READ ステートメントが実行されるごとに、次に高位のキーを持つ非ダミー・レコードが取り出されます。2 番目の READ ステートメントを繰り返し実行すると、次々により高位のキー・クラスからレコードが読み取られることになりますが、それは、キー・クラスの終わりに到達してもそのことが指摘されないからです。特定クラスのキーを超えて読み取りを続けたくない場合は、各自の責任で各キ

ーを検査してください。最初の READ ステートメントをもう一度実行すると、そのファイルはキー・クラス「ABC」の最初のレコードの位置に再配置されます。

指定クラス内のキーを持つレコードがデータ・セットにない場合、または指定クラスのキーを持つレコードがすべてダミー・レコードの場合は、KEY 条件が発生します。そのあと、データ・セットは上位のキーを持つ次のレコードに、あるいはファイルの終わりに位置付けられます。

GENKEY オプションの有無によって、KEYLEN サブパラメーターで指定されているキー長より短いソース・キーを提供する READ ステートメントの実行が影響を受けます。KEYLEN サブパラメーターは、索引付きデータ・セットを定義する DD ステートメント内にあります。GENKEY オプションを指定すると、それによってソース・キーが総称キーと解釈され、キーがソース・キーで始まるデータ・セット内の最初の非ダミー・レコードに、そのデータ・セットが位置付けられることになります。GENKEY オプションを指定しないと、指定したキー長になるよう、READ ステートメントの短いソース・キーの右側にブランクが埋め込まれ、データ・セットは、この埋め込まれたキーを持ったレコード (このようなレコードが存在する場合) に位置付けられます。WRITE ステートメントの場合は、短いソース・キーには常にブランクで埋め込まれます。

GENKEY オプションを使用しても、キー長が指定キー長以上であるソース・キーを提供してもその結果に影響はありません。必要に応じて右側が切り捨てられるソース・キーは、特定のレコード (キーがそのクラスの唯一のメンバーだと考えられるレコード) を識別します。

SCALARVARYING オプション - 可変長ストリング

SCALARVARYING オプションは可変長ストリングの入出力について使用します。このオプションではどのフォーマットのレコードで 사용할 こともできます。

▶▶—SCALARVARYING—◀◀

ストレージを可変長ストリング用に割り当てるとき、コンパイラーは、ストリングの現行長を指定するための 2 バイト接頭部を組み込みます。エレメント可変長ストリングの場合は、そのファイルに SCALARVARYING が指定されている場合にだけ、この接頭部が出力時に組み込まれるか、入力時に認識されます。

位置指定モード・ステートメント (LOCATE および READ SET) を使って、エレメント可変長ストリングを持つデータ・セットを作成し読み取る場合は、SCALARVARYING を指定して長さ接頭部の存在を認識させる必要があります。これは、バッファの位置を指定するポインターは、常に長さ接頭部の開始位置を指すと想定されるからです。

SCALARVARYING を指定して、エレメント可変長ストリングが送信される場合は、長さ接頭部を組み込むためにレコード長に 2 バイトを与える必要があります。

SCALARVARYING を使用して作成されるデータ・セットは、SCALARVARYING を指定しているファイルだけがアクセスするようにします。

同じファイルに SCALARVARYING と CTLASA/CTL360 を指定することはできません。それを行うと、最初のデータ・バイトがあいまいになるからです。

KEYLENGTH オプション

KEYED ファイルの記録済みキーの長さを指定するには、KEYLENGTH オプションを使用します。この場合、n は長さです。INDEXED ファイルにも KEYLENGTH オプションを指定することができます。

▶—KEYLENGTH—(—n—)————▶

検査の目的で VSAM ファイルの宣言に KEYLENGTH オプションを指定する場合、このオプションに指定したキー長がデータ・セットに定義されている値と矛盾すれば、UNDEFINEDFILE 条件が発生します。

ORGANIZATION オプション

ORGANIZATION は、PL/I ファイルに関連付けられているデータ・セットの編成を指定します。

▶—ORGANIZATION—(—

CONSECUTIVE
INDEXED
RELATIVE

—)————▶

CONSECUTIVE

該当のファイルが連続データ・セットに関連付けられていることを表します。連続ファイルは、ネイティブ・データ・セットでも、データ・セット VSAM、ESDS、RRDS、または KSDS でもかまいません。

RELATIVE

該当のファイルが相対データ・セットに関連付けられていることを表します。RELATIVE は、そのデータ・セットに記録済みキーがないレコードが含まれていることを指定します。相対ファイルは、VSAM 直接データ・セットです。相対キーの範囲は、1 から nnnn までです。

PL/I レコード入出力で使用されるデータ・セットのタイプ

RECORD 属性を持ったデータ・セットは、データがプログラム変数に入っているとおり補助記憶域との間で送受信されるレコード単位データ伝送によって処理されます。データ変換は行われません。データ・セット内のレコードは、プログラム内の各変数に対応しています。

表 14 に、PL/I レコード入出力で利用できるさまざまなデータ・セット・タイプで使用可能な機能を示します。

表 14. PL/I レコード入出力で利用できるデータ・セット・タイプの比較

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
SEQUENCE	キー順	入力順	番号付き	キー順	入力順	領域別
装置	DASD	DASD	DASD	DASD	DASD、 カードなど	DASD

表 14. PL/I レコード入出力で利用できるデータ・セット・タイプの比較 (続き)

		VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)
ACCESS							
1	キー	123	123	123	12	2	12
2	順次					3 テープのみ	
3	逆方向						
代替							
索引		123	123	いいえ	いいえ	いいえ	いいえ
アクセス							
(上と同様)							
拡張方法							
		新規 キーで	終了時 に	空の スロット の中	新規 キーで	終了時 に	空の スロット の中
DELETION							
		はい、1	いいえ	はい、1	はい、2	いいえ	はい、1
1	スペースは 再使用可能						
2	スペースは 再使用でき ない						

以下の各章は、各種データ・セットでのレコード入出力データ・セットの用法を説明しています。

- 227 ページの『第 8 章 連続データ・セットの定義と使用』
- 257 ページの『第 9 章 領域データ・セットの定義と使用』
- 271 ページの『第 10 章 VSAM データ・セットの定義と使用』

環境変数の設定

z/OS UNIX System Services の場合のみ

z/OS UNIX で使用できるようにするために、設定およびエクスポートを行える環境変数がいくつかあります。

すべてのユーザーがアクセスできるように環境変数をシステム全体用に設定するには、サブセクションで推奨されている行をファイル `/etc/profile` に追加します。ある特定ユーザーだけに環境変数を設定するには、該当するユーザーのホーム・ディレクトリーにあるファイル `.profile` にその環境変数を追加します。変数は、次にユーザーがログオンしたときに設定されます。

次の例に、環境変数の設定方法を示します。

```
LANG=ja_JP
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/prime/%N
LIBPATH=/home/joe/usr/lib:/home/joe/mylib:/usr/lib
export LANG NLSPATH LIBPATH
```

前述の例では、最後のステートメントの代わりに、`export` をその前の各行に追加することもできます (`export LANG=ja_JP...`)。

ECHO コマンドを使用すると、任意の環境変数の現行設定値を調べることができます。BYPASS の値を定義するには、次の 2 つの例のどちらかを使用します。

```
echo $LANG
```

```
echo $LIBPATH
```

z/OS UNIX System Services の場合のみ の終り

PL/I 標準ファイル (SYSPRINT と SYSIN)

z/OS UNIX System Services の場合のみ

デフォルトにより、SYSIN は stdin から読み取られ、SYSPRINT は stdout に送られます。関連付けを変更したい場合は、OPEN ステートメントの TITLE オプションを使うか、データ・セットまたは別の装置に名前を指定する DD_DDNAME 環境変数を設定する必要があります。環境変数については、218 ページの『環境変数の設定』で説明しています。

z/OS UNIX System Services の場合のみ の終り

標準入力、標準出力、および標準エラー装置のリダイレクト

z/OS UNIX System Services の場合のみ

標準入力、標準出力、および標準エラー装置をファイルにリダイレクトすることもできます。リダイレクトを使用できるのは、次のプログラムです。

```
Hello2: proc options(main);  
    put list('Hello!');  
end;
```

このプログラムのコンパイルおよびリンク後、コマンド行に次のように入力してプログラムを呼び出すことができます。

```
hello2 > hello2.out
```

stdout と stderr を 1 つのファイルにまとめる場合は、次のコマンドを入力します。

```
hello2 > hello2.out 2>&1
```

表示ステートメントの場合と同様に、> (より大) 記号を使用すると、その後ろに指定されているファイル、この場合は hello2.out に出力がリダイレクトされます。これは、'Hello' という語がファイル hello2.out に書き込まれることを意味します。PRINT 属性はデフォルトで SYSPRINT に適用されるため、出力にはプリンター制御文字も組み込まれるということに注意してください。

READ ステートメントは stdin からのデータにアクセスすることができます。ただし、そのデータが入るレコードには、値が 1 である LRECL がなければなりません。

z/OS UNIX System Services の場合のみ の終り

第 7 章 ライブラリーの使用

z/OS オペレーティング・システムでは、「区分データ・セット」「区分データ・セット/拡張」および「ライブラリー」の 3 つの用語は同義であり、他のデータ・セット（通常、ソース・モジュール、オブジェクト・モジュール、またはロード・モジュールの形のプログラム）の保管に使用できるタイプのデータ・セットを指します。ライブラリーは、直接アクセス・ストレージに保管する必要がある、全体が 1 つのボリューム内に入っていなければなりません。ライブラリーには、連続して編成された、メンバーと呼ばれる独立したデータ・セットが入っています。各メンバーには、ライブラリーの一部であるディレクトリーに保管される 8 文字を超えない長さの固有名が付いています。1 つのデータ・セット・ラベルだけが維持されるので、1 つのライブラリーに属するすべてのメンバーは同じデータ特性を持っている必要があります。

ディレクトリー内に新規項目を入れる十分なスペースが残っていないか、またはメンバーそのもののスペースが足りなくなるまでは、メンバーを個々に作成することができます。メンバーの名前を指定すれば、メンバーを個別にアクセスすることができます。

DD ステートメントまたはそれらに対応する会話型での同等機能を使用することにより、メンバーの作成およびアクセスを行うことができます。

メンバーは、IBM のユーティリティー・プログラム IEHPROGM を使用して削除することができます。このプログラムは、ディレクトリーからメンバー名を削除するので、そのメンバーには以降アクセスすることはできませんが、例えば IBM のユーティリティー・プログラム IEBCOPY を使用して、ライブラリーを作成しなおすか、または未使用スペースを圧縮しない限り、そのメンバーが占めていたスペースを使用することはできません。DD ステートメントの DISP パラメーターを使ってメンバーを削除しようとする、データ・セット全体が削除されてしまいます。

ライブラリーのタイプ

PL/I プログラムでは、次のタイプのライブラリーを使用することができます。

- ・ システム・プログラム・ライブラリー SYS1.LINKLIB またはこれと同等のもの。このライブラリーには、コンパイラーやリンケージ・エディターなどのすべてのシステム処理プログラムを収容することができます。
- ・ 専用プログラム・ライブラリー。このライブラリーには、通常、ユーザーが作成したプログラムが収容されます。多くの場合、一時専用ライブラリーを作成すれば、リンケージ・エディターからのロード・モジュール出力を、同一ジョブ内の後半のジョブ・ステップで実行するまでの間保管しておけるので便利です。一時ライブラリーは、ジョブの終了時に削除されます。専用ライブラリーは、自動ライブラリー呼び出し用にもリンケージ・エディターやローダーによって使用されます。

- ・ システム・プロシージャー・ライブラリー SYS1.PROCLIB またはこれと同等のもの。このライブラリーには、ユーザーのシステム用としてカタログされているジョブ制御プロシージャーが入っています。

ライブラリーの用法

PL/I プログラムは、ライブラリーを直接使用することができます。ユーザーが新たにメンバーをライブラリーに追加しようとする、オペレーティング・システムは、その関連ファイルがクローズされるときに、データ・セット名の一部として指定されているメンバー名を使って新規メンバーのディレクトリー項目を作成します。

ユーザーがライブラリーのメンバーにアクセスする場合、オペレーティング・システムは、ユーザーがデータ・セット名の一部として指定したメンバー名からそのディレクトリー項目を見つけ出すことができます。

同一ライブラリーの複数のメンバーを 1 つの PL/I プログラムで処理することはできますが、このようなファイルは同時に 1 つしか出力としてオープンすることはできません。DD ステートメントに入っているメンバー名を指定することにより、異なるメンバーにアクセスすることができます。

ライブラリーの作成

ライブラリーを作成するには、ジョブ・ステップ内に、表 15 に示してある情報の入った DD ステートメントを組み込みます。ライブラリーの作成に必要な情報は、SPACE パラメーターを除けば、連続して編成されたデータ・セット (246 ページの『レコード入出力の使用によるファイルの定義』参照) の場合とほとんど同じです。

表 15. ライブラリー作成時に必要な情報

必須情報	DD ステートメントのパラメーター
使用する装置のタイプ	UNIT=
ライブラリーを入れるボリュームの通し番号	VOLUME=SER
ライブラリー名	DSNAME=
ライブラリーのスペース必須容量	SPACE=
ライブラリーの後処理	DISP=

SPACE パラメーター

ライブラリーを定義するための DD ステートメント内の SPACE パラメーターは、常に次の形でなければなりません。

SPACE=(units,(quantity,increment,directory))

3 つ目の項目 (increment) は省略して、コンマでその省略を示すことができますが、割り当てられるディレクトリー・ブロック数を指定する最後の項目は常に指定する必要があります。

ライブラリーに必要な補助記憶域の大きさは、保管するメンバーの数とサイズと、メンバーがどのくらい頻繁に追加または置換されるかによって異なります。(削除

されたメンバーのスペースは解放されません。) 必要なディレクトリー・ブロック数は、メンバー数および別名数によって異なります。SPACE パラメーターに増分容量を指定すると、データ・セット作成時や新規メンバー追加時に必要が生じた場合、オペレーティング・システムがデータ・セットにさらに必要な領域を確保することができます。ただし、ディレクトリー・ブロック数は作成時に固定されるため、増やすことはできません。

例えば、DD ステートメントは次のようになります。

```
// PDS DD UNIT=SYSDA,VOL=SER=3412,  
// DSN=ALIB,  
// SPACE=(CYL,(5,,10)),  
// DISP=(,CATLG)
```

上記のステートメントは、ボリューム通し番号 3412 を持つ DASD の 5 つのシリンドラーを新規ライブラリー名 ALIB に割り振ることと、新規ライブラリー名をシステム・カタログに登録することを、ジョブ・スケジューラーに要求しています。SPACE パラメーターの最後の項目は、データ・セットに割り振られているスペースの一部を 10 個のディレクトリー・ブロック用に予約しています。

ライブラリー・メンバーの作成と更新

各ライブラリー・メンバーは、同じ特性を持っていなければなりません。同じ特性メンバーを持っていない場合は、あとのメンバー検索が難しくなります。同じ特性が必要な理由は、ボリューム目録 (VTOC) には、ライブラリー用のデータ・セット制御ブロック (DSCB) が 1 つ入っているだけで、各メンバー別用のものはないからです。PL/I プログラムを使ってメンバーを作成する場合は、オペレーティング・システムによりディレクトリー項目が作成されます。つまり、ユーザーはユーザー・データ・フィールドに情報を入力することはできません。

ライブラリーとメンバーを同時に作成するときには、DD ステートメント内に 222 ページの『ライブラリーの作成』の下にリストアップされているすべてのパラメーター (ただし、データ・セットを一時的なものにするのであれば、DISP パラメーターは省くことができる) を入れなければなりません。DSNAME パラメーターは、メンバー名を括弧で囲んで指定する必要があります。例えば、DSNAME=ALIB(MEM1) によりデータ・セット ALIB 内のメンバー MEM1 の名前が指定されます。メンバーがリンケージ・エディターによってライブラリー内に入れられると、DSNAME パラメーター内にメンバー名を入れる代わりに、リンケージ・エディターの NAME ステートメントまたは NAME コンパイル時オプションを使用することができます。メンバーの特性 (レコード・フォーマットなど) を DCB パラメーターあるいは PL/I プログラムに記述しなければなりません。これらの特性は、そのデータ・セットに追加される他のメンバーにも適用されます。

既存ライブラリーに追加するメンバーを作成するときには、SPACE パラメーターは必要ありません。元のスペース割り振りは、個々のメンバーに対してではなく、ライブラリー全体に対して適用されるからです。さらに、そのメンバーの特性を記述する必要もありません。その理由は、ライブラリー用の DSCB 内に既に特性が記録されているからです。

1 つのジョブ・ステップで複数のメンバーをライブラリーに追加するには、各メンバーごとに DD ステートメントを組み込み、そしてそのライブラリーを参照するファイルをクローズしてから次のファイルをオープンするにしなければなりません。

例

カタログ式プロシージャ IBMZC を使用して簡単な PL/I プログラムをコンパイルし、オブジェクト・モジュールを EXLIB という名前の新規ライブラリーに入れる例を、図 20 に示します。新規ライブラリーを定義し、オブジェクト・モジュールを指名する DD ステートメントは、カタログ式プロシージャ内の DD ステートメント SYSLIN を指定変更します。(PL/I プログラムは、関数プロシージャであり、TIME 組み込み関数で作成される文字ストリングの形に 2 つの値が指定されている場合、ミリ秒単位でその値の差を戻します。)

カタログ式プロシージャ IBMZCL を使用して PL/I プログラムをコンパイルおよびリンク・エディットし、ロード・モジュールを既存ライブラリー HPU8.CCLM に入れる例を、225 ページの図 21 に示します。

```
//OPT10#1 JOB
//TR      EXEC  IBMZC
//PLI.SYSLIN DD UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//          SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSIN DD *
      ELAPSE: PROC(TIME1,TIME2);
          DCL (TIME1,TIME2) CHAR(9),
              H1 PIC '99' DEF TIME1,
              M1 PIC '99' DEF TIME1 POS(3),
              MS1 PIC '99999' DEF TIME1 POS(5),
              H2 PIC '99' DEF TIME2,
              M2 PIC '99' DEF TIME2 POS(3),
              MS2 PIC '99999' DEF TIME2 POS(5),
              ETIME FIXED DEC(7);
          IF H2<H1 THEN H2=H2+24;
          ETIME=((H2*60+M2)*60000+MS2)-((H1*60+M1)*60000+MS1);
          RETURN(ETIME);
      END ELAPSE;
/*
```

図 20. コンパイルされたオブジェクト・モジュール用の新規ライブラリーの作成

```

//OPT10#2 JOB
//TRLE      EXEC  IBMZCL
//PLI.SYSIN DD *
  MNAME: PROC  OPTIONS(MAIN);
      .
      .
      .
      program
      .
      .
      .

  END MNAME;
/*
//LKED.SYSLMOD DD  DSN=HPU8.CCLM(DIRLIST),DISP=OLD

```

図 21. ロード・モジュールの既存ライブラリーへの配置

PL/I プログラムを使って、ライブラリーのメンバー内の 1 つ以上のレコードを追加または削除するには、そのライブラリー内の別の部分でそのメンバーを全部作成し直さなければなりません。メンバーがそれまで占有していたスペースは再度使用することができないため、これはあまり経済的な提案とは言えません。ユーザーの PL/I プログラム内でファイルを 2 つ使う必要がありますが、2 つとも同じ DD ステートメントと関連付けることができます。226 ページの図 23 に示すプログラムは、図 22 のプログラムで作成されたメンバーを更新します。このプログラムは、リンクだけのレコードを除き、元のメンバーのレコードをすべてコピーします。

```

//OPT10#3 JOB
//TREX EXEC IBMZCBG
//PLI.SYSIN DD *
  NMEM: PROC OPTIONS(MAIN);
    DCL IN FILE RECORD SEQUENTIAL INPUT,
        OUT FILE RECORD SEQUENTIAL OUTPUT,
        P POINTER,
        IOFIELD CHAR(80) BASED(P),
        EOF BIT(1) INIT('0'B);
    OPEN FILE(IN),FILE (OUT);
    ON ENDFILE(IN) EOF='1'B;
    READ FILE(IN) SET(P);
    DO WHILE (¬EOF);
      PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
      WRITE FILE(OUT) FROM(IOFIELD);
      READ FILE(IN) SET(P);
    END;
    CLOSE FILE(IN),FILE(OUT);
  END NMEM;
/*
//GO.OUT DD UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//      DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//      DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
  MEM: PROC OPTIONS(MAIN);
    /* this is an incomplete dummy library member */

```

図 22. PL/I プログラム内でのライブラリー・メンバーの作成

```
//OPT10#4 JOB
//TREX EXEC IBMZCBG
//PLI.SYSIN DD *
UPDTM: PROC OPTIONS(MAIN);
  DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
      EOF BIT(1) INIT('0'B),
      DATA CHAR(80);
  ON ENDFILE(OLD) EOF = '1'B;
  OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');
  READ FILE(OLD) INTO(DATA);
  DO WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
    IF DATA=' ' THEN ;
    ELSE WRITE FILE(NEW) FROM(DATA);
    READ FILE(OLD) INTO(DATA);
  END;
  CLOSE FILE(OLD),FILE(NEW);
END UPDTM;
/*
//GO.OLD DD DSN=HPU8.ALIB(NMEM),DISP=(OLD,KEEP)
```

図 23. ライブラリー・メンバーの更新

ライブラリー・ディレクトリーにある情報の取り出し

ライブラリー・ディレクトリーは、データ・セットの先頭に置かれる一連のレコード (項目) です。1 つ以上のディレクトリー項目が各メンバーに存在します。各項目には、メンバー名、ライブラリー内のメンバーの相対アドレス、および可変長のユーザー・データが入っています。

ユーザー・データは、メンバーを作成したプログラムによって挿入される情報です。リンケージ・エディターで作成されたメンバー (ロード・モジュール) を参照する項目は、システムのマニュアルに記載されている標準形式のユーザー・データを組み込みます。

PL/I プログラムを使ってメンバーを作成するときには、オペレーティング・システムがユーザーに代わってディレクトリー項目を作成するので、ユーザーはユーザー・データを書き込むことはできません。ただし、アセンブラー言語マクロ命令を使用すると、メンバーを作成したり、独自のユーザー・データを作成することができます。このためにマクロ命令を使用する方法は、データ管理資料に説明されています。

第 8 章 連続データ・セットの定義と使用

この章では、連続データ・セット編成について、さらにストリーム指向データ伝送およびレコード単位データ伝送用の連続データ・セットを定義するための ENVIRONMENT オプションについて述べています。その次に、伝送タイプごとに、連続データ・セットの作成、アクセスおよび更新方法についても説明します。

連続編成のデータ・セット内では、各レコードは連続する物理的な位置のみに基づいて編成されます。データ・セットが作成されるときは、レコードは提示される順番で連続的に書き込まれます。なお、レコードは、レコードが書き込まれた順序でのみ検索できます。連続データ・セットの場合の有効ファイル属性と ENVIRONMENT オプションに関しては、209 ページの表 13 を参照してください。

ストリーム指向データ伝送の使用

ここでは、STREAM 属性の PL/I ファイルで使用するデータ・セットの定義方法について説明します。使用できる ENVIRONMENT オプション、データ・セットを作成しデータ・セットにアクセスする方法について説明します。また、データ・セットの作成、アクセス時に使用する DD ステートメントの必須パラメーターを表にまとめ、文章による記述を説明するために PL/I プログラムの例もいくつか掲載しています。

STREAM 属性を持つデータ・セットは、ストリーム指向データ伝送で処理されます。そのため、PL/I プログラムは、ブロックやレコードの境界を無視して、各データ・セットを、文字の形またはグラフィックの形のデータ値の 1 つの連続するストリームとして扱うことができます。

ストリーム指向データ伝送用のデータ・セットを作成し、それにアクセスするには、「PL/I 言語解説書」で説明しているリスト指示、データ指示、および編集指示の入出力ステートメントを使用します。

出力の場合、PL/I は必要に応じてデータ項目をプログラム変数から文字の形に変換し、文字またはグラフィックスのストリームを、データ・セットに伝送するためにレコードに構築します。

入力の場合、PL/I はデータ・セットからレコードを取り出し、ユーザー・プログラムが要求した複数のデータ項目に分割し、さらにそれをプログラム変数に割り当て、それに適した形に変換します。

ストリーム指向データ伝送は、グラフィック・データの読み取りと書き込みに使用できます。適切なプログラミング・サポートがあれば、グラフィックスを表示し、印刷し、入力することができる端末、プリンター、およびデータ入力装置があります。ユーザーのデータが使用する装置または印刷ユーティリティー・プログラムで受け入れられているフォーマットになっているかどうか確認する必要があります。

ストリーム入出力の使用によるファイルの定義

ストリーム指向データ伝送用のファイルは、次のような属性を用いるファイル宣言で定義します。

```
DCL filename FILE STREAM
      INPUT | {OUTPUT [PRINT]}
      ENVIRONMENT(options);
```

デフォルト・ファイル属性は 209 ページの表 13 に示してあります。FILE 属性については、「PL/I 言語解説書」に説明があります。PRINT 属性の詳細については、235 ページの『ストリーム入出力による PRINT ファイルの使用』に説明があります。ENVIRONMENT 属性のオプションについては、以下に説明します。

ENVIRONMENT オプションの指定

209 ページの表 13 に、ENVIRONMENT オプションの要約があります。ストリーム指向データ伝送で利用できるオプションは、次のとおりです。

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
F|FB|FS|FBS|V|VB|VS|VBS|U
RECSIZE(record-length)
BLKSIZE(block-size)
GRAPHIC
LEAVE|REREAD
```

BLKSIZE については、212 ページから始まる『BLKSIZE オプション』を参照してください。LEAVE および REREAD については、本章の後にある 248 ページの『LEAVE|REREAD』の始めに記載されています。残りのオプションは、以下で説明します。

CONSECUTIVE

STREAM ファイルは CONSECUTIVE データ・セット編成を持っていないわけではありませんが、CONSECUTIVE はデフォルトのデータ・セット編成なので、これを ENVIRONMENT オプション内に指定する必要はありません。STREAM ファイルの CONSECUTIVE オプションは、203 ページの『データ・セットの編成』に記載されているものとまったく同じです。

▶▶—CONSECUTIVE—◀◀

レコード・フォーマット・オプション

ストリーム指向データ伝送では、レコード境界は無視されますが、データ・セットを作成する場合、レコード・フォーマットは重要な意味を持ちます。これは、レコード・フォーマットがデータ・セットが占有するストレージの容量と、データを処理するプログラムの効率に影響するためだけでなく、データ・セットが後で、レコード単位データ伝送でも処理できるようにするためです。

いったんレコード・フォーマットを指定したならば、ストリーム指向データ伝送を使う限り、レコードおよびブロックを意識する必要はありません。データ・セットは、行に配置された一連の文字またはグラフィックスと見なすことができます。また、SKIP オプションまたはフォーマット項目 (PRINT ファイルでは、PAGE オプション、LINE オプションおよびフォーマット項目) を使って、新しい行を選択することができます。



レコードは、次に挙げるフォーマットのうちいずれか 1 つを持つことができます。これらのフォーマットは 201 ページの『レコード・フォーマット』に説明してあります。

固定長	F	非ブロック化
	FB	ブロック化
	FS	非ブロック化、標準
	FBS	ブロック化、標準
可変長	V	非ブロック化
	VB	ブロック化
	VS	
	VBS	
不定長	U	(ブロック化できない)

なお、レコードのブロック化も非ブロック化も自動的に行われます。

RECSIZE

ストリーム指向データ伝送の場合の RECSIZE は 208 ページの『ENVIRONMENT 属性での特性の指定』に記載されているものと同じです。また、OPEN ステートメントの LINESIZE オプションで指定された値は、RECSIZE オプションで指定された値を指定変更します。LINESIZE については、「PL/I 言語解説書」で説明されています。

グラフィックスのリスト指示伝送およびデータ指示伝送についてのレコード・サイズに関する追加の考慮事項は、「PL/I 言語解説書」に記載されています。

レコード・フォーマット、BLKSIZE および RECSIZE のデフォルト値

レコード・フォーマット、BLKSIZE、または RECSIZE オプションを ENVIRONMENT 属性内またはそれに関連した DD ステートメントまたはデータ・セット・ラベル内で指定しないと、次のような処置がとられます。

入力ファイル:

デフォルトは、214 ページの『レコード・フォーマット、BLKSIZE、および RECSIZE のデフォルト』で説明されているレコード単位データ伝送と同様に適用されます。

出力ファイル

レコード・フォーマット

VB フォーマットに設定されます。

レコード長

指定した、またはデフォルトの LINESIZE の値が使用されます。

PRINT ファイル:

F、FB、FBS、または U: 行サイズ + 1

V または VB: 行サイズ + 5

非 PRINT ファイル:

F、FB、FBS、または U: 行サイズ

V または VB: 行サイズ + 4

ブロック・サイズ:

F、FB、または FBS: レコード長

V または VB: レコード長 + 4

GRAPHIC オプション

編集指示入出力の GRAPHIC オプションを指定します。

▶—GRAPHIC—◀

入力データや出力データにグラフィックスが含まれているのに、GRAPHIC オプションが指定されていない場合は、リスト指示入出力およびデータ指示入出力の ERROR 条件が発生します。

編集指示入出力で GRAPHIC オプションを指定すると、出力時に DBCS 変数と定数の左右に区切り文字が追加され、グラフィック入力時にも左右に区切り文字が付け加えられます。GRAPHIC オプションを指定しない場合は、出力データの左右に区切り文字は追加されず、また、入力のグラフィックスにも左右の区切り文字は必要ありません。また、GRAPHIC オプションを指定した場合は、入力データの左右に区切り文字がないと、ERROR 条件が発生します。

グラフィック・データ・タイプおよび編集指示入出力の G フォーマット項目の詳細については、「PL/I 言語解説書」を参照してください。

ストリーム入出力によるデータ・セットの作成

データ・セットを作成するには、ユーザーの PL/I プログラム内で、またはデータ・セットを定義する DD ステートメント内で、特定の情報をオペレーティング・システムに与える必要があります。z/OS UNIX では、次のような方法でこの情報を与えます。

- OPEN ステートメントの TITLE オプション
- DD_DDNAME 環境変数
- ENVIRONMENT 属性

次に、必須情報を示し、ユーザーが与えることのできるいくつかのオプションの情報について説明します。

必須情報

ユーザーのアプリケーションで STREAM ファイルを作成する場合は、PL/I は、次の入手源の 1 つから、優先順位順に (降順)、そのファイルの行サイズの値を導き出します。

- OPEN ステートメントの LINESIZE オプション

- ENVIRONMENT 属性の RECSIZE オプション
- OPEN ステートメントの TITLE オプションの RECSIZE オプション
- DD_DDNAME 環境変数の RECSIZE オプション
- PL/I 提供のデフォルト値

LINESIZE の値が与えられていて、RECSIZE の値が与えられていない場合は、PL/I は、次のようにレコード長を導き出します。

- V フォーマット PRINT ファイルの場合、値は LINESIZE + 5
- V フォーマット非 PRINT ファイルの場合、値は LINESIZE + 4
- F フォーマット PRINT ファイルの場合、値は LINESIZE + 1
- 上記以外の場合はすべて、値は LINESIZE

LINESIZE の値が与えられておらず、RECSIZE の値が与えられている場合は、PL/I は、次のように RECSIZE から行サイズの値を導き出します。

- V フォーマット PRINT ファイルの場合、値は RECSIZE - 5
- V フォーマット非 PRINT ファイルの場合、値は RECSIZE - 4
- F フォーマット PRINT ファイルの場合、値は RECSIZE - 1
- 上記以外の場合はすべて、値は RECSIZE

LINESIZE も RECSIZE も与えられていない場合は、PL/I は、ファイル属性および関連付けられたデータ・セットのタイプに基づいて、デフォルトの行サイズの値を決定します。PL/I が適切なデフォルトの行サイズを与えられない場合は、UNDEFINEDFILE 条件が発生します。

次の場合には、デフォルトの行サイズが OUTPUT ファイルに与えられます。

- ファイルは PRINT 属性を持っている。この場合、値はタブ制御テーブルから得られます。
- 関連データ・セットが端末 (stdout または stderr) である。この場合、行サイズの値は 120 です。

なお、LINESIZE オプションが (OPEN ステートメントで) 指定されていて、かつ (ENVIRONMENT 属性、TITLE オプション、または DD ステートメントで) RECSIZE も指定されていて、(レコード・フォーマットおよび適切な制御バイトのオーバーヘッドを考慮に入れて) レコード・サイズの値が小さ過ぎて LINESIZE を保持できない場合は、以下ようになります。

- LE for z/OS 1.9 以前のリリースのユーザーの場合

DD SYSOUT= ファイルでは、指定された LINESIZE に一致する新しいレコード・サイズを決定するために、LINESIZE オプションが使用されます。DD DSN= ファイルおよびその他のすべてのファイルでは、UNDEFINEDFILE 条件が発生します。

- LE for z/OS 1.9 より新しいリリースのユーザーの場合

すべてのファイルで、UNDEFINEDFILE 条件が発生します。

例

編集指示ストリーム指向データ伝送を使用して、直接アクセス・ストレージ・デバイス上にデータ・セットを作成する方法が、図 24 に示されています。ファイル SYSIN によって入力ストリームから読み取られるデータには、名前の付いていない 7 文字サブフィールドが入っているフィールド VREC が含まれており、フィールド NUM は情報が入っているこれらのサブフィールドの数を定義します。出力ファイル WORK は、ファイル FREC 全体と、VREC の中で情報の入っているサブフィールドだけをデータ・セットに送ります。

```
//EX7#2 JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
  PEOPLE: PROC OPTIONS(MAIN);
    DCL WORK FILE STREAM OUTPUT,
      1 REC,
      2 FREC,
      3 NAME CHAR(19),
      3 NUM CHAR(1),
      3 PAD CHAR(25),
      2 VREC CHAR(35),
      EOF BIT(1) INIT('0'B),
      IN CHAR(80) DEF REC;
    ON ENDFILE(SYSIN) EOF='1'B;
    OPEN FILE(WORK) LINESIZE(400);
    GET FILE(SYSIN) EDIT(IN)(A(80));
    DO WHILE (~EOF);
      PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
      GET FILE(SYSIN) EDIT(IN)(A(80));
    END;
    CLOSE FILE(WORK);
  END PEOPLE;

/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1))
//GO.SYSIN DD *
R.C.ANDERSON      0 202848 DOCTOR          VICTOR HAZEL
B.F.BENNETT       2 771239 PLUMBER          ELLEN VICTOR JOAN ANN OTTO
R.E.COLE          5 698635 COOK             FRANK CAROL DONALD NORMAN BRENDA
J.F.COOPER        5 418915 LAWYER           ALBERT ERIC JANET
A.J.CORNELL       3 237837 BARBER           GERALD ANNA MARY HAROLD
E.F.FERRIS        4 158636 CARPENTER
/*
```

図 24. ストリーム指向データ伝送によるデータ・セットの作成

233 ページの図 25 は、リスト指示出力を使ってグラフィックスをストリーム・ファイルに書き込むプログラムの例を示しています。なお、この例では、グラフィック・データを印刷できる出力装置があることを想定しています。このプログラムは、従業員レコードを読み取り、特定の地域に在住する従業員を選択します。このプログラムはさらに、住所フィールドを編集して、各住所項目間にグラフィック・ブランクを挿入し、従業員番号、氏名、および住所を印刷します。

```

//EX7#3 JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
% PROCESS GRAPHIC;
XAMPLE1: PROC OPTIONS(MAIN);
    DCL INFILE FILE INPUT RECORD,
        OUTFILE FILE OUTPUT STREAM ENV(GRAPHIC);
/* GRAPHIC OPTION MEANS DELIMITERS WILL BE INSERTED ON OUTPUT FILES. */
    DCL
        1 IN,
            3 EMPNO CHAR(6),
            3 SHIFT1 CHAR(1),
            3 NAME,
                5 LAST G(7),
                5 FIRST G(7),
            3 SHIFT2 CHAR(1),
            3 ADDRESS,
                5 ZIP CHAR(6),
                5 SHIFT3 CHAR(1),
                5 DISTRICT G(5),
                5 CITY G(5),
                5 OTHER G(8),
                5 SHIFT4 CHAR(1);
    DCL EOF BIT(1) INIT('0'B);
    DCL ADDRWK G(20);
    ON ENDFILE (INFILE) EOF = '1'B;
    READ FILE(INFILE) INTO(IN);
    DO WHILE(~EOF);
        DO;
            IF SUBSTR(ZIP,1,3)~='300'
                THEN LEAVE;
            L=0;
            ADDRWK=DISTRICT;
            DO I=1 TO 5;
                IF SUBSTR(DISTRICT,I,1)= < >
                    THEN LEAVE; /* SUBSTR BIF PICKS 3P */
                END; /* THE ITH GRAPHIC CHAR */
                L=L+I+1; /* IN DISTRICT */
            SUBSTR(ADDRWK,L,5)=CITY;
            DO I=1 TO 5;
                IF SUBSTR(CITY,I,1)= < >
                    THEN LEAVE;
            END;
            L=L+I;
            SUBSTR(ADDRWK,L,8)=OTHER;
            PUT FILE(OUTFILE) SKIP /* THIS DATA SET */
            EDIT(EMPNO,IN.LAST,FIRST,ADDRWK) /* REQUIRES UTILITY */
            (A(8),G(7),G(7),X(4),G(20)); /* TO PRINT GRAPHIC */
            /* DATA */
            END; /* END OF NON-ITERATIVE DO */
            READ FILE(INFILE) INTO (IN);
            END; /* END OF DO WHILE(~EOF) */
        END XAMPLE1;
/*
//GO.OUTFILE DD SYSOUT=A,DCB=(RECFM=VB,LRECL=121,BLKSIZE=129)
//GO.INFILE DD *
ABCDEF<
>300099< 3 3 3 3 3 3 >
ABCD <
>300011< 3 3 3 3 >
*/

```

図 25. グラフィック・データのストリーム・ファイルへの書き込み

ストリーム入出力によるデータ・セットへのアクセス

ストリーム指向データ伝送を使ってアクセスするデータ・セットは、ストリーム指向データ伝送で作成されたものである必要はありませんが、CONSECUTIVE 編成データ・セットである必要があり、かつ、データ・セット内の全データが文字の形またはグラフィックの形でなければなりません。入力のための関連ファイルをオープン

ンし、データ・セットに入っているレコードを読み取るか、あるいは出力のためにファイルをオープンし、終わりにレコードを追加してデータ・セットを拡張することができます。

データ・セットにアクセスするには、次のいずれかの方法で、そのデータ・セットを識別する必要があります。

- ENVIRONMENT 属性
- DD_DDNAME 環境変数
- OPEN ステートメントの TITLE オプション

以下に、DD ステートメントに組み込む必須のある情報と、ユーザーが与えることのできるオプション情報の一部について説明します。ただし、ここで述べる内容は、入力ストリームのデータ・セットには当てはまりません。

必須情報

ユーザー・アプリケーションで既存の STREAM ファイルにアクセスするには、PL/I はそのファイルのレコード長を入手する必要があります。レコード長の値は、次のいずれかの入手源から得ることができます。

- OPEN ステートメントの LINESIZE オプション
- ENVIRONMENT 属性の RECSIZE オプション
- DD_DDNAME 環境変数の RECSIZE オプション
- OPEN ステートメントの TITLE オプションの RECSIZE オプション
- PL/I 提供のデフォルト値

既存の OUTPUT ファイルを使用する場合、および RECSIZE の値を提供する場合、PL/I は、230 ページの『ストリーム入出力によるデータ・セットの作成』に説明されているようにレコード長を決定します。

次の場合、PL/I は、INPUT ファイルのデフォルトのレコード長を使用します。

- ファイルが SYSIN で、値が 80 の場合
- ファイルが端末 (stdout: または stderr:) に関連付けられており、値が 120 の場合

レコード・フォーマット

ストリーム指向データ伝送を使ってデータ・セットにアクセスするときには、そのデータ・セットのレコード・フォーマットを知っていなくてもかまいません (ブロック・サイズを指定しなければならない場合を除く)。各 GET ステートメントがそれぞれ個別の数の文字あるいはグラフィックスをデータ・ストリームからプログラムへ転送します。

ユーザーがレコード・フォーマットの情報を与える場合は、その情報はデータ・セットの実際の構造と矛盾しないものである必要があります。例えば、F フォーマット・レコード、600 バイトのレコード・サイズ、および 3600 バイトのブロック・サイズを使って作成したデータ・セットの場合、3600 バイトの最大ブロック・サイズを持った U フォーマット・レコードの場合と同様にそのレコードにアクセスすることができます。ただし、ブロック・サイズ 3500 を指定すると、データは切り捨てられます。

例

図 26 にあるプログラムは、232 ページの図 24 のプログラムで作成したデータ・セットを読み取って、ファイル SYSPRINT を使ってその中に入っているデータをリストします。(SYSPRINT の詳細については、240 ページの『SYSIN ファイルおよび SYSPRINT ファイルの使用法』を参照してください。) 各データのセットは GET ステートメントによって FREC と VREC の 2 つの変数の中に読み込まれます。FREC には常に 45 文字が含まれ、VREC には常に 35 文字が含まれます。GET ステートメントを実行すると、VREC は、式 $7 * \text{NUM}$ が生成する文字数と、総文字数を 35 文字にするのに足りる数のブランクという構成になります。DD ステートメントの DISP パラメーターは単に DISP=OLD とすることもできます。DELETE を省略すると既存のデータ・セットは削除されません。

```
//EX7#5 JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
  PEOPLE: PROC OPTIONS(MAIN);
    DCL WORK FILE STREAM INPUT,
      1 REC,
      2 FREC,
      3 NAME CHAR(19),
      3 NUM CHAR(1),
      3 SERNO CHAR(7),
      3 PROF CHAR(18),
      2 VREC CHAR(35),
    IN CHAR(80) DEF REC,
    EOF BIT(1) INIT('0'B);
  ON ENDFILE(WORK) EOF='1'B;
  OPEN FILE(WORK);
  GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
  DO WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
    GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
  END;
  CLOSE FILE(WORK);
  END PEOPLE;

/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)
```

図 26. ストリーム指向データ伝送によるデータ・セットへのアクセス

ストリーム入出力による PRINT ファイルの使用

オペレーティング・システムにも PL/I 言語にも、出力データのフォーマット設定を簡単に行えるようにするための機能がいくつかあります。オペレーティング・システムで、ユーザーが各レコードの最初のバイトを印刷制御文字として使用することができるようになっています。制御文字は、印刷されずにプリンターに改行や改ページを行わせます。(印刷制御文字の詳細については、248 ページの図 29 および 248 ページの表 17 を参照してください。)

PL/I プログラムでは、PRINT ファイルの使用は、ストリーム指向データ伝送からの印刷出力のレイアウトを制御するのに便利な方法です。コンパイラーは、PAGE、SKIP、LINE オプション、およびフォーマット項目に対応して印刷制御文字を自動的に挿入します。

関連付けられたデータ・セットを直接印刷するつもりでない場合でも、PRINT 属性を任意の STREAM OUTPUT ファイルに適用することができます。PRINT ファイルが直接アクセス・データ・セットに関連付けられていると、印刷制御文字はこのデータ・セットのレイアウトには効力がありませんが、レコード内のデータの一部として現れます。

FB または VB でオープンされた PRINT ファイルは、UNDEFINEDFILE 条件が生じる原因になります。PRINT ファイルは、"A" オプションすなわち FBA または VBA でオープンします。

コンパイラーは、PRINT ファイルで伝送される各レコードの最初のバイトが米国標準規格 (ANS) の印刷制御文字用に予約し、自動的に適切な文字を挿入します。

PRINT ファイルは、次の 5 つの印刷制御文字だけを使用します。

文字 処置

- | | |
|---|------------------------|
| | 1 行空けて (ブランク文字) から印刷する |
| 0 | 2 行空けてから印刷する |
| - | 3 行空けてから印刷する |
| + | 1 行目から印刷する |
| 1 | 改ページする |

コンパイラーは、現行レコードの残りにブランクを埋め込み、次のレコードに適切な制御文字を挿入することによって、PAGE、SKIP、LINE の各オプションやフォーマット項目を処理します。SKIP または LINE に 3 行を超えるスペースが指定されている場合は、コンパイラーは、適切な制御文字を使って必要な数のブランク・レコードを挿入し、必要なスペーシングを行います。印刷制御オプションやフォーマット項目がない場合は、レコードがフルになると、コンパイラーは、次のレコードの最初のバイトにブランク文字 (1 行のスペース) を挿入します。

PRINT ファイルの伝送先が端末の場合は、出力フォーマットを指定しない限り、PAGE、SKIP、LINE の各オプションに 3 行を超えるスキップをさせることはありません。

印刷する行の長さの制御

ユーザーの PL/I プログラムあるいは DD ステートメントにレコード長を指定する (ENVIRONMENT 属性) か、あるいは OPEN ステートメントに行サイズを指定する (LINESIZE オプション) ことによって、PRINT ファイルで作成された印刷行の長さを制限することができます。レコード長には印刷制御文字用の余分のバイト数を含める必要があります。すなわち、レコード長は印刷される行の長さより 1 バイト (V フォーマット・レコードの場合は、5 バイト) 長くなければなりません。ユーザーが LINESIZE オプションに指定した値は、印刷される行の文字数を参照します。コンパイラーは印刷制御文字を追加します。

レコードをブロック化しても、PRINT ファイルで生成される出力の外観に影響はありませんが、直接アクセス装置上でファイルがデータ・セットと関連付けられていれば、補助記憶域をより効率よく使えるようになります。なお、LINESIZE オプションを指定する場合は、行サイズとブロック・サイズとの互換性を確認する必要があります。F フォーマットのレコードの場合、ブロック・サイズは正確に (行サイ

ズ +1) の倍数でなくてはならず、V フォーマットのレコードの場合、ブロック・サイズは行サイズよりも最低 9 バイト大きくなければなりません。

ファイルをいったんクローズし、新しい行サイズでもう一度オープンすれば、実行中に PRINT ファイルの行サイズを変更できますが、PRINT ファイルを使って直接アクセス装置上にデータ・セットを作成する場合は、注意が必要です。ファイルを初めてオープンしたとき、データ・セットに設定したレコード・フォーマットは変更できません。OPEN ステートメントに指定した行サイズと、既に設定されているレコード長が矛盾すると、UNDEFINEDFILE 条件が発生します。このような矛盾が生じないように、使用する予定の最大行サイズより最低 9 バイト大きいブロック・サイズを指定して V フォーマット・レコードを指定するか、最初の OPEN ステートメントで最大行サイズを必ず指定するようにしてください。(プリンターに送る予定の出力は、直接プリンターへ送ろうとする場合でも、UNIT= を使ってプリンターを指定しない限り、一時的に直接アクセス装置に保管することができます。)

PRINT ファイルは 120 文字のデフォルトの行サイズを持っています。したがって、PRINT ファイルのレコード・フォーマットを指定する必要はありません。また、他の情報がない場合には、コンパイラーは V フォーマットのレコードであると見なします。完全なデフォルト値は、次のとおりです。

BLKSIZE=129

LRECL=125

RECFM=VBA.

例: 238 ページの図 27 は、PRINT ファイルと、ストリーム指向データ伝送ステートメントのオプションを使って、テーブルをフォーマット設定して、それを後で印刷できるように直接アクセス装置に書き込む方法を例示しています。このテーブルは、6' 間隔の、0° から 359° 54' までの角度の正弦から成ります。

ENDPAGE ON ユニット内のステートメントによって、各ページの下にページ番号が挿入され、次ページの見出しがセットアップされます。

このプログラムで作成されたデータ・セットを定義する DD ステートメントには、レコード・フォーマットに関する情報は含まれていません。ファイル TABLE をオープンするステートメントに指定されたファイル宣言および行サイズから、コンパイラーは次のように推測します。

レコード・フォーマット =

V (PRINT ファイルのデフォルト値)

レコード・サイズ =

98 (行サイズ + 1 バイト (印刷制御文字用) + 4 バイト (レコード制御フィールド用))

ブロック・サイズ =

102 (レコード長 + 4 バイト (ブロック制御フィールド用))

255 ページの図 31 のプログラムは、レコード単位データ伝送を使って、238 ページの図 27 のプログラムが作成するテーブルを印刷します。

```

%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

SINE: PROC OPTIONS(MAIN);
  DCL TABLE      FILE STREAM OUTPUT PRINT;
  DCL DEG          FIXED DEC(5,1) INIT(0); /* INIT(0) FOR ENDPAGE */
  DCL MIN          FIXED DEC(3,1);
  DCL PGNO         FIXED DEC(2)  INIT(0);
  DCL ONCODE       BUILTIN;

  ON ERROR
  BEGIN;
    ON ERROR SYSTEM;
    DISPLAY ('ONCODE = ' || ONCODE);
  END;

  ON ENDPAGE(TABLE)
  BEGIN;
    DCL I;
    IF PGNO ^= 0 THEN
      PUT FILE(TABLE) EDIT ('PAGE',PGNO)
        (LINE(55),COL(80),A,F(3));
    IF DEG ^= 360 THEN
      DO;
        PUT FILE(TABLE) PAGE EDIT ('NATURAL SINES') (A);
        IF PGNO ^= 0 THEN
          PUT FILE(TABLE) EDIT ((I DO I = 0 TO 54 BY 6))
            (SKIP(3),10 F(9));

        PGNO = PGNO + 1;
      END;
    ELSE
      PUT FILE(TABLE) PAGE;
  END;

  OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
  SIGNAL ENDPAGE(TABLE);

  PUT FILE(TABLE) EDIT
    ((DEG,(SIND(DEG+MIN) DO MIN = 0 TO .9 BY .1) DO DEG = 0 TO 359))
    (SKIP(2), 5 (COL(1), F(3), 10 F(9,4) ));
  PUT FILE(TABLE) SKIP(52);
END SINE;

```

図 27. ストリーム・データ伝送による印刷ファイルの作成： 255 ページの図 31 の例では、結果ファイルが印刷されます。

タブ制御テーブルの指定変更

PRINT ファイルへのデータ指示出力およびリスト指示出力は、事前設定されているタブ位置に合わせて配置されます。タブ・テーブルの宣言方法の例については、169 ページの図 14 および 240 ページの図 28 の例を参照してください。テーブル内にフィールドを定義する方法は、次のとおりです。

OFFSET OF TAB COUNT:

ハーフワード 2 進整数で表された『タブ・カウント』のオフセット。これは、使用するタブ数を示すためのフィールドです。

PAGESIZE:

デフォルトのページ・サイズを定義するハーフワード 2 進整数。ページ・サイズは、ストリーム出力時、および PLIDUMP データ・セットへのダンプ出力時に使われる値です。

LINESIZE:

デフォルトの行サイズを定義するハーフワード 2 進整数

PAGELength:

端末での印刷で使われるデフォルトのページ長を定義するハーフワード 2 進整数

FILLERS:

3 ハーフワードの 2 進整数。将来の利用のために予約済み

TAB COUNT:

テーブル内のタブ位置の数 (最大 255) を定義するハーフワード 2 進整数。タブ・カウント = 0 の場合は、指定されたタブ位置はすべて無視されます。

Tab1-Tabn:

印刷行内のタブ位置を定義する n 個のハーフワード 2 進整数。最初のタブ位置には 1 という番号がつき、最大値は 255 です。各タブの値は、テーブル内でその前にあるタブの値より大きくなければなりません。さもなければその値は無視されます。印刷される出力の最初のデータ・フィールドは、次の有効なタブ位置から始まります。

リンケージ・エディターを使って PLITABS への外部参照を解決すれば、ユーザー・プログラムで PL/I のデフォルトのタブ設定を変更することができます。この外部参照を解決するには、PLITABS という名前のテーブルを前述のフォーマットで与えます。

このタブ・テーブルを供給するには、ソース・プログラムに PLITABS という名前の PL/I 構造体を組み込みます。この構造体を、MAIN プロシージャ内、または MAIN プロシージャとリンクされるプログラム内で、STATIC EXTERNAL として宣言する必要があります。240 ページの図 28 は、PL/I 構造体の例です。この例では、位置 30、60、90 に 3 つのタブを設定し、ページ・サイズおよび行サイズにはデフォルト値が使われています。TAB1 は行上に印刷される 2 番目の項目の位置を識別することに注意してください。行上の第 1 項目は常に左マージンから始まります。構造体の第 1 項目は、NO_OF_TABS フィールドへのオフセットです。なお、FILL1、FILL2、FILL3 は、オフセットの値を -6 で調整すれば省略できます。

```
DCL 1 PLITABS STATIC EXT,  
  2 (OFFSET INIT(14),  
    PAGESIZE INIT(60),  
    LINESIZE INIT(120),  
    PAGELENGTH INIT(0),  
    FILL1 INIT(0),  
    FILL2 INIT(0),  
    FILL3 INIT(0),  
    NO_OF_TABS INIT(3),  
    TAB1 INIT(30),  
    TAB2 INIT(60),  
    TAB3 INIT(90)) FIXED BIN(15,0);
```

図 28. 事前設定済みのタブ設定を変更する場合の PL/I 構造体 PLITABS

SYSIN ファイルおよび SYSPRINT ファイルの使用方法

ユーザー・プログラムに FILE オプションを指定せずに GET ステートメントをコーディングした場合は、コンパイラーは SYSIN というファイル名を挿入します。また、FILE オプションを指定せずに PUT ステートメントをコーディングする場合は、コンパイラーは SYSPRINT という名前を挿入します。

SYSPRINT を宣言しないと、コンパイラーは、通常のデフォルト属性の他に、属性 PRINT をファイルに与えます。属性の完全なセットは、次のとおりです。

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

SYSPRINT は PRINT ファイルの一種であるので、コンパイラーはデフォルトの行サイズ (120 文字) や V フォーマット・レコードも与えます。ユーザーは最小限の情報のみを該当する DD ステートメントに与えるだけで済みます。クラス A のシステム出力装置がプリンターであるという通常の規則を使用するユーザーのシステムの場合には、次のステートメントで十分です。

```
//SYSPRINT DD SYSOUT=A
```

注: SYSIN および SYSPRINT は、初期化中のユーザー出口で設定されます。

SYSIN および SYSPRINT の IBM 提供のデフォルト値は両方とも端末に送られます。

コンパイラーによって SYSPRINT に与えられた属性は、ファイルを明示的に宣言またはオープンすることによって指定変更することができます。SYSPRINT と z/OS 言語環境プログラム メッセージ・ファイル・オプション間の対話の詳細については、「z/OS 言語環境プログラム プログラミングの手引き」を参照してください。

コンパイラーは入力ファイル SYSIN のために特別な属性は指定しません。その宣言を行わない場合は、デフォルトの属性のみを受け取ります。SYSIN に関連するデータ・セットは通常は入力ストリーム内にあります。それが入力ストリーム内にない場合には、完全な DD 情報を与えなくてはなりません。

SYSPRINT に関する詳細については 172 ページの『SYSPRINT の考慮事項』を参照してください。

端末からの入力の制御

次に挙げることを行えば、ユーザーの PL/I プログラムで、入力ファイルへのデータを端末から入力することができます。

1. CONSECUTIVE 環境オプションを指定して、明示的または暗黙的に入力ファイルを宣言する (ストリーム・ファイルはすべてこの条件を満たしています)。
2. 入力ファイルを端末に割り振る

ユーザーは通常、標準のデフォルト入力ファイル `SYSDIN` を使用することができます。このファイルはストリーム・ファイルであり、端末に割り振ることができるからです。

ストリーム・ファイルへの入力をユーザーに促すプロンプトは、コロン (:) で示されます。プログラムで `GET` ステートメントが実行されると、その度にコロンが表示されます。また、`GET` ステートメントが実行されると、システムは次の行に移動します。ユーザーはそこに必要なデータを入力することができます。`GET` ステートメントの実行を完了するのに十分なデータの入っていない行を入力すると、コロンの前に正符号の付いた別のプロンプト (+:) が表示されます。

継続させたい行の最後にハイフンを付け加えると、2 行以上のデータが入力されるまで、ユーザー・プログラムへのデータ伝送を遅らせることができます。

ユーザーのプログラムで、ユーザーに入力を求めるプロンプトを出す出力ステートメントを組み込んだ場合、ユーザー自身のプロンプトをコロンで終了して、初期システム・プロンプトが出ないようにすることができます。例えば、`GET` ステートメントの前に次のような `PUT` ステートメントを置くことができます。

```
PUT SKIP LIST('ENTER NEXT ITEM:');
```

次の `GET` ステートメントでシステム・プロンプトが表示されないようにするには、ユーザー独自のプロンプトは次の条件を満たしている必要があります。

1. 独自のプロンプトは、リスト指示か編集指示のどちらかでなければなりません。また、リスト指示の場合は、`PRINT` ファイルにあてたものでなければなりません。
2. プロンプトを伝送するファイルは、端末に割り振る必要があります。端末でファイルをコピーするだけの場合は、システム・プロンプトが表示されないようにすることはできません。

データのフォーマット

端末で入力するデータは、次に挙げる場合を除き、バッチ・モードのストリーム入力データとまったく同じフォーマットでなければなりません。

- 入力のための単純化された句読法: 別々の入力項目を別々の行に入力する場合、間にブランクあるいはコンマを入力する必要はなく、コンパイラーは各行の終わりにコンマを挿入します。

例えば、次のステートメントに応答するとします。

```
GET LIST(I,J,K);
```

端末での対話は、次のようになります。

```
:  
1  
+:2  
+:3
```

各項目の後ろに改行が付きます。これは、次に示すものとまったく同じです。

```
:  
1,2,3
```

ある項目を次の行まで続けたい場合は、1 行目の最後に継続文字を入力します。継続文字がなければ、GET LIST ステートメントまたは GET DATA ステートメントではコンマが挿入され、GET EDIT ステートメントでは埋め込みが行われます (以下参照)。

- **GET EDIT** に関する自動埋め込み: GET EDIT ステートメントに関しては、入力行の終わりにブランクを入力する必要はありません。ユーザーが入力する項目には、正しい長さになるまで埋め込みが行われます。

例えば次の PL/I ステートメントの場合 :

```
GET EDIT(NAME)(A(15));
```

次の 5 文字を入力することができます。

```
SMITH
```

上の文字の直後に改行を入力します。プログラムが 15 文字からなるストリングを受け取れるように、この項目には 10 個のブランクが埋め込まれます。ある項目を 2 番目の行または後続行に続けたければ、最終行を除くすべての行の終わりに連結文字を入れなければなりません。そうでなければ伝送される最初の行には、埋め込み処理が行われ、完結したデータ項目として扱われます。

- **SKIP** オプションあるいはフォーマット項目: GET ステートメント内の SKIP は、プログラムに対してまだ入力されていないデータを無視するように指示します。n が 1 より大きい SKIP(n) はすべて、SKIP(1) と見なされます。SKIP(1) は、現在行上にある未使用データはすべて無視されることを意味すると見なされます。

ストリーム・ファイルおよびレコード・ファイル

ストリーム・ファイルとレコード・ファイルを両方とも端末に割り振ることができます。しかし、この場合、レコード・ファイル用のプロンプトは表示されません。端末に複数のファイルを割り振ったとき、そのうちの 1 つ以上がレコード・ファイルであると、ファイル出力は必ずしも同期化されるとは限りません。なお、端末へのデータ伝送および端末からのデータ伝送の順序が、対応する PL/I の入出力ステートメントの実行順序と等しくなる保証はありません。

また、端末からのレコード・ファイルの入力は、TCAM の制限により、大文字で受け取られます。問題を避けるため、可能であればストリーム・ファイルをお使いください。

大文字と小文字

ストリーム・ファイルでは、文字ストリングは小文字または大文字で入力した通りにプログラムに送られます。一方、レコード・ファイルでは、文字はすべて大文字になります。

ファイルの終わり

桁 1 および桁 2 に /* があり他の文字はない行の /* は、ファイル・マークとして扱われます。すなわち、これらの文字は ENDFILE 条件を発生させます。

GET ステートメントの COPY オプション

GET ステートメントは COPY オプションを指定することができますが、入力ファイルとともに COPY ファイルが端末に割り振られた場合は、データのコピーは印刷されません。

端末への出力の制御

端末を使って、次の条件を両方とも満たす PL/I ファイルのデータを入手することができます。

1. CONSECUTIVE 環境オプションによって、明示的あるいは暗黙的に宣言されている。ストリーム・ファイルはすべてこの条件を満たしている。
2. 端末に割り振られている

標準の印刷ファイル SYSPRINT は、通常、これらの条件を 2 つとも満たしています。

PRINT ファイルのフォーマット

SYSPRINT または他の PRINT ファイルからのデータは、一般的に、端末でページの形にフォーマットされません。PAGE オプション、LINE オプション、およびフォーマット項目については、常に、3 行がスキップされます。通常、ENDPAGE 条件が発生することはありません。SKIP(n) では、n の値が 3 より大きい場合も、3 行だけスキップされます。SKIP(0) はバックスペースによってインプリメントされるので、バックスペース機能の備わっていない端末で使用してはなりません。

PRINT ファイルは、ユーザー・プログラムにタブ制御テーブルを挿入することによって、ページの形にフォーマット設定することができます。このテーブルは PLITABS という名前であればならず、また、テーブルの内容は 238 ページの『タブ制御テーブルの指定変更』で説明されています。ユーザーは、エレメント PAGELENGTH を必要なページ長、すなわち印刷可能な最大行数で表された、各ページを印刷する用紙の長さに初期化しなければなりません。エレメント PAGESIZE は、各ページに印刷しようとする実際の行数に初期化しなければなりません。PAGESIZE に指定した行数をページに印刷し終わると、ENDPAGE 条件が生じますが、これに対する標準システム動作は、PAGELENGTH から PAGESIZE を引いた値に等しい行数をスキップし、その後で次ページの印刷を開始することです。標準以外のレイアウトの場合は、PLITABS 内の他のエレメントを、169 ページの図 14 で示されている値に初期化しなければなりません。PLITABS を使って、リスト指示およびリスト指示出力のタブ位置を変更することもできます。ILC アプリケーションで改ページをフォーマット設定する必要があるとき、SYSPRINT の代わりに

PLITABS を使用することができます。改ページを制御するには、PAGESIZE を 32767 に設定して、PUT PAGE ステートメントを使用します。

端末の中にはタブ設定機能を備えたものもありますが、リスト指示およびリスト指示出力では、常に、ブランク文字を伝送することによってタブが設定されます。

ストリーム・ファイルおよびレコード・ファイル

ストリーム・ファイルとレコード・ファイルを両方とも端末に割り振ることができます。ただし、端末に複数のファイルを割り振った場合、そのうちの 1 つ以上がレコード・ファイルであると、ファイル出力は必ずしも同期化されるとは限りません。プログラムと端末との間でやりとりされる時のデータの順序が、それに対応する PL/I 入出力ステートメントが実行されるのと同じ順序になるという保証はありません。また、TCAM の制約により、端末でのレコード・ファイルへの出力はすべて大文字で印刷されます。そのため、可能であればストリーム・ファイルをご使用になることをお勧めします。

大文字と小文字

ストリーム・ファイルの場合、端末で表示される文字は、端末が表示することが可能でありさえすれば、プログラムに入っているとおりに表示されます。例えば、IBM 327 x 端末では、大文字と小文字は変換されず、そのままの形で表示されます。一方、レコード・ファイルでは、文字はすべて大文字に変換されます。プログラムが ASIS オペランド付きの EDIT コマンドを使って作成されている場合、または、そのプログラムが端末から小文字を読み取っている場合は、プログラムの変数または定数に小文字を入れることができます。

PUT EDIT コマンドの出力

PUT EDIT コマンドをからの端末への出力のフォーマットは、行モード TPUT であり、『フィールドの始まり』 および 『フィールドの末尾』 の文字は、画面上ではブランクとして表示されます。

レコード単位データ伝送の使用

PL/I は、RECORD 属性を持つさまざまなタイプのデータ・セットをサポートしています (250 ページの表 19 参照)。このセクションでは、連続データ・セットの使用法について説明します。

245 ページの表 16 は、レコード単位データ伝送を使って、連続データ・セットを作成したり、連続データ・セットにアクセスする場合に使用できるステートメントとオプションをリストしています。

表 16. 連続データ・セットの作成と連続データ・セットへのアクセスで利用できるステートメントとオプション

ファイル宣言 ¹	有効ステートメント、 ² および 必須オプション	指定できるその他の オプション
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference); LOCATE 基底付き変数 FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	

注:

1. 完全なファイル宣言には、属性 FILE、RECORD、および ENVIRONMENT が組み込まれています。
2. ステートメント READ FILE (file-reference) は、有効なステートメントであり、また READ FILE(file-reference) IGNORE (1) と同等のものです。

レコード・フォーマットの指定

レコード・フォーマット情報を与える場合には、その情報はデータ・セットの実際の構造に矛盾しないものである必要があります。例えば、レコード・フォーマットが FB、レコード・サイズが 600 バイト、ブロック・サイズが 3600 バイトのデータ・セットを作成すれば、最大ブロック・サイズが 3600 バイトの U フォーマット・レコードであるかのように、そのレコードにアクセスすることができます。ブロック・サイズを 3500 バイトに指定すると、データが切り捨てられます。

レコード入出力の使用によるファイルの定義

属性を次のように指定してファイルを宣言すれば、レコード単位データ伝送で使うファイルを定義できます。

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED
      ENVIRONMENT(options);
```

デフォルト・ファイル属性は 209 ページの表 13 のとおりです。ファイル属性については、「PL/I 言語解説書」で説明されています。ENVIRONMENT 属性のオプションについては、以下に説明しています。

ENVIRONMENT オプションの指定

連続データ・セットに適用できる ENVIRONMENT オプションは、次のとおりです。

```
F|FB|FS|FBS|V|VB|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
```

```
CONSECUTIVE or ORGANIZATION(CONSECUTIVE)
CTLASA|CTL360
LEAVE|REREAD
```

SCALARVARYING 全体のオプションについては、208 ページの『ENVIRONMENT 属性での特性の指定』で説明されています。SCALARVARYING の後のオプションについては以下に説明します。

どのオプションを指定する必要があるか、どれがオプションで、どれがデフォルト・オプションであるかを知るには、209 ページの表 13 を参照してください。

CONSECUTIVE

CONSECUTIVE オプションは、連続データ・セット編成のファイルを定義します。詳細は本章および 203 ページの『データ・セットの編成』に記述してあります。

▶—CONSECUTIVE—▶

CONSECUTIVE はデフォルト値です。

ORGANIZATION(CONSECUTIVE)

ファイルを連続データ・セットに関連付けることを指定します。 ORGANIZATION オプションの詳細は、217 ページの『ORGANIZATION オプション』に説明されています。

連続ファイルは、ネイティブ・データ・セットでも、VSAM データ・セットでも構いません。

CTLASA|CTL360

印刷制御機構オプション CTLASA および CTL360 は、連続データ・セットに関連付けられた OUTPUT ファイルだけに適用されます。この 2 つのオプションは、レコードの最初の文字を制御文字として解釈するように指定します。



CTLASA オプションは、米国標準規格垂直紙送り位置決め文字 (American National Standard Vertical Carriage Positioning Character) または米国標準規格ポケット選択文字 (American National Standard Pocket Select Character)(レベル 1) を指定するためのオプションです。また、CTL360 オプションは、IBM のマシン・コード制御文字を指定するためのオプションです。

248 ページの図 29 にリストされている米国標準規格の制御文字は、関連付けられたレコードが印刷、穿孔される前に、指定の処置を実行するための文字です。

なお、マシン・コード制御文字は、装置タイプによって異なります。プリンター用の IBM マシン・コード制御文字は、248 ページの表 17 にリストしてあります。

コード	処置
	1 行空けて (ブランク・コード) から印刷する
0	2 行空けてから印刷する
-	3 行空けてから印刷する
+	1 行目から印刷する
1	チャンネル 1 にスキップする
2	チャンネル 2 にスキップする
3	チャンネル 3 にスキップする
4	チャンネル 4 にスキップする
5	チャンネル 5 にスキップする
6	チャンネル 6 にスキップする
7	チャンネル 7 にスキップする
8	チャンネル 8 にスキップする
9	チャンネル 9 にスキップする
A	チャンネル 10 にスキップする
B	チャンネル 11 にスキップする
C	チャンネル 12 にスキップする
V	スタッカー 1 を選択する
W	スタッカー 2 を選択する

図 29. 米国標準規格の印刷およびカード穿孔制御文字 (CTLASA)

表 17. IBM マシン・コード印刷制御文字 (CTL360)

印刷してから 実行	処置	直ちに処置 (印刷は行わない)
コード・バイト		コード・バイト
00000001	印刷のみ (スペースなし)	—
00001001	スペース 1 行	00001011
00010001	スペース 2 行	00010011
00011001	スペース 3 行	00011011
10001001	チャンネル 1 にスキップする	10001011
10010001	チャンネル 2 にスキップする	10010011
10011001	チャンネル 3 にスキップする	10011011
10100001	チャンネル 4 にスキップする	10100011
10101001	チャンネル 5 にスキップする	10101011
10110001	チャンネル 6 にスキップする	10110011
10111001	チャンネル 7 にスキップする	10111011
11000001	チャンネル 8 にスキップする	11000011
11001001	チャンネル 9 にスキップする	11001011
11010001	チャンネル 10 にスキップする	11010011
11011001	チャンネル 11 にスキップする	11011011
11100001	チャンネル 12 にスキップする	11100011

LEAVE REREAD

磁気テープの処理オプション LEAVE および REREAD を使用すると、磁気テープ・ボリュームの終わりに達したとき、または磁気テープ・ボリューム上のデータ・セットが閉じたときにとるアクションを指定できます。LEAVE オプションは、

テープが巻き戻されないようにします。REREAD オプションは、テープを巻き戻して、データ・セットの再処理を可能にします。これらのいずれかを指定しない場合、ボリュームの終わりまたはデータ・セットのクローズ時のアクションは、関連 DD ステートメントの DISP パラメーターによって制御されます。



同じプログラムでデータ・セットを最初は順方向に読み取るか書き込み、次に逆方向で読み取る場合は、ファイルが閉じられたとき (または、マルチボリューム・データ・セットの場合、ボリューム切り替えが生じるとき) にボリュームが巻き戻されないように LEAVE オプションを指定してください。

LEAVE および REREAD オプションの影響は、表 18に要約されています。

表 18. LEAVE および REREAD オプションの影響

ENVIRONMENT		処置
option	DISP パラメーター	
REREAD	—	データ・セットを再処理する 現行ボリュームの位置を決め ます。BACKWARDS ファイル の位置は、データ・セット の物理終了に変更されます。 現行ボリュームをデータ・セ ットの論理終了に位置決めし ます。BACKWARDS ファイル の位置は、データ・セット の物理開始に変更されます。 データ・セットの終わりに ボリュームの位置を決める。
LEAVE	—	
REREAD も LEAVE も指定しない	PASS	
	DELETE KEEP, CATLG, UNCATLG	現行ボリュームを巻き戻す。 現行ボリュームを 巻き戻し・アンロードする。

レコード入出力によるデータ・セットの作成

連続データ・セットを作成するには、SEQUENTIAL OUTPUT の関連ファイルを開く必要があります。WRITE あるいは LOCATE ステートメントを使用してレコードを書くことができます。245 ページの表 16 は、連続データ・セットを作成するためのステートメントとオプションを示しています。

データ・セットを作成する際、DD ステートメント内で、オペレーティング・システムに対してそのデータ・セットを識別しなければなりません。次に、250 ページの表 19 に要約されている、DD ステートメントに入れる必要のある必須情報と、ユーザーが与えることのできるいくつかのオプション情報について説明します。

表 19. レコード入出力による連続データ・セットの作成: DD ステートメントの必須パラメーター

ストレージ装置	必要な場合	指定すべき事項	パラメーター
全種	常時	出力装置	UNIT= または SYSOUT= または VOLUME=REF=
		ブロック・サイズ ¹	DCB=(BLKSIZE=...
直接アクセス のみ	常時	必要な ストレージ・スペース	SPACE=
直接アクセス	他のジョブ・ステップで使用される が、ジョブの終了時に必要とされ ないデータ・セット	後処理	DISP=
	ジョブ終了後も保持する データ・セット	後処理	DISP=
		データ・セットの名前	DSNAME=
		ボリューム通し番号	VOLUME=SER= または VOLUME=REF=
	特定の装置上に存在する データ・セット		

注: ¹ または、ENVIRONMENT 属性を用いれば、PL/I プログラム内でブロック・サイズを指定することもできます。

必須情報

連続データ・セットを作成する場合は、次の事項を指定する必要があります。

- PL/I ファイルと関連付けるデータ・セットの名前。なお、連続編成のデータ・セットは、どのタイプの装置上にも存在できます。
- レコード長。レコード長は、ENVIRONMENT 属性、DD_DDNAME 環境変数、または OPEN ステートメントの TITLE オプションの RECSIZE オプションを使って指定することができます。

端末装置 (stdout: または stderr:) に関連付けられたファイルでは、RECSIZE オプションが指定されていないければ、PL/I はデフォルトのレコード長の 120 を使用します。

レコード入出力によるデータ・セットへのアクセスと更新

連続データ・セットを作成し終われば、順次入力、順次出力、または、直接アクセス装置上のデータ・セットの場合は、更新を行うために、その連続データ・セットにアクセスするためのファイルをオープンすることができます。253 ページの図 30 は、連続データ・セットにアクセスし、それを更新するプログラムの例です。出力用のファイルをオープンして、その終わりにレコードを追加してデータ・セットを拡張するには、DD ステートメント内に DISP=MOD を指定しなければなりません。それを指定しないと、そのデータ・セットは上書きされます。更新用のファイルをオープンしても、その既存順にしかレコードを更新することはできず、レコードを挿入したければ、新たにデータ・セットを作成しなければなりません。245 ページの表 16 は、連続データ・セットにアクセスして、値を更新するためのステートメントとオプションを示しています。

SEQUENTIAL UPDATE ファイルで連続データ・セットにアクセスするには、**READ** ステートメントを使ってレコードを取り出してから、**REWRITE** ステートメントでそれを更新しなければなりません。しかし、検索される各レコードの再書き込みは必要ありません。**REWRITE** ステートメントは、常に、最後に読み取られたレコードを更新します。

次のような場合を考慮します。

```
READ FILE(F) INTO(A);
.
.
.
READ FILE(F) INTO(B);
.
.
.
REWRITE FILE(F) FROM(A);
```

REWRITE ステートメントによって、2 つ目の **READ** ステートメントで読み取ったレコードが更新されます。最初のステートメントで読み取ったレコードは、2 つ目の **READ** ステートメントが実行されると再書き込みできません。

データ・セットにアクセスするには、**DD** ステートメントでそのデータ・セットをオペレーティング・システムに識別する必要があります。表 20 は、連続データ・セットにアクセスするのに必要な **DD** ステートメントのパラメーターを要約しています。

表 20. レコード入出力による連続データ・セットへのアクセス: **DD** ステートメントの必須パラメーター

パラメーター	指定すべき事項	必要な場合
DSNAME=	データ・セットの名前	常時
DISP=	データ・セットの後処理	
UNIT= または VOLUME=REF=	入力装置	データ・セットがカタログされていない場合 (すべての装置)
VOLUME=SER=	ボリューム通し番号	データ・セットがカタログされていない場合 (直接アクセス)
DCB=(BLKSIZE=	ブロック・サイズ ¹	データ・セットに標準ラベルが付いていない 場合

注: ¹ または、ENVIRONMENT 属性を用いれば、PL/I プログラム内でブロック・サイズを指定することもできます。

次に、**DD** ステートメントに入れるべき必須情報と、ユーザーが与えることのできるオプション情報のうちのいくつかについて説明します。ただし、ここで述べる内容は、入力ストリームのデータ・セットには当てはまりません。

必須情報

データ・セットがカタログされている場合は、**DD** ステートメントに次の情報だけを指定する必要があります。

- データ・セットの名前 (DSNAME パラメーター)。オペレーティング・システムは、システム・カタログ内でそのデータ・セットを記述した情報を検索し、また、必要があれば、オペレーターに、そのデータ・セットが入っているボリュームをマウントするよう要求します。
- データ・セットが存在することの確認 (DISP パラメーター)。データ・セットの終わりにレコードを追加して、データ・セットを拡張するために出力用のデータ・セットをオープンするには、DISP=MOD とコーディングします。それを行わないと、出力のためのデータ・セットのオープンが行われたときに、そのデータ・セットは上書きされます。

データ・セットがカタログされていない場合は、データ・セットを読み取る装置、直接アクセス装置、さらに、データ・セットが入っているボリュームの通し番号 (UNIT および VOLUME パラメーター) を指定する必要があります。

連続データ・セットの例

連続データ・セットを作成し、それにアクセスする方法については、253 ページの図 30 のプログラムに示されています。プログラムは、2 つのデータ・セットの内容を入力ストリーム内で組み合わせてから、それを新規データ・セット `&&TEMP;` に書き込みますが、元のデータ・セットにはおののおの、EBCDIC 照合順序に並べられた 15 バイトの固定長レコードが入っています。INPUT1 と INPUT2 の 2 つの入力ファイルはデフォルト属性 `BUFFERED` を持ち、関連データ・セットからレコードをそれぞれのバッファに読み取るのに位置指定モードが使われます。バッファ内の基底付き変数へのアクセスは、ファイルがクローズされた後は行わないでください。

```

//EXAMPLE JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

MERGE: PROC OPTIONS(MAIN);
    DCL (INPUT1,                                /* FIRST INPUT FILE */
        INPUT2,                                /* SECOND INPUT FILE */
        OUT ) FILE RECORD SEQUENTIAL; /* RESULTING MERGED FILE*/
    DCL SYSPRINT FILE PRINT; /* NORMAL PRINT FILE */

    DCL INPUT1_EOF BIT(1) INIT('0'B); /* EOF FLAG FOR INPUT1 */
    DCL INPUT2_EOF BIT(1) INIT('0'B); /* EOF FLAG FOR INPUT2 */
    DCL OUT_EOF BIT(1) INIT('0'B); /* EOF FLAG FOR OUT */
    DCL TRUE BIT(1) INIT('1'B); /* CONSTANT TRUE */
    DCL FALSE BIT(1) INIT('0'B); /* CONSTANT FALSE */

    DCL ITEM1 CHAR(15) BASED(A); /* ITEM FROM INPUT1 */
    DCL ITEM2 CHAR(15) BASED(B); /* ITEM FROM INPUT2 */
    DCL INPUT_LINE CHAR(15); /* INPUT FOR READ INTO */
    DCL A POINTER; /* POINTER VAR */
    DCL B POINTER; /* POINTER VAR */

    ON ENDFILE(INPUT1) INPUT1_EOF = TRUE;
    ON ENDFILE(INPUT2) INPUT2_EOF = TRUE;
    ON ENDFILE(OUT) OUT_EOF = TRUE;

    OPEN FILE(INPUT1) INPUT,
        FILE(INPUT2) INPUT,
        FILE(OUT) OUTPUT;

    READ FILE(INPUT1) SET(A); /* PRIMING READ */
    READ FILE(INPUT2) SET(B);

    DO WHILE ((INPUT1_EOF = FALSE) & (INPUT2_EOF = FALSE));
        IF ITEM1 > ITEM2 THEN
            DO;
                WRITE FILE(OUT) FROM(ITEM2);
                PUT FILE(SYSPRINT) SKIP EDIT('1>2', ITEM1, ITEM2)
                    (A(5),A,A);
                READ FILE(INPUT2) SET(B);
            END;
        ELSE
            DO;
                WRITE FILE(OUT) FROM(ITEM1);
                PUT FILE(SYSPRINT) SKIP EDIT('1<2', ITEM1, ITEM2)
                    (A(5),A,A);
                READ FILE(INPUT1) SET(A);
            END;
        END;
    END;

```

図 30. 連続データ・セットのマージ、ソート、作成と連続データ・セットへのアクセス (1/2)

```

DO WHILE (INPUT1_EOF = FALSE);          /* INPUT2 IS EXHAUSTED */
  WRITE FILE(OUT) FROM(ITEM1);
  PUT FILE(SYSPRINT) SKIP EDIT('1', ITEM1) (A(2),A);
  READ FILE(INPUT1) SET(A);
END;

DO WHILE (INPUT2_EOF = FALSE);          /* INPUT1 IS EXHAUSTED */
  WRITE FILE(OUT) FROM(ITEM2);
  PUT FILE(SYSPRINT) SKIP EDIT('2', ITEM2) (A(2),A);
  READ FILE(INPUT2) SET(B);
END;

CLOSE FILE(INPUT1), FILE(INPUT2), FILE(OUT);
PUT FILE(SYSPRINT) PAGE;
OPEN FILE(OUT) SEQUENTIAL INPUT;

READ FILE(OUT) INTO(INPUT_LINE);        /* DISPLAY OUT FILE */
DO WHILE (OUT_EOF = FALSE);
  PUT FILE(SYSPRINT) SKIP EDIT(INPUT_LINE) (A);
  READ FILE(OUT) INTO(INPUT_LINE);
END;
CLOSE FILE(OUT);

END MERGE;
/*
//GO.INPUT1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.INPUT2 DD *
BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,
//          DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))

```

図 30. 連続データ・セットのマージ、ソート、作成と連続データ・セットへのアクセス (2/2)

255 ページの図 31 のプログラムは、レコード単位データ伝送を使って、238 ページの図 27 のプログラムが作成するテーブルを印刷します。

```

%PROCESS INT F(I) AG A(F) OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

PRT: PROC OPTIONS(MAIN);
  DCL TABLE      FILE RECORD INPUT SEQUENTIAL;
  DCL PRINTER     FILE RECORD OUTPUT SEQL
                  ENV(V BLKSIZE(102) CTLASA);
  DCL LINE        CHAR(94) VAR;

  DCL TABLE_EOF  BIT(1) INIT('0'B);      /* EOF FLAG FOR TABLE */
  DCL TRUE        BIT(1) INIT('1'B);      /* CONSTANT TRUE       */
  DCL FALSE       BIT(1) INIT('0'B);      /* CONSTANT FALSE      */

  ON ENDFILE(TABLE) TABLE_EOF = TRUE;

  OPEN FILE(TABLE),
    FILE(PRINTER);

  READ FILE(TABLE) INTO(LINE);              /* PRIMING READ        */

  DO WHILE (TABLE_EOF = FALSE);
    WRITE FILE(PRINTER) FROM(LINE);
    READ FILE(TABLE) INTO(LINE);
  END;

  CLOSE FILE(TABLE),
    FILE(PRINTER);
END PRT;

```

図 31. レコード単位データ伝送の印刷

第 9 章 領域データ・セットの定義と使用

この章では、領域データ・セットの編成、データ伝送ステートメント、および領域データ・セットを定義する ENVIRONMENT オプションについて述べます。次に、領域編成のタイプごとに、領域データ・セットの作成方法と領域データ・セットへのアクセス方法について述べます。

領域編成のデータ・セットは 2 つの領域に分かれますが、それぞれの領域は領域番号で識別され、またそのおのにおに、領域編成のタイプに応じて、単数または複数のレコードを入れることができます。これらの領域には、ゼロから始まる連続番号が付けられ、レコードはデータ伝送ステートメント内に領域番号と一緒にキーを指定することによってアクセスすることができます。

領域データ・セットは、直接アクセス装置に限られます。

データ・セットを領域編成にすれば、データ・セット内でのレコードの物理配置を制御することができます。また、特定アプリケーションへのアクセス時間を最適化することができます。このような最適化は、連続編成または索引編成では使用することはできません。それは、これらの編成では、昇順キー値に応じて、連続レコードが厳密な物理順序または論理順序で書き込まれるからです。これらの方式のいずれも直接アクセス・ストレージ・デバイスの特性を十分には利用しません。

領域データ・セットは、連続データ・セットまたは索引付きデータ・セットと似た方法で、昇順の領域番号順にレコードを提示することによって作成することができます。別の方法として、直接アクセスを用いることができ、その場合、レコードはランダム順序で提示し、それらを事前にフォーマット設定された領域に直接挿入します。領域データ・セットを作成した後は、INPUT または UPDATE だけでなく SEQUENTIAL または DIRECT 属性を持ったファイルを使用してそのデータ・セットにアクセスすることができます。データ・セットが SEQUENTIAL INPUT ファイルまたは SEQUENTIAL UPDATE ファイルと関連付けていれば、領域番号またはキーを指定する必要はありません。ファイルに DIRECT 属性があれば、無作為にレコードを検索、追加、削除、および置換することができます。

領域データ・セット内のレコードは、有効なデータが入っている実際のレコードであるか、またはダミー・レコードのいずれかです。

領域編成での、他のタイプのデータ・セット編成よりも大きな利点は、ユーザーがレコードの相対配置を制御できることにあります。適切なプログラミングにより、装置の能力およびアプリケーションの要件に合わせレコード・アクセスを最適化することができます。

領域データ・セットの直接アクセスは、索引付きデータ・セットのアクセスより早く行うことができますが、領域データ・セットには、順次処理はレコードをランダム順に提示することがあるという欠点があります。順次検索の順序は必ずしもレコードが提示される順序ではなく、また必ずしも相対キー値と関連する必要もありません。

表 21 は、領域データ・セットを作成し、また領域データ・セットへアクセスできるためのデータ伝送ステートメントとオプションをリストしています。

表 21. 領域データ・セットの作成と領域データ・セットへのアクセスで利用できるステートメントとオプション

ファイル 宣言 ¹	有効ステートメント、 ² および 必須オプション	指定できるその他の オプション
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE 基底付き変数 FROM(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE ³	READ FILE(file-reference) INTO(reference);	KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	

表 21. 領域データ・セットの作成と領域データ・セットへのアクセスで利用できるステートメントとオプション (続き)

ファイル宣言 ¹	有効ステートメント、 ² および 必須オプション	指定できるその他の オプション
DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); DELETE FILE(file-reference) KEY(expression);	

注:

- 1. 完全なファイル宣言には属性 FILE、 RECORD、および ENVIRONMENT が含まれています。オプション KEY、KEYFROM、あるいは KEYTO のいずれかを使用する場合は、属性 KEYED も含めなくてはなりません。
- 2. ステートメント READ FILE(file-reference); は、ステートメント READ FILE(file-reference) IGNORE(1); と同等です。
- 3. 新たにデータ・セットを作成するときには、ファイルに UPDATE 属性があってはなりません。

領域データ・セット用のファイルの定義

順次領域データ・セットを定義するには、次の属性を指定したファイルを宣言を使用します。

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED
      [KEYED]
      ENVIRONMENT(options);
```

REGIONAL(1) データ・セットの BUFFERED と UNBUFFERED は同じ扱いになるので、ENV オプションでどちらのオプションを指定してもかまいません。例えば、UNBUFFERED が指定されていても、SEQUENTIAL UNBUFFERED ファイルの REWRITE には FROM オプションの必要がなく、OUTPUT SEQUENTIAL データ・セットに対する LOCATE ステートメントが許可されます。

直接領域データ・セットを定義するには、次の属性を指定したファイル宣言を使用します。

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      DIRECT
      ENVIRONMENT(options);
```

デフォルト・ファイル属性は 209 ページの表 13のとおりです。ファイル属性については、「PL/I 言語解説書」で説明されています。ENVIRONMENT 属性のオプションについては、以下に説明します。

ENVIRONMENT オプションの指定

領域データ・セットに適用できる ENVIRONMENT オプションは、次のとおりです。

```
REGIONAL({1})  
F  
RECSIZE(record-length)  
BLKSIZE(block-size)  
SCALARVARYING
```

REGIONAL オプション

領域編成のファイルを定義するには、REGIONAL オプションを使用します。

▶▶—REGIONAL—(—1—)—————▶▶

1 REGIONAL(1) を指定

REGIONAL(1)

データ・セットに、記録済みキーのない F フォーマットのレコードが入っていることを表します。データ・セット内の各領域にはただ 1 つのレコードが入っており、したがって、各領域番号はデータ・セット内の相対レコードに対応しています (すなわち、領域番号はデータ・セットの始めから 0 で始まります)。

REGIONAL(1) データ・セットには記録済みキーが 1 つもありませんが、REGIONAL(1) DIRECT INPUT ファイルまたは REGIONAL(1) DIRECT UPDATE ファイルを使えば、記録済みキーのないデータ・セットも処理できます。

RECSIZE(record-length)

BLKSIZE(block-size)

RECSIZE と BLKSIZE の両方を指定する場合は、それぞれに同じ値を指定する必要があります。

重複した領域番号がなく、また大半の領域がいっぱいになる (データ・セット内の無駄なスペースが削減される) アプリケーションの場合には、REGIONAL(1) 編成が最適です。

REGIONAL データ・セットでのキーの使用

キーには、記録済みキーとソース・キーの 2 種類があります。記録済みキーは、レコードを識別するためにデータ・セット内の各キーの直前に付く文字ストリングです。その長さは 255 文字を超えることはできません。ソース・キーは、ステートメントが参照するレコードを識別するためにデータ伝送ステートメントの KEY オプションまたは KEYFROM オプション内に現れる式の文字値です。領域データ・セット内のレコードにアクセスする場合は、ソース・キーは領域番号を与え、同時に、記録済みキーも与えることができます。

索引付きデータ・セット用のキーと異なり、領域データ・セットの記録済みキーはレコード内に埋め込まれることはありません。

REGIONAL(1) データ・セットの使用

REGIONAL(1) データ・セットでは、記録済みキーがないため、領域番号は特定のレコードを識別する唯一のキーとしての役割を果たします。ソース・キーの文字値は、16777215 を超えてはならない符号なし 10 進整数を表さなければなりません (許容される実際のレコード数は、レコード・サイズ、装置容量、および個々のアクセス方式での制限をどのように組み合わせるかによって、これより小さくなる場合があります。ただし、固定長レコードを含む直接 regional(1) ファイルでは、相対トラック・アドレッシングでアドレスできるトラック数の最大値は 65,536 です)。領域番号がこの数値を超える場合、領域番号はモジュロ 16777216 として扱われ、例えば、16777226 は 10 として扱われます。0 から 9 までの文字とブランク文字だけが、ソース・キー内では有効です。先行ブランクはゼロとして解釈されます。領域番号に埋め込まれたブランクを使用することはできません。したがって、埋め込まれたブランクが最初に見つかった時点で、その領域番号は終了します。ソース・キー中に 8 文字を超えて存在する場合は、右端の 8 文字のみが領域番号として使用され、8 文字未満の場合は、左側にブランク (ゼロとして解釈される) が挿入されます。

ダミー・レコード

REGIONAL(1) データ・セットには、有効なデータが入っている実際のレコード、またはダミー・レコードのいずれかが入っています。REGIONAL(1) データ・セット内のダミー・レコードは、レコードの最初のバイトの定数 (8)'1'B で識別されます。このようなダミー・レコードは、データ・セットの作成時かまたはレコードの削除時にデータ・セット内に挿入されますが、データ・セットが読み取られるときに無視されません。ユーザーの PL/I プログラムはそれらを認識するように作成されなくてはなりません。ダミー・レコードは、有効データで置き換えることができます。ダミー・レコードの最初のバイトに (8)'1'B を挿入した場合は、検索されないダミー・レコードを持っているデータ・セットにファイルをコピーすると、そのレコードは失われることがあります。

REGIONAL(1) データ・セットの作成

REGIONAL(1) データ・セットは、順次アクセスか直接アクセスのどちらかを使って作成することができます。258 ページの表 21 は、領域データ・セットを作成するためのステートメントとオプションを示しています。

SEQUENTIAL OUTPUT ファイルを使ってデータ・セットを作成するとき、ファイルをオープンすると、データ・セット上のすべてのトラックが消去され、各トラックの先頭に、そのトラック上で使用できるスペースの大きさを記録する容量レコードが書き込まれることになります。レコードは領域番号の昇順で提示しなくてはならず、そのシーケンスから省略された領域はダミー・レコードによって埋められます。このシーケンスにエラーがあると、あるいは、重複キーを提示すると、KEY 条件が発生します。ファイルがクローズされるときに、現行エクステンツの終わりのスペースにはダミー・レコードが埋め込まれます。

DIRECT OUTPUT ファイルを使ってデータ・セットを作成すると、そのデータ・セットに割り振られた 1 次エクステンツ全体が、ファイルのオープン時にダミー・レコードで埋められます。レコードをランダム順で提示することができますが、レコードを重複して提示した場合、既存レコードは上書きされます。

順次作成の場合、データ・セットは最大 15 までのエクステントを持つことができ、また複数のボリューム上にまたがっていてもかまいません。直接作成の場合、データ・セットは 1 つしかエクステントを持つことはできず、したがって 1 つのボリューム上にしか存在することはできません。

例

REGIONAL(1) データ・セットの作成例が、263 ページの図 32 に示してあります。この例のデータ・セットは、電話番号とその電話番号を割り当てる加入者の氏名リストです。電話番号は領域データ・セット内の領域番号と対応しており、各領域番号が占める領域には加入者名のデータが入っています。

```

//EX9    JOB
//STEP1  EXEC IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN DD *
CRR1:    PROC OPTIONS(MAIN);
/* CREATING A REGIONAL(1) DATA SET - PHONE DIRECTORY */

DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
DCL SYSIN FILE INPUT RECORD;
DCL SYSIN_REC BIT(1) INIT('1'B);
DCL 1 CARD,
      2 NAME CHAR(20),
      2 NUMBER CHAR( 2),
      2 CARD_1 CHAR(58);
DCL IOFIELD CHAR(20);

ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
OPEN FILE(NOS);
READ FILE(SYSIN) INTO(CARD);

DO WHILE(SYSIN_REC);
  IOFIELD = NAME;
  WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
  PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
  READ FILE(SYSIN) INTO(CARD);
END;

CLOSE FILE(NOS);
END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS DD DSN=MYID.NOS,UNIT=SYSDA,SPACE=(20,100),
// DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.       28
CHEESNAME,L.       11
CORY,G.            36
ELLIOTT,D.         85
FIGGINS,E.S.       43
HARVEY,C.D.W.      25
HASTINGS,G.M.      31
KENDALL,J.G.       24
LANCASTER,W.R.     64
MILES,R.           23
NEWMAN,M.W.        40
PITT,W.H.          55
ROLF,D.E.          14
SHEERS,C.D.        21
SURCLIFFE,M.       42
TAYLOR,G.C.        47
WILTON,L.W.        44
WINSTONE,E.M.      37
/*

```

図 32. REGIONAL(1) データ・セットの作成

REGIONAL(1) データ・セットへのアクセスと更新

いったん REGIONAL(1) データ・セットを作成すると、SEQUENTIAL INPUT および SEQUENTIAL UPDATE、または DIRECT INPUT および DIRECT UPDATE の

ためにそのデータ・セットにアクセスするファイルをオープンすることができます。既存データ・セットを上書きする場合にのみそれを OUTPUT のためにオープンすることができます。258 ページの表 21 は、領域データ・セットにアクセスするためのステートメントとオプションを示しています。

順次アクセス

REGIONAL(1) データ・セットを処理する SEQUENTIAL ファイルをオープンするには、INPUT 属性または UPDATE 属性を使用します。データ伝送ステートメントには KEY オプションを指定してはなりません、KEYTO オプションは使えるため、ファイルは KEYED 属性を持つことができます。KEYTO オプションで参照されるターゲット文字ストリングが 8 文字を超えると、返される値 (8 文字の領域番号) の左側にブランクが埋め込まれます。また、ターゲット・ストリングが 8 文字より短い場合は、返された値の左側が切り捨てられます。

順次アクセスは、領域番号の昇順で行われます。ダミー・レコードも実際のレコードも、レコードはすべて検索されるため、ユーザーの PL/I プログラムがダミー・レコードを認識するようにしておく必要があります。

REGIONAL(1) データ・セットで順次入力を使用すれば、領域番号の昇順ですべてのレコードを読み取ることができます。また、順次更新では、順番に各レコードを読み取り、もう一度書き込むことが可能です。

REGIONAL(1) データ・セットにアクセスする SEQUENTIAL UPDATE ファイルに対する READ ステートメントと REWRITE ステートメント間の関係を決める規則は、連続データ・セットの場合と同じです。連続データ・セットの詳細については、227 ページの『第 8 章 連続データ・セットの定義と使用』を参照してください。

直接アクセス

REGIONAL(1) データ・セットを処理する直接ファイルをオープンするには、INPUT 属性または UPDATE 属性を使用します。すべてのデータ伝送ステートメントはソース・キーを持っていなければならない、DIRECT 属性は KEYED 属性を暗黙指定します。

次の規則に従って REGIONAL(1) データ・セット内のレコードを検索、追加、削除、または置換するには、DIRECT UPDATE ファイルを使用します。

- 検索** ダミー・レコードも実際のレコードもすべて検索されます。したがって、ユーザー・プログラムがダミー・レコードを認識できなくてはなりません。
- 追加** WRITE ステートメントは、ソース・キーで指定された領域の既存レコード (実際のレコードまたはダミー・レコード) を新規レコードで置き換えます。
- 削除** DELETE ステートメントでソース・キーを使って指定したレコードは、ダミー・レコードに変換されます。
- 置換** REWRITE ステートメントでソース・キーを使って指定したレコードは、ダミー・レコードであれ実際のレコードであれ変換されます。

例

REGIONAL(1) データ・セットの更新は、266 ページの図 33 に示されています。このプログラムはデータ・セットを更新し、データ・セットの内容をリストします。

別のレコードや更新済みレコードを書き込む前に、その領域内の既存レコードをテストし、それがダミー・レコードであるかどうかを確認します。これは、たとえダミーでなくても **WRITE** ステートメントは **REGIONAL(1)** データ・セット中の既存レコードを上書きできるためです。同様に、データ・セットの内容を順番に読み取ったり印刷したりする際に、各レコードがテストされ、ダミー・レコードは印刷されません。

```

//EX10    JOB
//STEP2   EXEC  IBMZCBG,PARM.PLI='NOP,MAR(1,72)',PARM.BIND='LIST'
//PLI.SYSIN DD *
ACR1: PROC OPTIONS(MAIN);
  /* UPDATING A REGIONAL(1) DATA SET - PHONE DIRECTORY      */
  DCL NOS FILE RECORD KEYED ENV(REGIONAL(1));
  DCL SYSIN FILE INPUT RECORD;
  DCL (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
  DCL 1  CARD,
        2  NAME  CHAR(20),
        2  (NEWNO,OLDNO) CHAR( 2),
        2  CARD_1 CHAR( 1),
        2  CODE  CHAR( 1),
        2  CARD_2 CHAR(54);
  DCL IOFIELD CHAR(20);
  DCL BYTE  CHAR(1) DEF IOFIELD;

  ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
  OPEN FILE (NOS) DIRECT UPDATE;
  READ FILE(SYSIN) INTO(CARD);

  DO WHILE(SYSIN_REC);
    SELECT(CODE);
      WHEN('A','C') DO;
        IF CODE = 'C' THEN
          DELETE FILE(NOS) KEY(OLDNO);
          READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
          IF UNSPEC(BYTE) = (8)'1'B
            THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
          ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
        END;
      WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
      OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
    END;
    READ FILE(SYSIN) INTO(CARD);
  END;

  CLOSE FILE(SYSIN),FILE(NOS);
  PUT FILE(SYSPRINT) PAGE;
  OPEN FILE(NOS) SEQUENTIAL INPUT;
  ON ENDFILE(NOS) NOS_REC = '0'B;
  READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
  DO WHILE(NOS_REC);
    IF UNSPEC(BYTE) ^= (8)'1'B
      THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD) (A(2),X(3),A);
    PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
    READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
  END;
  CLOSE FILE(NOS);
END ACR1;
/*

```

図 33. REGIONAL(1) データ・セットの更新 (1/2)

```
//GO.NOS DD DSN=J44PLI.NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.      5640 C
GOODFELLOW,D.T.  89   A
MILES,R.         23   D
HARVEY,C.D.W.    29   A
BARTLETT,S.G.    13   A
CORY,G.          36   D
READ,K.M.        01   A
PITT,W.H.        55
ROLF,D.F.        14   D
ELLIOTT,D.       4285 C
HASTINGS,G.M.    31   D
BRAMLEY,O.H.     4928 C
/*
```

図 33. REGIONAL(1) データ・セットの更新 (2/2)

領域データ・セットの作成時、および領域データ・セットへのアクセス時の必須情報

領域データ・セットを作成するには、PL/I プログラム内か、またはそのデータ・セットを定義する DD ステートメント内で、オペレーティング・システムに特定の情報を提供しなければなりません。次に、必須情報を示し、ユーザーが与えることのできるいくつかのオプションの情報について説明します。

領域データ・セットを作成するには、次の情報を与える必要があります。

- データ・セットを書き込む装置 (DD ステートメントの UNIT パラメーターまたは VOLUME パラメーター)。
- ブロック・サイズ: ブロック・サイズは、PL/I プログラム (ENVIRONMENT 属性の BLKSIZE オプションで) または DD ステートメント (BLKSIZE サブパラメーター) 中で指定することができます。レコード長を指定しないと、非ブロック化レコードがデフォルトになり、レコード長はそのブロック・サイズから決められます。レコード長を指定する場合は、ブロック・サイズと等しくなければなりません。

データ・セットを保持したい場合 (すなわち、ジョブ終了時にオペレーティング・システムによってデータ・セットが削除されないようにするには)、DD ステートメントを使って、そのデータ・セット名とデータ・セットをどのように処理するのかを指定する必要があります (DSNAME パラメーターおよび DISP パラメーター)。後のステップでデータ・セットを使用したいが、ジョブが終了すればそのデータ・セットは必要なくなるのであれば、DISP パラメーターだけで十分です。

データ・セットを特定の直接アクセス装置に保管したければ、DD ステートメント内でボリューム通し番号を指示しなければなりません (VOLUME パラメーターの SER サブパラメーターまたは REF サブパラメーター)。とっておきたいデータ・セット用の通し番号を指定しないと、オペレーティング・システムが番号を割り振り、それをオペレーターに通知してから、ユーザーのプログラム・リスト上にその番号を印刷します。領域データ・セットを作成するのに DD ステートメント内に必

要な必須パラメーターについては、表 22 に要約してあります。また 269 ページの表 23 は必要な DCB サブパラメーターを列挙しています。DCB サブパラメーターの詳細については、「z/OS JCL User's Guide」を参照してください。

領域データ・セットは、システム出力 (SYSOUT) 装置上に置くことはできません。

DCB パラメーターで DSORG パラメーターを指定する場合は、DSORG=DA とコーディングすることによって、データ・セット編成を直接アクセスと指定しなければなりません。領域データ・セット用の DD ステートメント内では、DUMMY パラメーターまたは DSN=NULLFILE パラメーターを指定することはできません。DSORG=DA を使用すると、メッセージ IEC225I が出されることがあります。このメッセージは安全であり、無視できます。

表 22. 領域データ・セットの作成: DD ステートメントの必須パラメーター

パラメーター	指定すべき事項	必要な場合
UNIT= または VOLUME=REF=	出力装置 ¹	常時
SPACE=	必要なストレージ・スペース ²	
DCB=	データ制御ブロック情報 269 ページの表 23 を参照	
DISP=	後処理	データ・セットが別のジョブ・ステップで使われるが、別のジョブでは必要ない場合。
DISP=	後処理	ジョブ終了後も保持するデータ・セット
DSNAME=	データ・セットの名前	
VOLUME=SER= または VOLUME=REF=	ボリューム通し番号	特定のボリューム上に存在するデータ・セット

¹ 領域データ・セットは、直接アクセス装置に限られます。

² 順次アクセスでは、データ・セットは複数のボリューム上に存在できる最大 15 個のエクステンションを持つことができます。直接アクセスによる作成の場合、データ・セットはエクステンションを 1 つだけ持つことができます。

領域データ・セットにアクセスするには、DD ステートメントを使ってそのデータ・セットをオペレーティング・システムに識別する必要があります。次項に、DD ステートメントに入れなければならない最低限の情報を示します。この情報は 269 ページの表 24 に要約されています。

データ・セットがカタログされている場合は、DD ステートメントに次の情報だけを指定する必要があります。

- データ・セットの名前 (DSNAME パラメーター)。オペレーティング・システムは、システム・カタログ内で該当のデータ・セットを記述する情報を検出し、また、必要があれば、オペレーターに、そのデータ・セットが入っているボリュームをマウントするよう要求します。
- データ・セットが存在することの確認 (DISP パラメーター)。

データ・セットがカタログされていない場合は、さらにデータ・セットを読み取る装置、およびデータ・セットが入っているボリュームの通し番号 (UNIT パラメーターおよび VOLUME パラメーター) を指定する必要があります。

順次更新用にマルチボリューム領域データ・セットをオープンすると、最初のボリューム終了時に ENDFILE 条件が発生します。

表 23. 領域データ・セットの DCB サブパラメーター

サブパラメーター	指定するもの	必要な場合
RECFM=F	レコード・フォーマット ¹	常時
BLKSIZE=	ブロック・サイズ ¹	
DSORG=DA	データ・セットの編成	
¹ あるいは、ENVIRONMENT 属性内でブロック・サイズを指定することもできます。		

表 24. 領域データ・セットへのアクセス: DD ステートメントの必須パラメーター

パラメーター	指定すべき事項	必要な場合
DSNAME=	データ・セットの名前	常時
DISP=	データ・セットの後処理	
UNIT= または VOLUME=REF=	入力装置	データ・セットがカタログされていない場合
VOLUME=SER=	ボリューム通し番号	

第 10 章 VSAM データ・セットの定義と使用

この章では、レコード単位データ伝送用の VSAM (仮想記憶アクセス方式) 編成、VSAM ENVIRONMENT オプション、他の PL/I データ・セット編成との互換性、および PL/I がサポートする 3 つのタイプの VSAM データ・セット (入力順、キー順、および相対レコード) をロードしそれにアクセスするのに使用するステートメントについて述べます。そしてこの章の終わりには、VSAM データ・セットを作成してそれにアクセスするのに必要な、PL/I ステートメント、アクセス方式サービス・コマンド、および JCL ステートメントの一連の例を示しています。

Enterprise PL/I は、ISAM データ・セットをサポートしていません。

VSAM の各種機能、VSAM データ・セットと索引の構造、それらをアクセス方式サービスが定義する方法、および必須 JCL ステートメントに関する詳細は、ご使用のシステムの VSAM 資料を参照してください。

VSAM データ・セットの使用

VSAM データ・セットでのプログラムの実行

VSAM データ・セットにアクセスするプログラムを実行するには、次の事項があらかじめ分かっている必要があります。

- VSAM データ・セット名
- PL/I ファイルの名前
- データ・セットを他のユーザーと共用するかどうか

その後で、そのデータ・セットにアクセスするのに必要な DD ステートメントを作成することができます。

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

例えば、ファイルが PL1FILE という名前で、データ・セットは VSAMDS という名前であって、そのデータ・セットの排他制御を行いたければ、次のように入力します。

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

データ・セットを他のユーザーと共用するには、DISP=SHR を使用します。

Enterprise PL/I は、ISAM データ・セットをサポートしていません。

データ・セットに使用する VSAM バッファ数制御して VSAM のパフォーマンスを最適化するには、VSAM 資料を参照してください。

代替索引パスとファイルのペア化

代替索引を使用するには、基本データ・セット/代替索引のペアとユーザーの PL/I ファイルを関連付ける DD ステートメントの DSNAME パラメーターに *path* の名

前を指定するだけです。代替索引を使用する前に、処理についての制約事項に気を付ける必要があります。制限事項は 277 ページの表 26 にまとめてあります。

PL1FILE という PL/I ファイルがあり、PERSALPH という代替索引パスがあると想定すると、必要な DD ステートメントは次のようになります。

```
//PL1FILE DD DSN=PERSALPH,DISP=OLD
```

VSAM 編成

PL/Iは、次の 3 つのタイプの VSAM データ・セットをサポートします。

- キー順データ・セット (KSDS)
- 入力順データ・セット (ESDS)
- 相対レコード・データ・セット (RRDS)

上記のデータ・セットはそれぞれ、PL/I の索引付きデータ・セット編成、連続データ・セット編成、および領域データ・セット編成とほぼ対応しています。これらはすべて順序を付けられ、中にあるレコードに関連付けられたキーを持つことができます。3 つのタイプ全部において、順次アクセスとキーによるアクセスの両方を行うことができます。

キー順データ・セットだけがその論理レコードの一部としてキーを持つことができますが、入力順データ・セット (相対バイト・アドレスを使用) および相対レコード・データ・セット (相対レコード番号を使用) でもキー順アクセスを行うことができます。

VSAM データ・セットはすべて直接アクセス・ストレージ・デバイスに保持され、そのデータ・セットを使用するには、仮想記憶オペレーティング・システムが必要です。

VSAM データ・セットの物理編成は、他のアクセス方式で使用するものとは異なります。VSAM ではブロック化の概念は適用されず、また、相対レコード・データ・セットの場合を除き、レコードは固定長でなくてもかまいません。VSAM 編成のデータ・セットでは、データ項目は制御インターバルに並べられ、さらにそれが制御域内に並べられます。処理に備えて、制御インターバル内のデータ項目は論理レコード内に並べられます。制御インターバルは、1 つ以上の論理レコードを収容することができ、1つの論理レコードが複数の制御インターバルにスパンしてもかまいません。VSAM では、ブロック化因数とレコード長に対する配慮は大きく軽減されますが、レコードは、最大指定サイズを超えてはなりません。VSAM では制御インターバルへアクセスすることもできますが、このタイプのアクセスは PL/I ではサポートされていません。

VSAM データ・セットは、2 つの索引タイプを持つことができます。それは、基本と代替です。基本索引はデータ・セットを定義するときに設定される KSDS に対する索引で、常に存在し、また KSDS で可能な唯一の索引です。KSDS または ESDS に対して、1 つ以上の代替索引を使用することができます。ESDS に代替索引を定義すると、一般に ESDS を KSDS として使用できます。KSDS の代替索引は、基本索引とは異なる論理レコードのフィールドをキー・フィールドとして使用できます。代替索引は重複キーが使用できる非固有、重複キーが使用できない固有のいずれかにすることができます。基本索引には、重複キーがあってはなりません。

代替索引を持つデータ・セット内の変更は、今後使用するすべての索引に反映されなければなりません。この活動は、索引アップグレード として知られ、データ・セット内の索引アップグレード・セット 内の任意の索引について、VSAM が行います。(KSDS の場合は、基本索引は常に索引アップグレード・セットのメンバーです。) しかし、基本索引または固有代替索引に重複キーを作成する、データ・セット内の変更は避けてください。

VSAM データ・セットを初めて使用する前に、アクセス方式サービスの DEFINE コマンドを使ってそのデータ・セットをシステムに対して定義しなければなりません。これは、データ・セットのタイプ、構造、および必要スペースを完全に定義するのに使用できるコマンドです。このコマンドはまた、データ・セットが KSDS であるかまたは 1 つ以上の代替索引を持つ場合に、そのデータ・セットの索引 (キーの長さや位置も一緒に) と索引アップグレード・セットも定義します。したがって、VSAM データ・セットはアクセス方式サービスによって「作成」されます。

初期データを新たに作成した VSAM データ・セットに書き込む操作を、本書ではロードする という表現で示しています。

3 つのタイプのデータ・セットは、次の各目的別に使用します。

- 入力順データ・セット は、主として作成された順序 (またはその逆の順序) でアクセスするデータの場合に使用します。
- キー順データ・セット は、通常どおりにレコード内のキーを介してレコードにアクセスする場合に使用します (例えば、レコードにアクセスするのに部品番号が使用される在庫制御ファイル)。
- 相対レコード・データ・セット は、各項目が特定の番号を持っており、その番号によって相対レコードに通常アクセスするようなデータに使用します (例えば、各番号に関連したレコードを持った電話システム)。

どのタイプの VSAM データ・セット内のレコードにも、キーを使用して直接、または順次に (逆方向または順方向に) アクセスできます。2 方向の組み合わせを使用することもできます。キーで開始点を選択し、その点から順方向あるいは逆方向に読み取りを行います。

キー順および入力順データ・セットに代替索引 を作成できます。作成後は、多くのシーケンスを使用して、または多くのキーのうちのいずれかを使用して、ユーザー・データにアクセスできます。例えば、従業員番号順に保持されているか、索引を付けられているデータ・セットを使用して、代替索引 内に名前ですべてに索引を付けることができます。それから、英字順、逆英字順、または名前を直接キーとして使用して、そのデータ・セットにアクセスできます。従業員番号の同じ種類の組み合わせによって、データ・セットにアクセスすることもできます。

274 ページの表 25 に、同一データをどのようにこの 3 つの異なったタイプの VSAM データ・セット内に保持できるかを示し、それぞれの利点と欠点を示しています。

表 25. VSAM データ・セットのタイプと利点

データ・セット・タイプ	ロード方式	読み取り方式	更新方式	利点と欠点
キー順	順次に並んだ索引または固有でなければならない基本索引。	基本索引でレコードのキーを指定した KEYED。 任意の索引を逆方向または順方向に SEQUENTIAL。 キーで位置決めした後に、逆方向または順方向の順次読み取り。	任意の索引内で固有キーを指定して KEYED。 固有キーで位置決めした後に SEQUENTIAL。 レコードの削除が許可される。 レコードの追加が許可される。	利点 ：完全アクセスおよび更新。 欠点 ：ロード前にレコードが基本索引順にソートされていない。 使用 ：アクセスがキーと関連する場合に使用。
入力順	順次（順方向のみ）。 各レコードの RBA を入手して、キーとして使用可能。	SEQUENTIAL 逆方向または順方向。 RBA を使用して KEYED。 キーで位置決めをした後で、逆方向または順方向に順次。	新しいレコードは終わりにのみ。 既存レコード長は変更不可。 レコードの削除不可。	利点 ：簡単迅速に作成。 固有索引は不要。 欠点 ：限定された更新機能。 使用 ：データが主として順次式でアクセスされる場合。
相対レコード	スロット 1 から順次。 スロット番号を指定して KEYED。 キーで位置決めした後で、順次書き込み。	番号をキーとして指定して KEYED。 空きレコードを省いて、順方向または逆方向へ順次。	指定したスロットから開始し、次のスロットへ順次。 番号をキーとして指定して KEYED。 レコードの削除が許可される。 空きスロットへのレコード追加が許可される。	利点 ：番号によるレコードへの高速アクセス。 欠点 ：構造が番号付けに拘束される。 固定長レコード。 使用 ：レコードが番号によってアクセスされる場合に使用。

VSAM データ・セットのキー

VSAM データ・セットはすべて、それぞれのレコードに関連したキーを持つことができます。キー順データ・セット、および代替索引 を介して入力順データ・セットにアクセスする場合は、そのキーは論理レコード内の定義されたフィールドになります。入力順データ・セットの場合、キーは、レコードの相対バイト・アドレス (RBA) になります。相対レコード・データ・セットの場合、キーは、相対レコード番号 になります。

索引付き VSAM データ・セットのキー

キー順データ・セット、および代替索引 を介して入力順データ・セットにアクセスする場合のキーは、データ・セットに記録された論理レコードの一部になります。データ・セットの作成時に、キーの長さと位置を定義します。

KEY、KEYFROM、および KEYTO オプションでキーを参照する方法は、「PL/I 言語解説書」の第 12 章の『KEY(expression) オプション』、『KEYFROM(expression) オプション』、『KEYTO(reference) オプション』に説明があります。

相対バイト・アドレス (RBA)

相対バイト・アドレスを使えば、KEYED SEQUENTIAL ファイルに関連した ESDS 上でキーによるアクセスを使用することができます。RBA、つまりキーは、長さが 4 文字の文字ストリングで、その値は VSAM によって定義されます。PL/I では RBA を構成または操作することはできません。しかし、データ・セット内のレコードの相対位置を判別するためにそれらの値を比較することはできます。RBA は通常は印刷できません。

レコードの RBA を求めるには、データ・セットをロードまたは拡張するときに WRITE ステートメント上で、またはデータ・セットを読み取るときに READ ステートメント上で KEYTO オプションを使用します。これで、READ ステートメントまたは REWRITE ステートメントの KEY オプションで、上記のどちらかの方法で取得した RBA を後になってから使用することができます。

WRITE ステートメントの KEYFROM オプションでは、RBA を使用しないでください。

VSAM では、相対バイト・アドレスを KSDS に対するキーとして使用できますが、この使用は PL/I ではサポートされていません。

相対レコード番号

RRDS 内のレコードの識別は、1 から始まり、その後 1 つのレコードにつき 1 ずつ増えていく相対レコード番号で行います。この相対レコード番号は、データ・セットへのキー順アクセスで、キーとして使用することができます。

相対レコード番号として使用されるキーは、長さ 8 の文字ストリングです。KEY オプションや KEYFROM オプションで使用するソース・キーの文字値は、符号なし整数を表していなければなりません。ソース・キーが 8 文字の長さでなければ、左側で切り捨てられるか、あるいは、ブランク (ゼロと解釈される) が埋められます。KEYTO オプションが戻す値は、先行ゼロを抑止された長さ 8 の文字ストリングです。

データ・セット・タイプの選択

プログラムの設計時には、まず使用するデータ・セットのタイプを決める必要があります。使用できる VSAM データ・セットには 3 つのタイプがあり、非 VSAM データ・セットには 5 つのタイプがあります。VSAM データ・セットには、他のタイプのデータ・セットにあるすべての機能に加えて、VSAM だけで使用できる機能が備わっています。VSAM はパフォーマンスの点では、通常、他のデータ・セットと互角であり、上回ることも多くあります。ただし、VSAM の方が、機能が誤用された場合に、パフォーマンスが低下する可能性が高くなります。

217 ページの表 14 に与えられているデータ・セットの 8 つのタイプすべての比較は有用です。しかし、大規模インストール・システムでのデータ・セット・タイプの選択での多数の要因は本書の範囲外です。

各種 VSAM データ・セットのうちのいずれかを選択するときには、個々のデータを必要とする最も一般的な順序に基づいて選択する必要があります。次に、ユーザーが必要な機能を確保するための、データ・セットと索引の組み合わせかたとして、参考のための手順の案を示します。

1. データ・タイプを決めて、それにアクセスする方法を決めます。
 - a. 主として順次の場合 - ESDS を優先
 - b. 主としてキーによる場合 - KSDS を優先
 - c. 主として番号による場合 - RRDS を優先
2. データ・セットのロード方法を決めます。 KSDS はキー・シーケンスでロードしなければならないことに注意してください。したがって、一部のアプリケーションでは、代替索引 パスを持つ ESDS をより実用的な代替として使用することもできます。
3. 代替索引 パスを使用してアクセスする必要があるかどうかを決めます。これらは、KSDS と ESDS でのみサポートされます。代替索引 パスが必要な場合は、代替索引 が固有キーを持つか、非固有キーを持つかを決めます。非固有キーを使用する場合は、キー処理の制限が生じます。しかし、今後のレコードすべてに固有キーを使用するということは実用的ではありません。固有キー用に作成した索引に非固有キーを持つレコードを挿入しようとすると、エラーが発生します。
4. ユーザーが必要とするデータ・セットとパスが決まったら、ユーザーが想定している操作がサポートされているかどうか確認してください。 277 ページの図 34 を参考にしてください。

ダミーの VSAM データ・セットにはアクセスしないでください。未定義ファイルがあることを示すエラー・メッセージを受け取るためです。

283 ページの表 27、286 ページの表 28、および 299 ページの表 29 はそれぞれ、入力順データ・セット、索引付きデータ・セット、および相対レコード・データ・セットで利用できるステートメントを示しています。

	SEQUENTIAL	KEYED SEQUENTIAL	DIRECT
INPUT	ESDS KSDS RRDS Path(N) Path(U)	ESDS KSDS RRDS Path(N) Path(U)	KSDS RRDS Path(U)
OUTPUT	ESDS RRDS	ESDS KSDS RRDS	KSDS RRDS Path(U)
UPDATE	ESDS KSDS RRDS Path(N) Path(U)	ESDS KSDS RRDS Path(N) Path(U)	KSDS RRDS Path(U)

Key: ESDS Entry-sequenced data set
 KSDS Key-sequenced data set
 RRDS Relative record data set
 Path(N) Alternate index path with nonunique keys
 Path(U) Alternate index path with unique keys

You can combine the attributes on the left with those at the top of the figure for the data sets and paths shown. For example, only an ESDS and an RRDS can be SEQUENTIAL OUTPUT.

PL/I does not support dummy VSAM data sets.

図 34. VSAM データ・セットと使用できるファイル属性

表 26. 代替索引パスで実行できる処理

基本クラスター・タイプ	代替索引キー・タイプ	処理	制約事項
KSDS	固有キー	通常の KSDS として	キーのアクセスを変更できない。
	非固有キー	制限つきキー・アクセス	キーのアクセスを変更できない。
ESDS	固有キー	KSDS として	削除なし。
	非固有キー	制限つきキー・アクセス	キーのアクセスを変更できない。 削除なし。
			キーのアクセスを変更できない。

VSAM データ・セットのファイルの定義

順次 VSAM データ・セットを定義するには、次の属性を指定したファイル宣言を使用します。

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED
      [KEYED]
      ENVIRONMENT(options);
```

直接 VSAM データ・セットを定義するには、次の属性を指定したファイル宣言を使用します。

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      DIRECT
      [KEYED]
      ENVIRONMENT(options);
```

209 ページの表 13 はデフォルト属性を示しています。ファイル属性については、「PL/I 言語解説書」で説明されています。ENVIRONMENT 属性のオプションについては、以下に説明します。

ファイル属性 INPUT、OUTPUT、UPDATE、と DIRECT、SEQUENTIAL、KEYED SEQUENTIAL とのいくつかの組み合わせは、特定のタイプの VSAM データ・セットにしか使用できません。277 ページの図 34 に互換性のある組み合わせを示しています。

ENVIRONMENT オプションの指定

データ・セット構造に影響を与える ENVIRONMENT 属性のオプションの多くは、VSAM データ・セットでは必要ありません。このオプションを指定しても、無視されるか、または検査の目的で使用するだけです。そこで検査されたものと、そのデータ・セット用に定義された値が矛盾していると、ファイルをオープンしようとした場合、UNDEFINEDFILE 条件が生じます。

VSAM データ・セットに使用できる ENVIRONMENT オプションは、次のとおりです。

```
BKWD
BUFND (n)
BUFNI (n)
BUFSP (n)
GENKEY
PASSWORD (password-specification)
REUSE
SCALARVARYING
SKIP
VSAM
```

GENKEY および SCALARVARYING オプションは、非 VSAM データ・セットで使用されたときと同じ効果を持ちます。VSAM RLS 環境では、オプション BUFND、BUFNI、および BUFSP は無視されるので注意してください。

VSAM データ・セットで検査されるオプションは RECSIZE で、キー順データ・セットで検査されるのは KEYLENGTH と KEYLOC の各オプションです。209 ページの表 13 は VSAM ではどのオプションが無視されるかを示しています。209 ページの表 13 はまた必須オプションおよびデフォルト・オプションも示しています。

VSAM データ・セットの場合、データ・セットを定義するときに、アクセス方式サービス・ユーティリティーに対して、レコードの最大長と平均長を指定します。検

査の目的でファイル宣言に RECSIZE オプションを組み込む場合は、最大レコード・サイズを指定します。指定した RECSIZE がデータ・セットに定義されている値と矛盾する場合は、UNDEFINEDFILE 条件が発生します。

BKWD オプション

BKWD オプションは、VSAM データ・セットと関連した SEQUENTIAL INPUT ファイルまたは SEQUENTIAL UPDATE ファイルでの逆方向処理を指定するのに使用します。

▶▶—BKWD—▶▶

順次読み取り（すなわち、KEY オプションなしの読み取り）では、直前の順序にあるレコードが検索されます。索引付きデータ・セットの場合は、直前のレコードは、一般的に、その次の低位キーを持つレコードのことです。しかし、非固有な代替索引を介してデータ・セットにアクセスしている場合は、同じキーを持つレコードは通常のシーケンスでリカバリーされます。例えば、次のような順序になったレコードの場合を考えます。

A B C1 C2 C3 D E

ここで、C1、C2、および C3 は同じキーを持つと、次の順序で回復されます。

E D C1 C2 C3 B A

BKWD オプションを指定したファイルをオープンすると、データ・セットは最終レコードに位置決めされます。そのデータ・セットの先頭まで来ると、通常どおりに ENDFILE が生じます。

REUSE オプションや GENKEY オプションと一緒に、BKWD オプションを指定しないでください。また、BKWD オプションを指定して宣言されたファイルには、WRITE ステートメントは使用できません。

BUFND オプション

BUFND オプションは、VSAM データ・セットに必要なデータ・バッファの数を指定するのに使用します。

▶▶—BUFND—(n)—▶▶

n 整数、または属性 FIXED BINARY(31) STATIC を持つ変数を指定します。

ファイルが SEQUENTIAL 属性を持ち、連続レコードの長いグループを順次処理する場合は、複数のデータ・バッファを使用するとパフォーマンス面で有効です。

BUFNI オプション

BUFNI オプションは、VSAM キー順データ・セットに必要な索引バッファの数を指定するのに使用します。

▶▶—BUFNI—(n)—▶▶

n 整数、または属性 FIXED BINARY(31) STATIC を持つ変数を指定します。

ファイルが KEYED 属性を持つ場合は、複数の索引バッファを使用するとパフォーマンス面で有効です。少なくとも、索引のレベル数と同数の索引バッファ数を指定してください。

BUFSP オプション

BUFSP オプションは、VSAM データ・セットに必要な合計バッファ・スペースをバイト単位で指定するのに使用します (データ・コンポーネントと索引コンポーネントの両方についての合計)。

►►—BUFSP—(*n*)——►►

n 整数、または属性 FIXED BINARY(31) STATIC を持つ変数を指定します。

通常は BUFSP でなく、BUFNI と BUFND オプションを指定することをお勧めします。

GENKEY オプション

このオプションは、214 ページの『GENKEY オプション - キーの分類』で説明しています。

PASSWORD オプション

システムに VSAM データ・セットを定義すると (アクセス方式サービスの DEFINE コマンドを使用)、READ パスワードと UPDATE パスワードをシステムに関連付けることができます。その時点から、データ・セットへのアクセスに使用する PL/I ファイルの宣言には、該当するパスワードを組み込む必要があります。

►►—PASSWORD—(—*password-specification*—)——►►

password-specification

プログラムが必要とするアクセス・タイプに使用するパスワードを指定する文字定数または文字変数です。定数を指定する場合は、反復因数を含めることはできません。変数を指定する場合は、レベル 1、エレメント、静的、および添え字なしでなければなりません。

文字ストリングは埋め込みまたは切り捨てが行われて 8 文字にされた後、VSAM に渡されて検査されます。誤ったパスワードだった場合、システム・オペレーターには、正しいパスワードを指定するための機会が何回か与えられます。その許容回数は、データ・セットを定義するときに指定します。失敗した試行がこの回数に達すると、UNDEFINEDFILE 条件が発生します。

REUSE オプション

REUSE オプションは、VSAM データ・セットに関連した OUTPUT ファイルを作業ファイルとして使用することを指定するのに使用します。

►►—REUSE——►►

データ・セットは、ファイルをオープンするたびに、空のデータ・セットとして扱われます。そのデータ・セット用のすべての 2 次割り振りが解放され、データ・セットは初めてオープンされたときと同様に取り扱われます。

REUSE オプションが指定されたファイルを、代替索引を持つデータ・セット、あるいは BKWD オプションが指定されているデータ・セットに関連付けたり、INPUT や UPDATE を行うためにオープンしたりしないでください。

REUSE オプションが有効なのは、アクセス方式サービス DEFINE CLUSTER コマンド内で REUSE を指定した場合だけです。

SKIP オプション

ENVIRONMENT 属性の SKIP オプションは、VSAM OPTCD の「SKP」を、可能であればいつでも使用することを指定するのに使用します。このオプションは、KEYED SEQUENTIAL INPUT または UPDATE ファイルを使用してアクセスするキー順データ・セットに適用できます。

▶▶—SKIP—▶▶

このオプションは、データ・セット全体に分散している個々のレコードにプログラムがアクセスし、そのアクセスが主に昇順のキー順で行われる場合に、ファイルに指定します。

プログラムが KEY オプションを使用せずに多数のレコードを順次読み取る場合、またはプログラムがデータ・セット内の特定のポイントに多数のレコードを挿入する場合は (大量順次挿入)、このオプションを省略してください。

SKIP オプションを指定 (または省略) することは、決して間違った使い方ではありません。このオプションがパフォーマンスに大きく影響するのは、前述の場合のみです。

VSAM オプション

VSAM データ・セットに VSAM オプションを指定する必要があります。

▶▶—VSAM—▶▶

パフォーマンス・オプション

DD ステートメントの AMP パラメーター内にバッファ・オプションを指定することもできます。これらは、アクセス方式サービスの資料に解説があります。

代替索引パス用のファイルの定義

VSAM では、キー・シーケンスおよび入力順データ・セットで代替索引を定義できます。この機能によって、基本索引以外の多くの方法でキー・シーケンス・データ・セットにアクセスできるようになります。また、入力順データ・セットに索引を付け、キーで、またはキー・シーケンスにアクセスできるようにもなります。そ

の結果、1 つの形式で作成されたデータに、さまざまな方法でアクセスできるようになります。例えば、ある従業員ファイルが、個人番号、名前、および部門番号によって索引付けられていたとします。

代替索引が作成されると、ユーザーは、実際は代替索引パス (代替索引とデータ・セット間の接続として機能する) とされた 3 番目のオブジェクトを通じてデータ・セットにアクセスします。

ここでは、2 つのタイプの代替索引が使用できます。固有キーと非固有キーです。固有キー代替索引の場合は、それぞれのレコードが別の代替キーを持っていないければなりません。非固有キー代替索引の場合は、任意の数のレコードが同じ代替キーを持つことができます。上記の例では、名前を使用している代替索引は固有キー代替索引になります (それぞれの人が別の名前を持つと仮定します)。部門番号を使用している代替索引は、各部門に複数の人が所属することができるので非固有キー代替索引になります。

ほとんどの点で、固有キー代替索引パスを通じてアクセスされたデータ・セットを、基本索引を通じてアクセスされた KSDS の場合と同様に扱うことができます。レコードはキーによって、または順次にアクセスでき、ユーザーはレコードを更新し、さらに新規のレコードを追加できます。データ・セットが KSDS であれば、レコードを削除し、更新したレコードの長さを変更できます。制約事項と許可されている処理については、277 ページの表 26 で示しています。レコードを追加または削除したら、データ・セットに関連したすべての索引が、新しい状況を反映するために変更されます (デフォルト)。

非固有キー代替索引パスを使用してデータ・セットにアクセスする場合、アクセスされるレコードはキーとシーケンスによって判別されます。キーを使用して位置決めを行えば、順次アクセスを続けることができます。キーを使用して、最初のレコードにアクセスできます。データ・セットを逆方向に読み取る場合は、キーの順番のみが逆になります。同じキーを持つレコードの順番は、データ・セットをいずれの方向で読み取るにしても、同じままです。

VSAM データ・セットの定義

VSAM データ・セットを定義し、カタログするには、アクセス方式サービスの `DEFINE CLUSTER` コマンドを使用します。 `DEFINE` コマンドを使用するには、次の事項をあらかじめ理解している必要があります。

- マスター・カタログがパスワード保護されていれば、マスター・カタログの名前とパスワード。
- マスター・カタログを使用しないのであれば、使用する VSAM 専用カタログの名前とパスワード。
- VSAM スペースをデータ・セットに使用できるかどうか。
- 作成する VSAM データ・セットのタイプ。
- データ・セットを入れる先のボリューム。
- データ・セット内の平均および最大レコード・サイズ。
- 索引付きデータ・セット用のキーの位置と長さ。
- データ・セットに割り振られるスペース。

- DEFINE コマンドのコーディング法。
- アクセス方式サービス・プログラムの使用方法。

上記の情報が得られたら、DEFINE コマンドをコーディングしてから、アクセス方式サービスを使用してデータ・セットを定義し、カタログすることができます。

入力順データ・セット

ESDS に関連したファイルで利用できるステートメントとオプションを、表 27 に示します。

表 27. VSAM 入力順データ・セットのロードと入力順データ・セットへのアクセスで利用できるステートメントとオプション

ファイル宣言 ¹	有効ステートメントおよび必須オプション	指定できるその他のオプション
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	LOCATE 基底付き変数 FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) または KEY(expression) ³
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) または KEY(expression) ³
	READ FILE(file-reference);	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) または KEY(expression) ³
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) または KEY(expression) ³
	READ FILE(file-reference) ²	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) および/または KEY(expression) ³

注:

1. 完全なファイル宣言には属性 FILE、RECORD、および ENVIRONMENT が含まれます。オプションの KEY あるいは KEYTO のいずれかを使用する場合は、属性 KEYED も含めなくてはなりません。
2. ステートメント「READ FILE(file-reference);」はステートメント「READ FILE(file-reference) IGNORE (1);」と同等です。
3. KEY オプション内で使用する式は、あらかじめ KEYTO オプションで入手した相対バイト・アドレスでなければなりません。

ESDS のロード

ESDS がロードされるときには、SEQUENTIAL OUTPUT 用に関連ファイルをオープンする必要があります。レコードは、提示された順序で保持されます。

KEYTO オプションを使用すれば、各レコードが書き込まれるときの相対バイト・アドレスを入手することができます。後でこれらのキーを使用すれば、このデータ・セットにキーによるアクセスを行うことができます。

SEQUENTIAL ファイルの使用による ESDS へのアクセス

ESDS にアクセスするのに使用する SEQUENTIAL ファイルは、INPUT 属性または UPDATE 属性を使用してオープンすることができます。KEY オプションまたは KEYTO オプションを使用する場合には、ファイルには、KEYED 属性も必要です。

順次アクセスの順序は、レコードをデータ・セットに初めにロードしたときと同じです。読み取られるレコードの RBA を回復するには、READ ステートメントで KEYTO オプションを使用します。KEY オプションを使用すると、回復されるレコードは、ユーザーが指定する RBA を持つレコードになります。次の順次アクセスは、データ・セットの新しい場所から開始されます。

UPDATE ファイルの場合、WRITE ステートメントは、データ・セットの終わりに新たにレコードを付け加えます。REWRITE ステートメントでは、再書き込みの行われるレコードは、KEY オプションを使用する場合は、指定された RBA を持つものであり、そうでない場合には、直前の READ でアクセスされたレコードです。REWRITE ステートメントで置き換えようとするレコードの長さを変更してはなりません。

入力順データ・セットでは、DELETE ステートメントは使用できません。

ESDS の定義とロード

285 ページの図 35 では、データ・セットは DEFINE CLUSTER コマンドで定義され、PLIVSAM.AJC1.BASE という名前が与えられています。NONINDEXED キーワードを使用すると、ESDS が定義されることになります。

PL/I プログラムは、SEQUENTIAL OUTPUT ファイルと WRITE FROM ステートメントを使ってデータ・セットを書き込みます。このファイル用の DD ステートメントには、DEFINE CLUSTER コマンドの NAME パラメーターで与えられたデータ・セットの DSNAME が入っています。

レコードの RBA は、KEYED ファイル内のキーとして後で使用するために、書き込み時に入手しておくこともできます。それを行うには、キーを保持する適切な変数と、使用する WRITE...KEYTO ステートメントを宣言しておく必要があります。次に例を示します。

```
DCL CHARS CHAR(4);  
WRITE FILE(FAMFILE) FROM (STRING)  
  KEYTO(CHARS);
```

通常、キーは印刷できませんが、後で使用するののためにとっておくことができることに注意してください。

カタログ式プロシージャー IBMZCBG を使用しています。同じプログラム (図 35 参照) を使ってデータ・セットにレコードを追加できるため、このプログラムはライブラリーに保持されています。このプロシージャーの例を次に示します。

```
//OPT9#7 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
  DEFINE CLUSTER -
    (NAME(PLIVSAM.AJC1.BASE) -
    VOLUMES(nnnnnn) -
    NONINDEXED -
    RECORDSIZE(80 80) -
    TRACKS(2 2))
/*
//STEP2 EXEC IBMZCLG
//PLI.SYSIN DD *
  CREATE: PROC OPTIONS(MAIN);

  DCL
    FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
    IN FILE RECORD INPUT,
    STRING CHAR(80),
    EOF BIT(1) INIT('0'B);

  ON ENDFILE(IN) EOF='1'B;

  READ FILE(IN) INTO (STRING);
  DO I=1 BY 1 WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
    WRITE FILE(FAMFILE) FROM (STRING);
    READ FILE(IN) INTO (STRING);
    END;

  PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
  END;
/*
//LKED.SYSLMOD DD DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
//              UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.FAMFILE DD DSN=PLIVSAM.AJC1.BASE,DISP=OLD
//GO.IN DD *
FRED          69          M
ANDY          70          M
SUZAN         72          F
/*
```

図 35. 入力順データ・セット (ESDS) の定義とロード

ESDS の更新

図 36 は、ESDS の終わりに新しいレコードを追加する例を示しています。これは、図 35 に示したプログラムを再度実行して行います。SEQUENTIAL OUTPUT ファイルが使用され、図 35 に示した DEFINE コマンドに指定された名前 PLIVSAM.AJC1.BASE を指定する DSNNAME パラメーターを使用して、データ・セットをこのファイルと関連付けています。

```
//OPT9#8 JOB
//STEP1 EXEC PGM=PGMA
//STEPLIB DD DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP)
// DD DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=A
//FAMFILE DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//IN DD *
JANE 75 F
//
```

図 36. ESDS の更新

レコードの長さを変えないかぎり、ESDS 内の既存レコードを再書き込みすることができます。これには、SEQUENTIAL または KEYED SEQUENTIAL 更新ファイルを使用します。キーを使用する場合は、RBA または代替索引 パスのキーを使用することができます。

ESDS では削除を行うことはできません。

キー順および索引付き入力順データ・セット

索引付き VSAM データ・セットで使用できるステートメントとオプションは、表 28 に示してあります。索引付きデータ・セットは基本索引を持つ KSDS、または代替索引を持つ KSDS または ESDS にすることができます。特に断り書きがなければ、次の説明はすべての索引付き VSAM データ・セットに適用されます。

表 28. VSAM 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション

ファイル 宣言 ¹	有効ステートメント および必須オプション	指定できるその他の オプション
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE 基底付き変数 FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference); ²	IGNORE(expression)

表 28. VSAM 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション (続き)

ファイル 宣言 ¹	有効ステートメント および必須オプション	指定できるその他の オプション
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference); ²	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	REWRITE FILE(file-reference);	FROM(reference) および/または KEY(expression)
	DELETE FILE(file-reference)	KEY(expression)
DIRECT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	DELETE FILE(file-reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

表 28. VSAM 索引付きデータ・セットのロードとそれへのアクセスに使用できるステートメントとオプション (続き)

ファイル 宣言 ¹	有効ステートメント および必須オプション	指定できるその他の オプション
-------------------------	-------------------------	--------------------

注:

1. 完全なファイル宣言には、属性 FILE と RECORD が組み込まれています。
KEY、KEYFROM または KEYTO オプションのどれか 1 つを使用するときは、宣言内に KEYED 属性も入れる必要があります。
2. ステートメント READ FILE(file-reference); は、ステートメント READ FILE(file-reference) IGNORE(1); と同等です。
3. SEQUENTIAL OUTPUT ファイルを、代替索引 を介してアクセスするデータ・セットに関連させないでください。
4. DIRECT ファイルを、非固有代替索引 を介してアクセスするデータ・セットに関連させないでください。
5. DELETE ステートメントは、代替索引 を介してアクセスする ESDS と関連させたファイルで使用することはできません。

KSDS または索引付き ESDS のロード: KSDS をロードするときには、KEYED SEQUENTIAL OUTPUT 用の関連ファイルをオープンする必要があります。レコードは昇順のキー順で提示しなければならず、KEYFROM オプションを使用しなければなりません。データ・セットをロードするためには基本索引を使用しなければならないことに注意してください。代替索引 を介して VSAM データ・セットをロードすることはできません。

KSDS に既にレコードがあって、SEQUENTIAL 属性と OUTPUT 属性を持つ関連ファイルをオープンする場合は、データ・セットの終わりにしかレコードを追加することはできません。前項のルールが適用されます。特に、指定する最初のレコードはデータ・セット上に存在する最高位のキーより大きいキーを持っていないてはなりません。

289 ページの図 37 は、KSDS を定義するのに使用する DEFINE コマンドを示しています。データ・セットには PLIVSAM.AJC2.BASE という名前が与えられ、INDEXED オペランドが使用されているため、KSDS と定義されています。レコード内のキーの位置は、KEYS オペランド内で定義されます。

PL/I プログラムでは、WRITE...FROM...KEYFROM ステートメントで KEYED SEQUENTIAL OUTPUT ファイルが使用されます。データは、昇順のキー順で提示されます。KSDS はこの方法でロードしなければなりません。

ファイルは、DEFINE コマンドで付けられた名前を DSNAME パラメーターとして使用する DD ステートメントによってデータ・セットに関連付けられます。

```

//OPT9#12 JOB
// EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
  DEFINE CLUSTER -
    (NAME(PLIVSAM.AJC2.BASE) -
    VOLUMES(nnnnnn) -
    INDEXED -
    TRACKS(3 1) -
    KEYS(20 0) -
    RECORDSIZE(23 80))
/*
// EXEC IBMZCBG
//PLI.SYSIN DD *
  TELNOS: PROC OPTIONS(MAIN);

    DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
      CARD CHAR(80),
      NAME CHAR(20) DEF CARD POS(1),
      NUMBER CHAR(3) DEF CARD POS(21),
      OUTREC CHAR(23) DEF CARD POS(1),
      EOF BIT(1) INIT('0'B);

    ON ENDFILE(SYSIN) EOF='1'B;

    OPEN FILE(DIREC) OUTPUT;

    GET FILE(SYSIN) EDIT(CARD)(A(80));
    DO WHILE (~EOF);
      WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
      GET FILE(SYSIN) EDIT(CARD)(A(80));
    END;

    CLOSE FILE(DIREC);

    END TELNOS;
/*
//GO.DIREC DD DSN=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.          162
BAKER,R.           152
BRAMLEY,O.H.       248
CHEESEMAN,D.       141
CORY,G.            336
ELLIOTT,D.         875
FIGGINS,S.         413
HARVEY,C.D.W.      205
HASTINGS,G.M.      391
KENDALL,J.G.       294
LANCASTER,W.R.     624
MILES,R.           233
NEWMAN,M.W.        450
PITT,W.H.          515
ROLF,D.E.          114
SHEERS,C.D.        241
SUTCLIFFE,M.       472
TAYLOR,G.C.        407
WILTON,L.W.        404
WINSTONE,E.M.     307
//

```

図 37. キー順データ・セット (KSDS) の定義とロード

SEQUENTIAL ファイルを使用した KSDS または索引付き ESDS へのアクセス:
 KSDS にアクセスするのに使用する SEQUENTIAL ファイルは、INPUT 属性または UPDATE 属性を使用してオープンすることができます。

KEY オプションを指定しない READ ステートメントの場合、レコードは昇順のキー順 (ただし、BKWD オプションを使用すると、降順のキー順) で回復されます。このように、KEY オプションによって、回復されたレコードのキーを得ることができます。

KEY オプションを使用すると、READ ステートメントで回復されるレコードは、指定したキーを持つレコードになります。このような READ ステートメントはデータ・セットを指定されたレコードに位置付け、その後の順次読み取りは後続のレコードを順番に回復します。

KEYFROM オプションのある WRITE ステートメントを、KEYED SEQUENTIAL UPDATE ファイルで使用することができます。前回行ったアクセスの位置に関係なく、データ・セット内の任意の位置に挿入を行うことができます。固有索引によってデータ・セットにアクセスするときは、そのデータ・セットに既にあるレコードと同じキーを持つレコードを挿入しようとする、KEY 条件が生じます。非固有索引の場合には、同一キーを持つレコードを後になって検索すると、レコードがデータ・セットに追加された順に、検索が行われます。

UPDATE ファイルでは、REWRITE ステートメントは、KEY オプションのあるなしに関係なく使用できます。KEY オプションを使用した場合、再書き込みされるレコードは、指定されたキーを持つ最初のレコードであり、それ以外の場合は直前の READ ステートメントによってアクセスされたレコードです。代替索引を使用してレコードを再書き込みする場合は、レコードの基本キーを変更しないでください。

DIRECT ファイルを使用した KSDS または索引付き ESDS へのアクセス: 索引付き VSAM データ・セットにアクセスするのに使用する DIRECT ファイルをオープンするには、INPUT 属性、OUTPUT 属性または UPDATE 属性を指定します。DIRECT ファイルを、非固有索引によってデータ・セットにアクセスするのに使用しないでください。

DIRECT OUTPUT ファイルを使ってデータ・セットにレコードを追加するときに、そのデータ・セットに既にあるレコードと同じキーを持つレコードを挿入しようすると、KEY 条件が生じます。

DIRECT INPUT ファイルまたは DIRECT UPDATE ファイルを使用すれば、KEYED SEQUENTIAL ファイルの場合と同様に、レコードの読み取り、書き込み、再書き込み、または削除が行えます。

291 ページの図 38 は、基本索引を使って KSDS を更新する 1 つの方法を示しています。

```

//OPT9#13 JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
DIRUPDT: PROC OPTIONS(MAIN);

    DCL DIREC FILE RECORD KEYED ENV(VSAM),
        ONCODE BUILTIN,
        OUTREC CHAR(23),
        NUMBER CHAR(3) DEF OUTREC POS(21),
        NAME CHAR(20) DEF OUTREC,
        CODE CHAR(1),
        EOF BIT(1) INIT('0'B);

    ON ENDFILE(SYSIN) EOF='1'B;

    ON KEY(DIREC) BEGIN;
        IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
            ('NOT FOUND: ',NAME)(A(15),A);
        IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
            ('DUPLICATE: ',NAME)(A(15),A);
    END;

    OPEN FILE(DIREC) DIRECT UPDATE;

    GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
        (COLUMN(1),A(20),A(3),A(1));
    DO WHILE (~EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
            (A(1),A(20),A(1),A(3),A(1),A(1));
        SELECT (CODE);
            WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
            WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
            WHEN('D') DELETE FILE(DIREC) KEY(NAME);
            OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
                ('INVALID CODE: ',NAME) (A(15),A);
        END;
        GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
            (COLUMN(1),A(20),A(3),A(1));
    END;

    CLOSE FILE(DIREC);
    PUT FILE(SYSPRINT) PAGE;
    OPEN FILE(DIREC) SEQUENTIAL INPUT;

    EOF='0'B;
    ON ENDFILE(DIREC) EOF='1'B;

    READ FILE(DIREC) INTO(OUTREC);
    DO WHILE(~EOF);
        PUT FILE(SYSPRINT) SKIP EDIT(OUTREC)(A);
        READ FILE(DIREC) INTO(OUTREC);
    END;
    CLOSE FILE(DIREC);
END DIRUPDT;

```

図 38. KSDS の更新 (1/2)

```

/*
//GO.DIREC DD DSN=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.          516C
GOODFELLOW,D.T.      889A
MILES,R.              D
HARVEY,C.D.W.         209A
BARTLETT,S.G.         183A
CORY,G.               D
READ,K.M.             001A
PITT,W.H.
ROLF,D.F.             D
ELLIOTT,D.            291C
HASTINGS,G.M.         D
BRAMLEY,O.H.          439C
/*

```

図 38. KSDS の更新 (2/2)

DIRECT 更新ファイルが使用され、ファイル SYSIN 内のレコードに渡されたコードにしたがって、データが変更されます。

- A 新しいレコードの追加
- C 既存名の番号の変更
- D レコードの削除

ラベル NEXT で、名前、番号、およびコードが読み取られ、そのコードの値に従って処理されます。KEY ON ユニットを使用して、誤ったキーに対する処理がとられます。更新が (ラベル PRINT で) 終了すると、ファイル DIREC はクローズされてから、属性 SEQUENTIAL INPUT でもう一度オープンされます。次に、このファイルは順次読み取られ、印刷されます。

このファイルは、289 ページの図 37 のアクセス方式サービスの DEFINE CLUSTER コマンドで定義された DSN の PLIVSAM.AJC2.BASE を使用する DD ステートメントによって、データ・セットと関連付けられます。

KSDS の更新方式: KSDS を更新する方法は、いくつかあります。ここに示す DIRECT ファイルを使う方法が、例に示したデータに適しています。大量順次挿入の場合は、KEYED SEQUENTIAL UPDATE ファイルを使用します。この方法によると、より高速のパフォーマンスが得られます。その理由は、データは各 WRITE ステートメントの後ではなく、本当に必要なときにしかデータ・セットに書き込まれず、データ・セット内のフリー・スペースの平衡が保たれるからです。

効率的な大量順次挿入を行うためのステートメントは、次のとおりです。

```

DCL DIREC KEYED SEQUENTIAL UPDATE
  ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
  KEYFROM(NAME);

```

PL/I 入出力ルーチンは、キーが順々になっていることを検出すると、VSAM に正しい要求を出します。キーが順々になっていない場合には、そのことが検出され、パフォーマンスの利点は失われますがエラーにはなりません。

KSDS または索引付き ESDS の代替索引

代替索引を使用すると、固有キーまたは非固有キーを使用して、さまざまな方法で KSDS または索引付き ESDS にアクセスできます。

固有キー代替索引パス: 図 39 では、285 ページの図 35 に定義されロードされた ESDS の固有キー代替索引パスを作成する方法を説明しています。このパスを使用すると、レコードの最初の 15 バイトの子の名前によって、データ・セットへの索引付けが行われます。

使用されるアクセス方式サービス・コマンドは、次のとおりです。

DEFINE ALTERNATEINDEX: 代替索引をデータ・セットとして VSAM に定義します。

BLDINDEX: 関連レコードへのポインターを代替索引に入れます。

DEFINE PATH: PL/I ファイルと関連付けるエンティティを、DD ステートメントで定義します。

DD ステートメントは、BLDINDEX の INFILE および OUTFILE オペランド、ならびにソート・ファイルが必要です。さまざまな個所で正しい名前を指定するように注意してください。

```
//OPT9#9      JOB
//STEP1       EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT    DD  SYSOUT=A
//SYSIN       DD  *
               DEFINE ALTERNATEINDEX -
                 (NAME(PLIVSAM.AJC1.ALPHIND) -
                  VOLUMES(nnnnnn) -
                  TRACKS(4 1) -
                  KEYS(15 0) -
                  RECORDSIZE(20 40) -
                  UNIQUEKEY -
                  RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2       EXEC PGM=IDCAMS,REGION=512K
//DD1         DD  DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2         DD  DSN=PLIVSAM.AJC1.ALPHIND,DISP=SHR
//SYSPRINT    DD  SYSOUT=A
//SYSIN       DD  *
               BLDINDEX INFILE(DD1) OUTFILE(DD2)
               DEFINE PATH -
                 (NAME(PLIVSAM.AJC1.ALPHPATH) -
                  PATHENTRY(PLIVSAM.AJC1.ALPHIND))
//
```

図 39. ESDS 用の固有キー代替索引パスの作成

非固有キー代替索引パス: 294 ページの図 40 では、ESDS の非固有キー代替索引パスを作成する方法を説明しています。代替索引を使用すると、子の性によってデータを選択できます。これにより少女または少年のデータに別々にアクセスでき、キーを使用して、それぞれのグループの各メンバーにアクセスできます。

使用されるアクセス方式サービス・コマンドは以下の 3 つです。

DEFINE ALTERNATEINDEX: 代替索引をデータ・セットとして VSAM に定義します。

BLDINDEX: 関連レコードへのポインターを代替索引に入れます。

DEFINE PATH: PL/I ファイルと関連付けるエンティティを、DD ステートメントで定義します。

DD ステートメントは、BLDINDEX の INFILE および OUTFILE オペランド、ならびにソート・ファイルが必要です。さまざまな個所で正しい名前を指定するように注意してください。

NONUNIQUEKEY オペランドを使用すると、索引が非固有キーを持つように指定できます。非固有キーを持つ索引を作成する場合は、指定する RECORDSIZE が十分大きくなるように注意してください。非固有代替索引では、それぞれの代替索引レコードが、関連した索引キーを持つすべてのレコードへのポインターを含みます。ポインターは、ESDS の場合は RBA の形式で、KSDS の場合は基本キーの形式です。多くのレコードが同じキーを持つ場合は、大きなレコードが必要です。

```
//OPT9#10 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
/* care must be taken with recordsize */
DEFINE ALTERNATEINDEX -
  (NAME(PLIVSAM.AJC1.SEXIND) -
   VOLUMES(nnnnnn) -
   TRACKS(4 1) -
   KEYS(1 37) -
   RECORDSIZE(20 400) -
   NONUNIQUEKEY -
   RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2 EXEC PGM=IDCAMS,REGION=512K
//DD1 DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2 DD DSN=PLIVSAM.AJC1.SEXIND,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
BLDINDEX INFILE(DD1) OUTFILE(DD2)
DEFINE PATH -
  (NAME(PLIVSAM.AJC1.SEXPATH) -
   PATHENTRY(PLIVSAM.AJC1.SEXIND))
//
```

図 40. ESDS 用の非固有キー代替索引パスの作成

295 ページの図 41 では、KSDS の固有キー代替索引パスを作成する方法を説明しています。データ・セットは電話番号による索引付けが行われ、その番号をキーとして使用することで、電話番号の持ち主の名前を検索することができるようになります。UNIQUEKEY を指定すると、キーを固有とすることができます。また、どの番号が使用されていないかを示すために、データ・セットを数値順にリストすることができます。使用されるアクセス方式サービス・コマンドは以下の 3 つです。

DEFINE ALTERNATEINDEX: 代替索引データを保持するデータ・セットを定義します。

BLDINDEX: 関連レコードへのポインターを代替索引に入れます。

DEFINE PATH: PL/I ファイルを関連付けるエンティティを、DD ステートメントで定義します。

DD ステートメントは、BLDINDEX の INFILE および OUTFILE、ならびにソート・ファイルが必要です。組み込まれる名前を混同することがないようにしてください。

```

//OPT9#14  JOB
//STEP1    EXEC  PGM=IDCAMS,REGION=512K
//SYSPRINT DD  SYSOUT=A
//SYSIN    DD  *
      DEFINE ALTERNATEINDEX -
        (NAME(PLIVSAM.AJC2.NUMIND) -
         VOLUMES(nnnnnn) -
         TRACKS(4 4) -
         KEYS(3 20) -
         RECORDSIZE(24 48) -
         UNIQUEKEY -
         RELATE(PLIVSAM.AJC2.BASE))
/*
//STEP2    EXEC  PGM=IDCAMS,REGION=512K
//DD1      DD  DSN=PLIVSAM.AJC2.BASE,DISP=SHR
//DD2      DD  DSN=PLIVSAM.AJC2.NUMIND,DISP=SHR
//SYSPRINT DD  SYSOUT=A
//SYSIN    DD  *
      BLDINDEX INFILE(DD1) OUTFILE(DD2)
      DEFINE PATH -
        (NAME(PLIVSAM.AJC2.NUMPATH) -
         PATHENTRY(PLIVSAM.AJC2.NUMIND))
//

```

図 41. KSDS 用の固有キー代替索引パスの作成

固有キーを使用して代替索引を作成する場合は、同じ代替キーで他のレコードが組み込まれることがないようにしてください。実際、固有キー代替索引は、電話帳として使用するには、完全に満足のものとはいえません。なぜなら、固有キー代替索引では、2 人の人間が同じ番号を持つことができないためです。同様に、基本キーでは 1 人の人間が 2 つの電話番号を持てません。解決策として、ESDS に 2 つの非固有キー代替索引を持たせるか、または 1 人の人間が複数の番号を持てるようにデータ・フォーマットを再構築して、複数の番号に対応した非固有キー代替索引を 1 つを持たせるようにします。

非固有代替索引キーの検出: 代替索引パスを使用して VSAM データ・セットにアクセスしている場合は、SAMEKEY 組み込み関数を使用して、非固有キーの有無を検出できます。非固有キーを検出した場合、SAMEKEY は、検索されたレコードと同じ代替索引キーを持つレコードが他に存在するかどうかを示します。このため、レコードの読み取りは、非固有キーを持つ一連のレコードの最後で停止して、それ以降のレコードを読み取らないことがあります。SAMEKEY (ファイル参照) は、入出力ステートメントが正しく完了し、アクセスしたレコードの次に、同じキーを持つ別のレコードが続く場合は '1'B を戻します。そうでない場合は '0'B を戻します。

ESDS での代替索引の使用: 297 ページの図 42 は、ESDS での代替索引の使用と、逆方向読み取りについて説明します。このプログラムは以下の 4 つのファイルを使用します。

BASEFLE

基本データ・セットを順方向に読み取る。

BACKFLE

基本データ・セットを逆方向に読み取る。

ALPHFLE

名前によって子の索引付けを行う英字代替索引パス。

SEXFILE

子の性別に対応する代替索引パス。

すべてのファイルに DD ステートメントがあります。DSNAME パラメーターに基本データ・セット名を指定して、BASEFLE と BACKFLE を基本データ・セットに接続し、293 ページの図 39 と 294 ページの図 40 で示したパス名を指定して ALPHFLE と SEXFILE を接続します。

プログラムではデータにアクセスするために SEQUENTIAL ファイルを使用し、最初はそれを通常の順で書き込みし、それから逆順で書き込みします。AGEQUERY ラベルでは、固有代替索引の代替索引キーに関連付けられたデータを読み取るために、DIRECT ファイルが使用されます。

最後に、SPRINT ラベルでは、非固有キー代替索引パスを使用して、ファミリー内の女性のリストを書き込むために KEYED SEQUENTIAL ファイルが使用されます。同じキーを持つすべてのレコードを読み取るために、SAMEKEY 組み込み関数が使用されます。女性名はもともと入力された順でアクセスされます。これが行われるのはファイルが順方向または逆方向に読み取られる場合です。非固有キー・パスの場合は、BKWD オプションはキーの読み取り順にのみ影響します。同じキーを持つ項目の順番はファイルが順方向に読み取られる場合と同じです。

削除: 例の終わりでは、アクセス方式サービスの DELETE コマンドが基本データ・セットを削除するために使用されています。これが行われると、関連する代替索引とパスも削除されます。

KSDS での代替索引の使用: 299 ページの図 43 では、KSDS を更新するための、固有な代替索引キーを持つパスの使用法と代替索引の順にそれにアクセスし出力する方法を説明します。

代替索引パスは、パスの名前 (PLIVSAM.AJC2.NUMPATH、295 ページの図 41 の DEFINE PATH コマンドで指定されたもの) を DSNAME として指定する DD ステートメントによって、PL/I ファイルに関連付けられています。

プログラムの最初のセクションでは、代替索引キーを使用して新規のレコードを挿入するために、DIRECT OUTPUT ファイルが使用されます。代替索引を用いて変更を加える場合は、既存のレコードへのアクセス用の基本キーや代替索引キーを変更しないでください。また、変更の際に、基本索引や固有キー代替索引に、重複するキーを追加しないでください。

プログラムの 2 番目のセクションでは (PRINTIT ラベルで)、データ・セットが SEQUENTIAL INPUT ファイルを使用して代替索引キーの順で読み取られます。その後、SYSPRINT に書き込まれます。

```

//OPT9#15 JOB
//STEP1 EXEC IBMZCLG
//PLI.SYSIN DD *
READIT: PROC OPTIONS(MAIN);
  DCL BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
        /*File to read base data set forward */
  BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
        /*File to read base data set backward */
  ALPHFLE FILE DIRECT INPUT ENV(VSAM),
        /*File to access via unique alternate index path */
  SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
        /*File to access via nonunique alternate index path */
  STRING CHAR(80), /*String to be read into */
  1 STRUC DEF (STRING),
    2 NAME CHAR(25),
    2 DATE_OF_BIRTH CHAR(2),
    2 FILL CHAR(10),
    2 SEX CHAR(1);
  DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;
  DCL EOF BIT(1) INIT('0'B);

  /*Print out the family eldest first*/

  ON ENDFILE(BASEFLE) EOF='1'B;
  PUT EDIT('FAMILY ELDEST FIRST')(A);
  READ FILE(BASEFLE) INTO (STRING);
  DO WHILE(~EOF);
    PUT SKIP EDIT(STRING)(A);
    READ FILE(BASEFLE) INTO (STRING);
  END;
  CLOSE FILE(BASEFLE);
  PUT SKIP(2);
  /*Close before using data set from other file not
  necessary but good practice to prevent potential
  problems*/

  EOF='0'B;
  ON ENDFILE(BACKFLE) EOF='1'B;
  PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);
  READ FILE(BACKFLE) INTO(STRING);
  DO WHILE(~EOF);
    PUT SKIP EDIT(STRING)(A);
    READ FILE(BACKFLE) INTO (STRING);
  END;

  CLOSE FILE(BACKFLE);
  PUT SKIP(2);

  /*Print date of birth of child specified in the file
  SYSIN*/
  ON KEY(ALPHFLE) BEGIN;
    PUT SKIP EDIT
      (NAMEHOLD,' NOT A MEMBER OF THE SMITH FAMILY') (A);
    GO TO SPRINT;
  END;

```

図 42. ESDS での代替索引パスと逆方向読み取り (1/2)

```

AGEQUERY:
  EOF='0'B;
  ON ENDFILE(SYSIN) EOF='1'B;
  GET SKIP EDIT(NAMEHOLD)(A(25));
  DO WHILE(¬EOF);
    READ FILE(ALPHFLE) INTO (STRING) KEY(NAMEHOLD);
    PUT SKIP (2) EDIT(NAMEHOLD,' WAS BORN IN ',
      DATE_OF_BIRTH)(A,X(1),A,X(1),A);
    GET SKIP EDIT(NAMEHOLD)(A(25));
  END;
SPRINT:
  CLOSE FILE(ALPHFLE);
  PUT SKIP(1);

/*Use the alternate index to print out all the females in the
family*/
  ON ENDFILE(SEXFILE) GOTO FINITO;
  PUT SKIP(2) EDIT('ALL THE FEMALES')(A);
  READ FILE(SEXFILE) INTO (STRING) KEY('F');
  PUT SKIP EDIT(STRING)(A);
  DO WHILE(SAMEKEY(SEXFILE));
    READ FILE(SEXFILE) INTO (STRING);
    PUT SKIP EDIT(STRING)(A);
  END;

FINITO:
  END;

/*
//GO.BASEFLE DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//GO.BACKFLE DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//GO.ALPHFLE DD DSN=PLIVSAM.AJC1.ALPHPATH,DISP=SHR
//GO.SEXFILE DD DSN=PLIVSAM.AJC1.SEXPATH,DISP=SHR
//GO.SYSIN DD *
ANDY
/*
//STEP2 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DELETE -
        PLIVSAM.AJC1.BASE
//

```

図 42. ESDS での代替索引パスと逆方向読み取り (2/2)

```

//OPT9#16 JOB
//STEP1 EXEC IBMZCLG,REGION.GO=256K
//PLI.SYSIN DD *
ALTER: PROC OPTIONS(MAIN);
  DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
  NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
  IN FILE RECORD,
  STRING CHAR(80),
  NAME CHAR(20) DEF STRING,
  NUMBER CHAR(3) DEF STRING POS(21),
  DATA CHAR(23) DEF STRING,
  EOF BIT(1) INIT('0'B);

  ON KEY (NUMFLE1) BEGIN;
    PUT SKIP EDIT('DUPLICATE NUMBER')(A);
  END;

  ON ENDFILE(IN) EOF='1'B;

  READ FILE(IN) INTO (STRING);
  DO WHILE(~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
    WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
    READ FILE(IN) INTO (STRING);
  END;

  CLOSE FILE(NUMFLE1);

  EOF='0'B;
  ON ENDFILE(NUMFLE2) EOF='1'B;

  READ FILE(NUMFLE2) INTO (STRING);
  DO WHILE(~EOF);
    PUT SKIP EDIT(DATA) (A);
    READ FILE(NUMFLE2) INTO (STRING);
  END;

  PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
END;
/*
//GO.IN DD *
RIERA L 123
/*
//NUMFLE1 DD DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//NUMFLE2 DD DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//STEP2 EXEC PGM=IDCAMS,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DELETE -
PLIVSAM.AJC2.BASE
//

```

図 43. KSDS アクセス用の固有代替索引パスの使用

相対レコード・データ・セット

VSAM 相対レコード・データ・セット (RRDS) で使用できるステートメントとオプションを、表 29 に示します。

表 29. VSAM 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション

ファイル 宣言 ¹	有効ステートメント および必須オプション	指定できるその他の オプション
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) または KEYTO(reference)
	LOCATE 基底付き変数 FILE(file-reference);	SET(pointer-reference)

表 29. VSAM 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション (続き)

ファイル 宣言 ¹	有効ステートメント および必須オプション	指定できるその他の オプション
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference); ²	IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference); ²	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) または KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) および/または KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	

表 29. VSAM 相対レコード・データ・セットのロードとそれへのアクセスに使用できるステートメントとオプション (続き)

ファイル宣言 ¹	有効ステートメントおよび必須オプション	指定できるその他のオプション
DIRECT UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression); READ FILE(file-reference) SET(pointer-reference) KEY(expression); REWRITE FILE(file-reference) FROM(reference) KEY(expression); DELETE FILE(file-reference) KEY(expression); WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

注:

1. 完全なファイル宣言には、属性 FILE と RECORD が組み込まれています。
KEY、KEYFROM、KEYTO オプションのどれか 1 つを使用する場合は、宣言に属性 KEYED も指定する必要があります。

DIRECT UPDATE ファイルでの UNLOCK ステートメントは VSAM RRDS に関連したファイルに使用されると無視されます。

2. ステートメント READ FILE(file-reference); は、ステートメント READ FILE(file-reference) IGNORE(1); と同等です。

RRDS のロード: RRDS をロードするときには、OUTPUT 用の関連ファイルをオープンする必要があります。DIRECT ファイルまたは SEQUENTIAL ファイルを使用します。

DIRECT OUTPUT ファイルの場合、各レコードは WRITE ステートメントの KEYFROM オプション内の相対レコード番号 (つまりキー) で指定された位置に入れます (274 ページの『VSAM データ・セットのキー』を参照)。

SEQUENTIAL OUTPUT ファイルの場合、KEYFROM オプションの指定の有無に関係なく WRITE ステートメントを使用します。KEYFROM オプションを指定すると、レコードは指定されたスロット内に置かれ、省略した場合は、レコードは現在位置に続くスロット内に置かれます。レコードを昇順の相対レコード番号順に並べる必要はありません。KEYFROM オプションを省略しても、KEYTO オプションにより、書き込まれたレコードの相対レコード番号を入手することができます。

KEYFROM オプションも KEYTO オプションも使わずに、RRDS を順次にロードする場合は、ファイルは KEYED 属性を持っている必要はありません。

既にレコードが存在する位置にレコードをロードしようとすることは誤りです。
KEYFROM オプションを使用する場合は、KEY 条件が生じ、それを省略する場合は ERROR 条件が発生します。

303 ページの図 44 では、データ・セットは DEFINE CLUSTER コマンドで定義され、PLIVSAM.AJC3.BASE という名前が与えられます。それが RRDS であるという事実は、NUMBERED キーワードによって判断されます。PL/I プログラムでは、データ・セットは DIRECT OUTPUT ファイルによってロードされ、WRITE...FROM...KEYFROM ステートメントが使用されます。

データが順に並んでいてキーが順序どおりになっていれば、SEQUENTIAL ファイルを使用して、先頭からデータ・セットに書き込むことができたはずですが。その後レコードは、次に使用できるスロットに入れられ、該当する番号が付けられます。各レコード用のキーの番号が、KEYTO オプションを使って戻されたはずですが。

PL/I ファイルは、DEFINE CLUSTER コマンドで指定された名前を DSNAME として使用する DD ステートメントによってデータ・セットへ関連付けられます。

```

//OPT9#17 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
        DEFINE CLUSTER -
            (NAME(PLIVSAM.AJC3.BASE) -
             VOLUMES(nnnnnn) -
             NUMBERED -
             TRACKS(2 2) -
             RECORDSIZE(20 20))

/*
//STEP2 EXEC IBMZCBG
//PLI.SYSIN DD *
CRR1:  PROC OPTIONS(MAIN);
        DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
            CARD CHAR(80),
            NAME CHAR(20) DEF CARD,
            NUMBER CHAR(2) DEF CARD POS(21),
            IOFIELD CHAR(20),
            EOF BIT(1) INIT('0'B);
        ON ENDFILE (SYSIN) EOF='1'B;
        OPEN FILE(NOS);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
        DO WHILE (~EOF);
            PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
            IOFIELD=NAME;
            WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
            GET FILE(SYSIN) EDIT(CARD)(A(80));
        END;
        CLOSE FILE(NOS);
    END CRR1;
/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.       28
CHEESNAME,L.       11
CORY,G.            36
ELLIOTT,D.         85
FIGGINS,E.S.       43
HARVEY,C.D.W.      25
HASTINGS,G.M.      31
KENDALL,J.G.       24
LANCASTER,W.R.     64
MILES,R.           23
NEWMAN,M.W.        40
PITT,W.H.          55
ROLF,D.E.          14
SHEERS,C.D.        21
SURCLIFFE,M.       42
TAYLOR,G.C.        47
WILTON,L.W.        44
WINSTONE,E.M.      37
//

```

図 44. 相対レコード・データ・セット (RRDS) の定義とロード

SEQUENTIAL ファイルを使用した RRDS へのアクセス: RRDS にアクセスするのに使用する SEQUENTIAL ファイルは、INPUT 属性または UPDATE 属性を使用してオープンすることができます。KEY、KEYTO、KEYFROM オプションのいずれかを使用する場合は、属性 KEYED も指定する必要があります。

KEY オプションが指定されていない READ ステートメントの場合、レコードは、昇順の相対レコード番号順に回復されます。データ・セット内に空きスロットがあれば、すべてスキップされます。

KEY オプションを指定すると、READ ステートメントで回復されるレコードは、指定した相対レコード番号を持つレコードになります。このような READ ステートメントはデータ・セットを指定されたレコードに位置付け、その後の順次読み取りは後続のレコードを順番に回復します。

KEYFROM オプションが指定されていなくても、WRITE ステートメントは、KEYED SEQUENTIAL UPDATE ファイルに使用することができます。前回行ったアクセスの位置に関係なく、データ・セット内の任意の位置に挿入を行うことができます。KEYFROM オプションが指定された WRITE ステートメントの場合、そのデータ・セットに既に存在するレコードと同じ相対レコード番号を持つレコードを挿入しようとする、KEY 条件が発生します。KEYFROM オプションを省略すると、現在位置から見て次のスロットにレコードが書き込まれます。このスロットが空いていなければ、ERROR 条件が発生します。

KEYTO オプションを使えば、KEYFROM オプションを指定していない WRITE ステートメントにより追加されるレコードのキーを回復することができます。

REWRITE ステートメントは、KEY オプションのあるなしに関係なく、UPDATE ファイルで使用できます。KEY オプションを使用した場合、再書き込みされるレコードは指定された相対レコード番号を持つレコードであり、そうでない場合は直前の READ ステートメントによってアクセスされたレコードです。

DELETE ステートメントは、KEY オプションのあるなしに関係なく、データ・セットにあるレコードを削除するのに使用することができます。

DIRECT ファイルを使った RRDS へのアクセス：RRDS にアクセスするのに使用する DIRECT ファイルは、OUTPUT 属性、INPUT 属性、UPDATE 属性を持つことができます。KEYED SEQUENTIAL ファイルを使用する場合と同様に、レコードの読み取り、書き込み、再書き込み、または削除が行えます。

305 ページの図 45 は、RRDS の更新の例を示しています。DIRECT UPDATE ファイルが使用され、キーによって新しいレコードが書き込まれます。VSAM では空のレコードは使用できないため、レコードが空かどうかを検査する必要はありません。

ラベル PRINT で始まるプログラムの後半では、更新済みファイルが印刷されます。ここでも、REGIONAL(1) にあるので、空のレコードの検査は必要ありません。

PL/I ファイルは、305 ページの図 45 の DEFINE CLUSTER コマンド内で定義された DSNAME の PLIVSAM.AJC3.BASE を指定する DD ステートメントで、データ・セットと関連づけられます。

例の終わりに、DELETE コマンドを使ってデータ・セットの削除が示されています。

```

/** NOTE: WITH A WRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
/**      A DUPLICATE MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
/**      WHOSE NEWNO CORRESPONDS TO AN EXISTING NUMBER IN THE LIST,
/**      FOR EXAMPLE, ELLIOT.
/**      WITH A REWRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
/**      A NOT FOUND MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
/**      WHOSE NEWNO DOES NOT CORRESPOND TO AN EXISTING NUMBER IN
/**      THE LIST, FOR EXAMPLE, NEWMAN AND BRAMLEY.
//OPT9#18 JOB
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
ACR1: PROC OPTIONS(MAIN);
        DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
        (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
        BYTE CHAR(1) DEF IOFIELD, EOF BIT(1) INIT('0'B),
        ONCODE BUILTIN;
ON ENDFILE(SYSIN) EOF='1'B;
OPEN FILE(NOS) DIRECT UPDATE;
ON KEY(NOS) BEGIN;
        IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
        ('NOT FOUND: ',NAME)(A(15),A);
        IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
        ('DUPLICATE: ',NAME)(A(15),A);
END;
GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
(COLUMN(1),A(20),A(2),A(2),A(1));
DO WHILE (~EOF);
PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
(A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
SELECT(CODE);
        WHEN('A') WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
        WHEN('C') DO;
                DELETE FILE(NOS) KEY(OLDNO);
                WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
        END;
        WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
        OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
        ('INVALID CODE: ',NAME)(A(15),A);
END;

```

図 45. RRDS の更新 (1/2)

```

        GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
          (COLUMN(1),A(20),A(2),A(2),A(1));
      END;
      CLOSE FILE(NOS);
      PRINT:
      PUT FILE(SYSPRINT) PAGE;
      OPEN FILE(NOS) SEQUENTIAL INPUT;
      EOF='0'B;
      ON ENDFILE(NOS) EOF='1'B;
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
      DO WHILE (~EOF);
      PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD) (A(5),A);
      READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
      END;
      CLOSE FILE(NOS);
    END ACRI;

/*
//GO.NOS      DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN     DD *
NEWMAN,M.W.      5640C
GOODFELLOW,D.T.  89  A
MILES,R.         23D
HARVEY,C.D.W.    29  A
BARTLETT,S.G.    13  A
CORY,G.          36D
READ,K.M.        01  A
PITT,W.H.        55
ROLF,D.F.        14D
ELLIOTT,D.       4285C
HASTINGS,G.M.    31D
BRAMLEY,O.H.     4928C
//STEP3       EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN       DD *
              DELETE -
              PLIVSAM.AJC3.BASE
//

```

図 45. RRDS の更新 (2/2)

非 VSAM データ・セット用に定義されたファイルの使用

共用データ・セットの使用

PL/I では、データ・セットの領域間またはシステム間での共用が許されます。このタイプのサポートは、VSAM によって提供されます。上記のサポートについての詳細は、以下の DFSMS の資料を参照してください。

- DFSMS: データ・セットの使用法
- DFSMS: カタログのためのアクセス方式サービス・プログラム

第 3 部 プログラムの改良

第 11 章 パフォーマンスの向上	309
最適なパフォーマンスのためのコンパイラー・オプションの選択	309
OPTIMIZE	309
GONUMBER.	310
ARCH	310
REDUCE	310
RULES	310
IBM/ANS	310
(NO)LAXCTL	311
PREFIX	311
CONVERSION	312
FIXEDOVERFLOW	312
DEFAULT	312
BYADDR または BYVALUE	312
(NON)CONNECTED	314
(NO)DESCRIPTOR.	314
(NO)INLINE.	314
LINKAGE	315
(RE)ORDER	315
NOOVERLAP	315
RETURNS(BYVALUE) または RETURNS(BYADDR).	315
パフォーマンスを向上させるコンパイラー・オプションの要約	315
パフォーマンス向上のためのコーディング.	316
DATA ディレクティブ入出力	316
入力専用パラメーター	317
GOTO ステートメント	317
ストリングの割り当て	317
ループ制御変数.	318
PACKAGE 対ネストされた PROCEDURE	318
ネストされたプロシージャの例.	319
パッケージ化されたプロシージャの例	319
REDUCIBLE 関数.	319
DESCLOCATOR または DESCLIST	320
DEFINED 対 UNION.	320
名前付き定数対静的変数.	320
意味のある名前が付けられていない最適なコードの例	321
意味のある名前が付けられた最適でないコードの例.	321
意味のある名前が付けられた最適なコードの例	322
ライブラリー・ルーチンの呼び出しの回避.	322
ライブラリー・ルーチンのプリロード	323

第 11 章 パフォーマンスの向上

ユーザーのプログラムの速度を向上することに関する多数の考慮事項は、使用するコンパイラーとそれを実行するプラットフォームには関係ありません。しかし、この章では、考慮事項の中でも PL/I コンパイラーとそれによって生成されるコードに特有な考慮事項を識別して説明します。

最適なパフォーマンスのためのコンパイラー・オプションの選択

選択するコンパイラー・オプションに応じて、コンパイラーによって生成されるコードのパフォーマンスを大幅に向上できることがあります。ただし、ほとんどのパフォーマンスの考慮事項と同様に、オプションの選択にはトレードオフがあります。幸いなことに、コンパイラー・オプションはコマンド行または構成ファイルで指定できるため、ソース・コードを編集しなくても、コンパイル時オプションに伴うトレードオフについて比較検討することができます。

詳細を省きたい場合は、生成されるコードのパフォーマンスを向上させる最も簡単な方法として、次の (非デフォルトの) コンパイラー・オプションを指定する方法があります。

OPT(2) または OPT(3)
DFT(REORDER)

次のセクションでは、パフォーマンスの向上と、特定のコンパイラー・オプションに伴うトレードオフについて、より詳しく説明します。

OPTIMIZE

OPTIMIZE オプションを指定すると、プログラムの速度を上げることができます。このオプションを指定しないと、コンパイラーは基本的な最適化のみを実行します。

OPTIMIZE(2) を選択すると、コンパイラーは、パフォーマンスを改善するコードを生成するように指示されます。通常、結果として生成されるコードは、プログラムが NOOPTIMIZE を使ってコンパイルされるときより短くなります。しかし、場合によっては、長い命令シーケンスが、短い命令シーケンスよりも実行時間が短いこともあります。例えば、WHEN 文節の値にギャップが指定されている SELECT ステートメント用にブランチ・テーブルが作成された場合などがこれに該当します。生成される命令の数が増えた分は、通常、ほかの場所での命令の実行数が減ることによって相殺されます。

OPTIMIZE(3) を選択すると、コンパイラーは、パフォーマンスを改善するコードを生成するように指示されます。しかし、OPTIMIZE(3) オプションを指定すると、OPTIMIZE(2) を指定したときよりもコンパイルに長い (ときには非常に長い) 時間がかかります。

GONUMBER

このオプションを使用すると、結果として、デバッグに使われるステートメント番号表が作成されます。この追加情報は、デバッグ時に非常に役立つことがあります。ステートメント番号表を含めると、実行可能ファイルのサイズが大きくなります。実行可能ファイルが大きくなると、ロードに時間がかかります。

ARCH

ARCH オプションの最高値を使用すると、コンパイラーは z/OS のもとで使用できる最も幅広い命令セットから命令を選択できるので、最適なコードの生成が可能になります。

REDUCE

REDUCE オプションは、コンパイラーが、構造体に対するヌル・ストリングの割り当てを縮小して、単純なコピー操作にすることを認められることを指定します。その操作が、埋め込みバイトの上書きを意味するにすぎない場合もあります。

REDUCE オプションを使用すると、ヌル・ストリングを構造体に割り当てるために生成されるコードの行数が少なくなり、その結果として通常はコンパイルが高速になり、コードの実行速度が大きく向上します。しかし、埋め込みバイトはゼロにリセットされることがあります。

例えば次の構造体では、*field11* と *field12* の間に 1 バイトの埋め込みがあります。

```
dc1
1 sample ext,
  5 field10      bin fixed(31),
  5 field11      dec fixed(13),
  5 field12      bin fixed(31),
  5 field13      bin fixed(31),
  5 field14      bit(32),
  5 field15      bin fixed(31),
  5 field16      bit(32),
  5 field17      bin fixed(31);
```

ここで、割り当て *sample = ''*; について考えてみます。

NOREDUCE オプションを指定した場合、8 つの割り当てが生成されますが、埋め込みバイトは変更されません。

しかし、REDUCE を指定した場合、その割り当ては 3 演算に削減されます。

RULES

RULES サブオプションのほとんどは、特定のコーディング方法 (変数を宣言しないことなど) にフラグを立てる基準となる重大度だけに影響し、パフォーマンスには影響しません。ただし、次のサブオプションはパフォーマンスに影響します。

IBM/ANS

RULES(IBM) オプションを使うと、コンパイラーはスケールされた FIXED BINARY をサポートします。パフォーマンスについてより重大なことは、いくつかの操作により、コンパイラーは、スケールされた FIXED BINARY の結果を生成します。RULES(ANS) を指定すると、スケールされた FIXED BINARY はサポートされず、スケールされた FIXED BINARY の結果は生成されません。つまり、

RULES(ANS) を指定して生成されたコードは、常に RULES(IBM) を指定して生成されたコードと少なくとも同じ速さで実行され、場合によっては実行速度が速くなります。

例えば、次のような部分コードがあるとします。

```
dc1 (i,j,k) fixed bin(15);
.
.
.
i = j / k;
```

RULES(IBM) を指定した場合は、割り算の結果に属性 FIXED BIN(31,16) が含まれます。つまり、割り算の前にはシフト命令が必要で、割り当てを実行するためにさらにいくつかの命令が必要になります。

RULES(ANS) のもとでは、割り算の結果は属性 FIXED BIN(15,0) を持ちます。つまり、割り算の前にはシフトは必要なく、割り当てを実行するために追加の命令は必要ありません。

(NO)LAXCTL

RULES(LAXCTL) のもとでは、CONTROLLED 変数を固定エクステントによって宣言しても、異なるエクステントに割り振ることができます。

例えば、RULES(LAXCTL) を指定すると、次のような構造体を宣言できます。

```
dc1
  1 a controlled,
  2 b char(17),
  2 c char(29);
```

しかし、その後でこの構造体を次のように割り振ることが可能です。

```
allocate
  1 a,
  2 b char(170),
  2 c char(290);
```

このことは、パフォーマンスに非常に悪い結果をもたらします。その原因は、コンパイラーが構造体 A への参照やその構造体のメンバーへの参照を見つけたときに、構造体内のフィールドの長さ、次元、またはオフセットに関して何の情報もないと想定しなければならないからです。

一方、RULES(NOLAXCTL) オプションはこのようなコーディング方法を禁止します。RULES(NOLAXCTL) を指定した場合に、CONTROLLED 変数を可変エクステントに割り振るには、そのエクステントをアスタリスクまたは非定数式によって宣言する必要があります。したがって、RULES(NOLAXCTL) を指定し、固定エクステントを使用して CONTROLLED 変数を宣言すれば、コンパイラーはその変数を参照するためのコードをはるかに適切に生成できます。

PREFIX

このオプションは、選択した PL/I 条件がデフォルトにより使用可能になっているかどうかを判別します。PREFIX のデフォルトのサブオプションは、PL/I 言語定義に

適合するように設定されますが、デフォルトを指定変更すると、ユーザーのプログラムのパフォーマンスに大きい影響を与えることがあります。デフォルトのサブオプションは次のとおりです。

CONVERSION
INVALIDOP
FIXEDOVERFLOW
OVERFLOW
INVALIDOP
NOSIZE
NOSTRINGRANGE
NOSTRINGSIZE
NOSUBSCRIPTRANGE
UNDERFLOW
ZERODIVIDE

SIZE、STRINGRANGE、STRINGSIZE、または SUBSCRIPTRANGE サブオプションを指定すると、コンパイラーによってエクストラ・コードが生成されます。このコードは、他の方法では見つけるのが難しいソース内のいろいろな問題領域の位置を正確に示すのに役立ちます。ただし、この追加のコードにより、プログラムのパフォーマンスが大幅に低下することがあります。

CONVERSION

CONVERSION 条件を使用不可にすると、いくつかの文字から数値への変換がインラインで、ソースの妥当性検査が行われずに実行されます。したがって、NOCONVERSION を指定した場合も、プログラムのパフォーマンスに影響がありません。

FIXEDOVERFLOW

プラットフォームによっては、ハードウェアによって FIXEDOVERFLOW 条件が生じるため、コンパイラーはその検出に追加のコードを生成しなくてもよい場合があります。

DEFAULT

DEFAULT オプションを使うと、属性のデフォルトを選択できます。PREFIX オプションの場合と同様に、DEFAULT のサブオプションは、PL/I 言語定義に適合するように設定されます。デフォルトを変更すると、パフォーマンスに影響が及ぶ場合があります。

IBM/ANS および ASSIGNABLE/NONASSIGNABLE など、サブオプションのいくつかはプログラムのパフォーマンスに影響しません。ただし、ほかのすべてのサブオプションは、多かれ少なかれパフォーマンスに影響を与えることがあり、不適切に適用された場合は、プログラムが無効になることもあります。これらのサブオプションの中でも重要なものを次に説明します。

BYADDR または BYVALUE

DEFAULT(BYADDR) オプションが有効な場合は、入り口宣言の属性でほかのものが指定されていない限り、参照によって (PL/I の必要性に応じて) 引数が渡されます。参照によって引数が渡されると、変数そのものが渡されるときに、引数のアド

レスが、あるルーチン (呼び出しルーチン) から別のルーチン (呼び出されたルーチン) に渡されます。呼び出されたルーチン側に制御があるときに引数に変更されると、呼び出しルーチンの実行が再開されたときに呼び出しルーチンにそれが反映されます。

多くの場合、プログラム・ロジックは、参照によって渡される変数に応じて異なります。しかし、参照によって変数を渡すと、次の 2 つのようなパフォーマンスの低下が起きる場合があります。

1. そのパラメーターに対するすべての参照に、追加の命令が必要になる。
2. 変数のアドレスが別のルーチンに渡されると、コンパイラは、その変数に変更されるときを想定して、その変数の参照用のコードとして非常に保守的なコードを生成することを強制される。

したがって、プログラム・ロジック上で認められるときはいつでも、BYVALUE サブオプションを使い、値によってパラメーターを渡す必要があります。BYADDR 属性を使って、1 つのパラメーターが参照によって渡されるように指示するときでも、DEFAULT(BYVALUE) オプションを使って、ほかのすべてのパラメーターが値によって渡されるように設定できます。

プロシージャが BYADDR によって渡される 1 つのパラメーターだけを受け取り、それを変更する場合は、値によってそのパラメーターを受け取る関数にプロシージャを変換することを考慮してください。その関数は、パラメーターの更新値が含まれた RETURN ステートメントで終了します。

BYADDR パラメーターが指定されたプロシージャ:

```
a: proc( parm1, parm2, ..., parmN );

    dcl parm1 byaddr ...;
    dcl parm2 byvalue ...;
    .
    .
    .
    dcl parmN byvalue ...;

    /* program logic */

end;
```

BYVALUE パラメーターが指定された、より高速の同等な関数:

```
a: proc( parm1, parm2, ..., parmN )
    returns( ... /* attributes of parm1 */ );

    dcl parm1 byvalue ...;
    dcl parm2 byvalue ...;
    .
    .
    .
    dcl parmN byvalue ...;

    /* program logic */

    return( parm1 );

end;
```

(NON)CONNECTED

DEFAULT(NONCONNECTED) オプションは、コンパイラーが集合体パラメーターをすべて NONCONNECTED と想定していることを示します。NONCONNECTED 集合体パラメーターの要素に対する参照では、コンパイラーがパラメーターの記述子にアクセスするためのコードを生成する必要があります。これは、集合体が固定エクステントを使って宣言されている場合にも同様です。

集合体パラメーターに固定エクステントが指定されており、CONNECTED が指定されている場合、コンパイラーはこれらの命令を生成しません。したがって、アプリケーションが NONCONNECTED パラメーターを渡さない場合は、

DEFAULT(CONNECTED) オプションを使うとコードがより最適化されます。

(NO)DESCRIPTOR

DEFAULT(DESCRIPTOR) オプションは、デフォルトでは、ストリング、領域、または集合体パラメーター用に記述子が渡されることを示します。しかし、記述子は、パラメーターに非固定エクステントが指定されている場合、またはパラメーターが NONCONNECTED 属性を持つ配列である場合にだけ、使われます。この場合は、記述子を渡すのに必要な命令とスペースは利益を与えず、かなりの負担になります (構造化記述子のサイズは、構造体そのもののサイズよりも大きいことが多いためです)。したがって、PROCEDURE ステートメントと ENTRY 宣言で必要となるときにだけ DEFAULT(NODESCRIPTOR) を指定し、OPTIONS(DESCRIPTOR) を使うと、コードの実行がより最適化されます。

(NO)INLINE

サブオプション NOINLINE は、プロシーチャーと開始ブロックがインライン化されないように指示します。

インライン化は、最適化を指定するときだけに発生します。

ユーザー・コードをインライン化すると、関数呼び出しとリンケージのオーバーヘッドが除去され、関数のコードが最適化プログラムに公開されて、結果としてコード・パフォーマンスが向上します。インライン化は、関数のオーバーヘッドが無視できるようなものではないとき、例えば、関数がネストされたループ内で呼び出されるときなどに、最高の結果をもたらします。また、インライン化は、インライン化された関数によってさらに最適化の機会が与えられるとき、例えば、定数引数が使われるときなどにも、有益です。

ネストされていない多くのプロシーチャーが含まれたプログラムの場合は、次のようになります。

- プロシーチャーの規模が小さく、ほんのいくつかの場所からしか呼び出されない場合は、INLINE を指定することによってパフォーマンスを向上させることができます。
- プロシーチャーの規模が大きく、複数の場所から呼び出される場合は、インライン化によって、プログラム全体にわたってコードの重複が起こります。このようなプログラム・サイズの増大は、速度の増大を相殺します。この場合は、NOINLINE をデフォルトのままにして、個別に選択したプロシーチャーだけで OPTIONS(INLINE) を指定することをお勧めします。

インライン化を使う場合は、スタック・スペースを拡大する必要があります。関数が呼び出されると、そのローカル・ストレージが呼び出し時に割り振られ、呼び出し関数に戻るときに解放されます。その関数がインライン化されると、そのストレージは、それを呼び出す関数に入ったときに割り振られ、呼び出し関数が終了するまで解放されません。インライン化した関数のローカル・ストレージ用に、十分なスタック・スペースがあることを確認してください。

LINKAGE

このサブオプションにより、コンパイラーには、**OPTIONS** 属性の **LINKAGE** サブオプションまたは入りのオプションが指定されなかったときに使用するデフォルト・リンケージが通知されます。

コンパイラーは、それぞれが固有のパフォーマンス特性を持つ各種のリンケージをサポートします。外部エンティティー（例えば、オペレーティング・システムなど）によって提供された **ENTRY** を呼び出すときには、その **ENTRY** 用に事前に定義されたリンケージを使う必要があります。

ただし、独自のアプリケーションを作成するときには、リンケージ規則を選択できます。**OPTLINK** リンケージは、ほかのリンケージ規則よりも大幅にパフォーマンスを向上させるので、これを選択することをお勧めします。

(RE)ORDER

DEFAULT(ORDER) オプションは、**ORDER** オプションがすべてのブロックに適用されることを示します。つまり、**ON** ユニットで参照されるそのブロック（または **ON** ユニットから動的に派生するブロック）内の変数に最新の値が適用されることを示します。これにより、そのような変数でのほとんどすべての最適化が実際上禁止されます。したがって、プログラム・ロジックによって認められる場合は、**DEFAULT(REORDER)** を使って、上位コードを生成してください。

NOOVERLAP

DEFAULT(NOOVERLAP) オプションを指定すると、コンパイラーは割り当て時にソースとターゲットがオーバーラップしていないと想定するため、より小さく速いコードを生成することができます。

しかし、このオプションを使用する場合は、割り当て時にソースとターゲットがオーバーラップしていないことを確認してください。例えば、**DEFAULT(NOOVERLAP)** オプションの下では、この例での割り当ては無効になります。

```
decl c char(20);
substr(c,2,5) = substr(c,1,5);
```

RETURNS(BYVALUE) または RETURNS(BYADDR)

DEFAULT(RETURNS(BYVALUE)) オプションが有効なときには、**BYADDR** を指定しないすべての **RETURNS** 記述リストに、**BYVALUE** 属性が適用されます。つまり、これらの関数は、最適なコードを生成するために、可能なときにはレジスターに値を戻します。

パフォーマンスを向上させるコンパイラー・オプションの要約

要約すると、次に挙げるオプション（ユーザーのアプリケーションに適用できる場合）はパフォーマンスを向上させることができます。

```
OPTIMIZE(3)
ARCH(5)
REDUCE
RULES(ANS NOLAXCTL)
次のサブオプションが指定された DEFAULT
  BYVALUE
  CONNECTED
  NODESCRIPTOR
  INLINE
  LINKAGE(OPTLINK)
  REORDER
  NOOVERLAP
  RETURNS(BYVALUE)
```

パフォーマンス向上のためのコーディング

コードを作成するときには、指定されたタスクを実行するために適する方法が複数あるのが普通です。多くの重要な要素、例えば、読み易さや保守容易性などによって、コーディングするスタイルの選択は変わってきます。次のセクションでは、コーディングを行うときにプログラムのパフォーマンスに影響を及ぼす可能性がある選択肢について説明します。

DATA ディレクティブ入出力

デバッグに GET DATA ステートメントと PUT DATA ステートメントを使うと、非常に有効であることがあります。ただし、これらのステートメントを使うと、一般的にはパフォーマンスが低下という犠牲が伴います。このパフォーマンスの低下は、変数リストを使わずに GET DATA または PUT DATA を使うと、非常に大きくなります。

多くのプログラマーは、次の例に示すように、ON ERROR コード内で PUT DATA ステートメントを使います。

```
on error
begin;
  on error system;
  .
  .
  .
  put data;
  .
  .
  .
end;
```

この場合、PUT DATA ステートメントで、選択された変数のリストを組み込むことにより、プログラムがより最適化されます。

上記の例の ON ERROR ブロックには、PUT DATA ステートメントの前に ON ERROR システム・ステートメントが含まれています。これにより、PUT DATA ステートメントでエラーが起きても（このエラーは、リストされる変数に無効な

FIXED DECIMAL 値が含まれている場合に起きる可能性がある)、ON ERROR ブロックのほかの場所でエラーが起きても、プログラムが無限ループに入ることが回避されます。

入力専用パラメーター

プロシージャに、入力専用に使われる BYADDR パラメーターが含まれている場合は、そのパラメーターを NONASSIGNABLE として宣言するのが (ASSIGNABLE のデフォルト属性を取得させるのではなく) 最良の方法です。プロシージャがあとからそのパラメーターの定数を使って呼び出された場合、コンパイラーは静的ストレージに定数を入れ、その静的領域のアドレスを渡します。

この方法は、レジスターに渡すことができないストリングやその他のパラメーターに特に役立ちます (レジスターに渡すことができる入力専用パラメーターは、BYVALUE として宣言するのが最良です)。

例えば次の宣言では、getenv に対する最初のパラメーターが、入力専用の CHAR VARYINGZ ストリングです。

```
dc1 getenv      entry( char(*) varyingz nonasgn byaddr,
                      pointer byaddr )
                  returns( native fixed bin(31) optional )
                  options( nodescrptor );
```

ストリング「IBM_OPTIONS」が指定されてこの関数が呼び出されると、コンパイラーは、コンパイラー生成一時記憶域にそのストリングを割り当て、その領域のアドレスを渡すのではなく、ストリングのアドレスを渡すことができます。

GOTO ステートメント

別のブロック内のラベルまたはラベル変数を使う GOTO ステートメントは、コンパイラーが実行する最適化を厳しく制限します。ラベル配列が初期化され、暗黙的または明示的に AUTOMATIC と宣言された場合、その配列のエレメントへの GOTO は最適化の妨げとなります。しかし、配列が STATIC と宣言された場合は、コンパイラーがその配列に対して CONSTANT 属性を想定し、最適化は妨げられません。

ストリングの割り当て

あるストリングが別のストリングに割り当てられるときに、コンパイラーは次のことを確認します。

- ソースとターゲットがオーバーラップしている場合でも、ターゲットが正しい値を持っている。
- ソース・ストリングがターゲットよりも長い場合は、ソース・ストリングは切り捨てられる。

この確認は、いくつかの追加の命令が必要になるという犠牲を払って行われます。コンパイラーは、これらの追加の命令を必要なときにだけ生成しようとしますが、コンパイラーが必要でないということに確信を持ってない場合、ユーザーは、プログラマーとして、その追加の命令が必要ないということを知っていることが多いのです。例えば、ソースとターゲットが基底付き文字ストリングであって、ユーザーはそれらがオーバーラップすることがあり得ないことを知っている場合、コンパイラーであれば生成するように強制されるところを、PLIMOVE 組み込み関数を使ってその追加のコードを除去することができます。

次の例では、2 番目の代入ステートメント用に、より速いコードが生成されます。

```
dc1 based_Str   char(64) based( null() );
dc1 target_Addr pointer;
dc1 source_Addr pointer;

target_Addr->based_Str = source_Addr->based_Str;

call plimove( target_Addr, source_Addr, stg(based_Str) );
```

ユーザーがソースとターゲットがオーバーラップするのではないかと疑いがある場合、またはターゲットがソースを収容できる大きさであるかどうか疑われる場合は、ユーザーは PLIMOVE 組み込み関数を使ってはなりません。

ループ制御変数

プログラムのパフォーマンスは、ループ制御変数が次のリストにあるいずれかのタイプである場合に、向上します。ユーザーは、まれに必要な場合を除き、これ以外のタイプの変数を使うべきではありません。

ゼロのスケール因数を持つ FIXED BINARY
FLOAT
ORDINAL
HANDLE
POINTER
OFFSET

また、ループ制御変数が配列、構造体、または共用体のメンバーでない場合にも、パフォーマンスは向上します。ループ制御変数がこのようなメンバーである場合、コンパイラーは警告メッセージを出します。AUTOMATIC であり、ほかの目的で使われないループ制御変数は、コードの生成を最適化します。

ループ制御変数が FIXED BIN である場合は、精度が 31 で SIGNED である場合に、パフォーマンスが最高に達します。

プログラムがループ制御変数の値だけでなく、そのアドレスにも依存している場合は、パフォーマンスが低下します。例えば、ADDR 組み込み関数に変数に適用される場合、あるいは変数が BYADDR で別のルーチンに渡される場合などです。

PACKAGE 対ネストされた PROCEDURE

呼び出し側のネストされたプロシージャは、追加の隠しパラメーター (逆チェーン・ポインター) が渡されるように求めます。結果として、アプリケーションに含まれているネストされたプロシージャが少なくなればなるほど、実行速度は速くなります。

アプリケーションのパフォーマンスを向上させるには、ネストされたプロシージャの母娘の対を、パッケージ内部のレベル 1 の姉妹プロシージャに変換します。この変換は、ネストされたプロシージャが、その親プロシージャで宣言された自動かつ内部の静的変数に依存していない場合に可能です。

ネストされたプロシージャの例のプロシージャ b に、a で宣言された変数が使われていない場合は、両方のプロシージャを パッケージ化されたプロシージャの例に示されているパッケージに再編成することによって、これらのプロシージャのパフォーマンスを向上させることができます。

ネストされたプロシーチャーの例

```
a: proc;

  dcl (i,j,k) fixed bin;
  dcl ib      based fixed bin;
  .
  .
  .
  call b( addr(i) );
  .
  .
  .
  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;
end;
```

パッケージ化されたプロシーチャーの例

```
p: package exports( a );

  dcl ib      based fixed bin;

  a: proc;

    dcl (i,j,k) fixed bin;
    .
    .
    .
    call b( addr(i) );
    .
    .
    .
  end;

  b: proc( px );
    dcl px      pointer;
    display( px->ib );
  end;

end p;
```

REDUCIBLE 関数

REDUCIBLE は、引数 (1 つ以上) が変更されない限り、プロシーチャーまたは入り口を複数回呼び出す必要がないこと、およびプロシーチャーの呼び出しに副次作用がないことを示します。

例えば、変更されないデータに基づいて結果を計算するユーザー作成の関数には、REDUCIBLE が宣言されなければなりません。乱数や時刻などの、変更されるデータに基づいて結果を計算する関数は、IRREDUCIBLE として宣言する必要があります。

次の例では、REDUCIBLE が宣言の一部になっているため、 f が一度だけ呼び出されます。宣言に IRREDUCIBLE が使われていると、 f が 2 度呼び出されます。

```
dcl (f) entry options( reducible ) returns( fixed bin );

select;
  when( f(x) < 0 )
  .
  .
```

```
      .  
      when( f(x) > 0 )  
      .  
      .  
      .  
      otherwise  
      .  
      .  
      .  
end;
```

DESCLOCATOR または DESCLIST

DEFAULT(DESCLOCATOR) オプションが有効になっている場合、コンパイラーは従来のコンパイラーとほぼ同じように、記述子 (ストリングや構造体など) を必要とする引数を渡すために記述子ロケーターを使用します。記述子の概要、および記述子を渡す方法の詳細については、467 ページの『第 21 章 PL/I 記述子』を参照してください。

このオプションを使用すれば、エントリー・ポイントが宣言している引数をすべて渡さなくても、エントリー・ポイントを呼び出すことができます。

またこのオプションを使用すると、構造体を渡してからそれをポインターとして受け取るといった、あまり賢明ではないプログラミング方法を従来どおり使うことができます。

ただし、DEFAULT(DESCLOCATOR) を指定した場合にコンパイラーによって生成されるコードは、状況によっては DEFAULT(DESCLIST) の場合のコードよりもパフォーマンスが悪くなることがあります。

DEFINED 対 UNION

UNION 属性は、DEFINED 属性よりも強力で、より多くの機能を提供します。さらにコンパイラーは、共用体参照の場合、より優れたコードを生成します。

次の例では、変数の対 b3 と b4 が、b1 および b2 と同じ機能を実行しますが、コンパイラーは共用体の対の場合に、より最適化されたコードを生成します。

```
dc1 b1 bit(31);  
dc1 b2 bit(16) def b1;  
  
dc1  
  1 * union,  
  2 b3 bit(32),  
  2 b4 bit(16);
```

DEFINED 属性ではなく UNION を使うコードは、誤って解釈されることが少なくなります。共用体の中の変数宣言は 1 個所にあるので、共用体のメンバーが変更されたり、ほかのすべてのメンバーが変更されても、それを理解しやすくなっています。この動的変更は、DEFINED 変数を使う宣言では宣言ステートメントが複数行離れていることがあるので、認識が難しくなります。

名前付き定数対静的変数

名前付き定数は、VALUE 属性を使って変数を宣言することによって定義できます。INITIAL 属性を指定して静的変数を使い、変数を変更しない場合は、VALUE

属性を使って変数を名前付き定数として宣言する必要があります。コンパイラーは `NONASSIGNABLE` のスカラー `STATIC` 変数を、真の名前付き定数として扱いません。

コンパイラーは、コンパイル時に式が評価される時にはいつでも、より最適化されたコードを生成するので、名前付き定数を使って、読み易さを低下させずに効果的なコードを生成できます。例えば次の例では、`VERIFY` 組み込み関数の 2 通りの使用法を用いて、同一のオブジェクト・コードが生成されています。

```
dc1 numeric char value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );
```

次の例は、`VALUE` 属性を使って、読み易さを低下させずに最適なコードを取得できる方法を示しています。

意味のある名前が付けられていない最適なコードの例

```
dc1 x bit(8) aligned;

select( x );
  when( '01'b4 )
    .
    .
    .
  when( '02'b4 )
    .
    .
    .
  when( '03'b4 )
    .
    .
    .
end;
```

意味のある名前が付けられた最適でないコードの例

```
dc1 ( a1 init( '01'b4)
      ,a2 init( '02'b4)
      ,a3 init( '03'b4)
      ,a4 init( '04'b4)
      ,a5 init( '05'b4)
      ) bit(8) aligned static nonassignable;

dc1 x bit(8) aligned;

select( x );
  when( a1 )
    .
    .
    .
  when( a2 )
    .
    .
    .
  when( a3 )
    .
    .
    .
end;
```

意味のある名前が付けられた最適なコードの例

```
dc1 ( a1 value( '01'b4)
      ,a2 value( '02'b4)
      ,a3 value( '03'b4)
      ,a4 value( '04'b4)
      ,a5 value( '05'b4)
      ) bit(8);

dc1 x bit(8) aligned;

select( x );
when( a1 )
.
.
.
when( a2 )
.
.
.
when( a3 )
.
.
.
end;
```

ライブラリー・ルーチンの呼び出しの回避

ビット単位操作 (接頭部 NOT、2 項演算子 AND、2 項演算子 OR、および 2 項演算子 EXCLUSIVE OR) は、多くの場合、ライブラリー・ルーチンと呼ばい出すと評価されます。ただし、これらの操作は、次のいずれかの条件が真である場合に、ライブラリーが呼び出されずに処理されます。

- 両方のオペランドが bit(1) である
- 両方のオペランドの位置が合い、同じ固定長である

特定の割り当て、式、および組み込み関数参照の場合は、コンパイラーがライブラリー・ルーチンの呼び出しを生成します。これらの呼び出しを回避すると、一般的にコードの実行速度が速くなります。

コンパイラーがこのような呼び出しを生成する時点を判断の助けとして、ライブラリー・ルーチンを使って変換が行われるときにはいつでも、コンパイラーがメッセージを生成します。

コードが、REFER を使用して BASED 構造体の 1 つのメンバーを参照している場合、実行時にその構造体にマップするために、コンパイラーがライブラリー・ルーチンに対する 1 つ以上の呼び出しを生成することがあります。これらの呼び出しは効率を低下させる可能性があるため、コンパイラーがこれらの呼び出しを行う場合には、メッセージを出して、コード内の問題となる可能性のある場所を見つけることができるようにしています。

REFER を使用した BASED 構造体を使用するコードがあるにもかかわらず、コンパイラーがフラグを立て、このメッセージが出た場合は、* エクステント付きの対応する構造体を宣言するサブルーチンに構造体を受け渡すことによって、パフォーマンスの向上が可能になる場合があります。これにより、構造体はいったん CALL ステートメントでマップされますが、呼び出されたサブルーチンでアクセスされるときにさらに再マップされることはなくなります。

ライブラリー・ルーチンのプリロード

PL/I ライブラリーには、低レベル・システム入出力機能 (IBMPOIOA) で使用される 1 つの RMODE 24 ルーチンが含まれています。コードでレコード入出力を行う場合、または (STDSYS オプションを指定してコンパイルせずに) SYSPRINT を STREAM OUTPUT ファイルとして使用する場合には、このルーチンをプリロードするか (E)LPA に配置することによって、パフォーマンスが大幅に改善されます。

第 4 部 他の製品に対するインターフェースの使用

第 12 章 ソート・プログラムの使用	327	ステップ 3: PL/I プログラムの作成	369
ソート・プログラムの使用準備	328	ステップ 4: PL/I プログラムのコンパイルと リンク	371
ソート・タイプの選択	328	ステップ 5: サンプル・プログラムの実行	372
ソート・フィールドの指定	331	JNI サンプル・プログラム #2 - スtringの引き 渡し	372
ソートするレコードの指定	333	Java サンプル・プログラム #2 の作成	372
最大レコード長	334	ステップ 1: Java プログラムの作成	372
ソート・プログラムに必要なストレージの決定	334	ステップ 2: Java プログラムのコンパイル	373
主記憶域	334	ステップ 3: PL/I プログラムの作成	374
補助記憶域	334	ステップ 4: PL/I プログラムのコンパイルと リンク	375
ソート・プログラムの呼び出し	334	ステップ 5: サンプル・プログラムの実行	376
例 1	336	JNI サンプル・プログラム #3 - 整数の引き渡し	376
例 2	336	Java サンプル・プログラム #3 の作成	376
例 3	337	ステップ 1: Java プログラムの作成	376
例 4	337	ステップ 2: Java プログラムのコンパイル	379
例 5	337	ステップ 3: PL/I プログラムの作成	379
ソートが成功したかどうかの判別	337	ステップ 4: PL/I プログラムのコンパイルと リンク	380
ソート・プログラム用のデータ・セットの確立	338	ステップ 5: サンプル・プログラムの実行	381
ソート作業データ・セット	338	JNI サンプル・プログラム #4 - Java 呼び出し API	381
入力データ・セット	338	Java サンプル・プログラム #4 の作成	381
出力データ・セット	339	ステップ 1: Java プログラムの作成	381
チェックポイント・データ・セット	339	ステップ 2: Java プログラムのコンパイル	382
ソート・データの入出力	339	ステップ 3: PL/I プログラムの作成	382
データ入出力処理ルーチン	339	ステップ 4: PL/I プログラムのコンパイルと リンク	385
E15 - 入力処理ルーチン (ソート出口 E15)	340	ステップ 5: サンプル・プログラムの実行	385
E35 - 出力処理ルーチン (ソート出口 E35)	343	Java および PL/I の同等なデータ・タイプの判別	385
PLISRTA の呼び出し例	344		
PLISRTB の呼び出し例	346		
PLISRTC の呼び出し例	347		
PLISRTD の呼び出し例	348		
可変長レコードのソートの例	350		
第 13 章 C との ILC	353		
同等なデータ・タイプ	353		
単純なタイプの一致	353		
struct タイプの一致	354		
enum タイプの一致	354		
ファイル・タイプの一致	355		
C 関数の使用	355		
一致する単純パラメーター・タイプ	357		
一致する String・パラメーター・タイプ	359		
ENTRY を戻す関数	361		
リンケージ	362		
出力の共用	364		
要約	364		
第 14 章 Java とのインターフェース	367		
Java Native Interface (JNI) の概要	367		
JNI サンプル・プログラム #1 - 'Hello World'	368		
Java サンプル・プログラム #1 の作成	368		
ステップ 1: Java プログラムの作成	368		
ステップ 2: Java プログラムのコンパイル	369		

第 12 章 ソート・プログラムの使用

コンパイラーは、PLISRTx (x = A、B、C、または D) という名前のインターフェースを提供しますが、このインターフェースを用いれば、IBM 提供のソート・プログラムを使用できます。

ソート・プログラムを PLISRTx と併用するには、次の操作を行う必要があります。

1. PLISRTx のいずれかのエントリー・ポイントに対する呼び出しを組み込み、ソートされるフィールドの情報をそのエントリー・ポイントに渡します。この情報には、レコード長、使用ストレージの最大値、戻りコードとして使用する変数の名前、およびソートを実行するのに必要なその他の情報が含まれます。
2. JCL DD ステートメント内で、ソート・プログラムに必要なデータ・セットを指定します。

ソート・プログラムは、PL/I から使用されると、大量のソート・フィールド上の正常な長さのレコードすべてをソートします。ほとんどのタイプのデータは、昇順または降順でソートできます。ソートされるデータのソースは、データ・セットである場合と、ソートにレコードが必要になるたびにソート・プログラムによって呼び出されるユーザー作成の PL/I プロシージャである場合があります。同様に、ソートの宛先も、データ・セットでも、ソートされたレコードを処理する PL/I プロシージャでもかまいません。

PL/I プロシージャを使えば、ソート自体の前でも後でも処理を行えるので、1 回のソート・インターフェースの呼び出しで、ソート操作をすべて完了することができます。入出力を処理する PL/I プロシージャは、ソート・プログラム自身から呼び出されるため、事実上ソート・プログラムの一部となることを理解しておくことが大切です。

PL/I は、DFSORT™ と一緒に、または同じインターフェースを持ったプログラムと一緒に稼働することができます。DFSORT は、プログラム・プロダクト 5740-SM1 の 1 リリースです。DFSORT はプログラム論理を書く必要をなくすために使用できる多数の組み込み機能 (例えば、INCLUDE、OMIT、OUTREC および SUM ステートメント、プラス多数の ICETOOL オペレーター) を持っています。詳細については「*DFSORT Application Programming Guide*」を、またチュートリアルとしては「*Getting Started with DFSORT*」を参照してください。

次の資料は DFSORT に適用されます。DFSORT 以外のプログラムを使用することもできるため、実際の機能や制約事項はさまざまです。このような機能や制約事項に関しては、「*DFSORT Application Programming Guide*」または、ご使用になるソート製品用のこれに相当する資料を参照してください。

ソート・プログラムを使うには、ソース・プログラム内に正しい PL/I ステートメントを入れなければならない、また JCL 内で正しいデータ・セットを指定しなければならない。

ソート・プログラムの使用準備

ソート・プログラムを使用するには、まず必要とするソートのタイプ、データ内のソート・フィールドの長さフォーマット、データ・レコード長、ソートに使う補助記憶域と主記憶域の大きさを決めます。

使用する PLISRTx エントリー・ポイントを決定するには、未ソート・データのソースと、ソート済みデータの宛先を決める必要があります。データ・セットと PL/I サブルーチンのどちらかを選択します。データ・セットを使用する方がより分かりやすく、パフォーマンスも速くなります。PL/I サブルーチンを使用すると、より高い柔軟性と機能性が得られるため、データをソートする前に処理したり印刷することができ、ソート済みの形のまま直ちにデータを使用することができます。入力または出力の処理サブルーチンを使用するには、339 ページの『データ入出力処理ルーチン』に目を通す必要があります。

エントリー・ポイント、データのソース、宛先を次に示します。

エントリー・ポイント	ソース	宛先
PLISRTA	データ・セット	データ・セット
PLISRTB	サブルーチン	データ・セット
PLISRTC	データ・セット	サブルーチン
PLISRTD	サブルーチン	サブルーチン

使用するエントリー・ポイントを決定したならば、今度は、データ・セットに関して次の事項を決定する必要があります。

- ソート・フィールドの位置。これらは 1 つのレコード全体あるいはその任意の一部 (複数の部分も可能) となります。
- これらのフィールドが表すデータのタイプ (例えば文字または 2 進数など)。
- 各フィールドでのソートを昇順にするか、降順にするか。
- 同じレコードは、入力されたとおりの順序で保持するか、またはソート時に順序を変更することができるのか。

PLISRTx への最初の引数である SORT ステートメント上でこれらのオプションを指定します。上記の事項を決定したならば、ソートするレコードに関して次の 2 点を決定する必要があります。

- レコード・フォーマットが固定フォーマットであるか、可変フォーマットであるか。
- レコード長。これは可変フォーマットの最大長です。

これらを PLISRTx への 2 番目の引数である RECORD ステートメント上で指定します。

最後に、ソート・プログラム用に確保しようとする主記憶域と補助記憶域の大きさを決めなければなりません。詳細については、334 ページの『ソート・プログラムに必要なストレージの決定』を参照してください。

ソート・タイプの選択

ソート・プログラムを最大限活用するためには、ソート・プログラムの働きについても多少の知識が必要です。PL/I プログラム内で、ソート・インターフェース・サ

ブルーチン **PLISRTx** に対して **CALL** ステートメントを使うことによって、ソートを指定します。このサブブルーチンは $x=A, B, C$ 、および D の 4 つのエントリー・ポイントを持っています。それぞれが未ソート・データの異なるソースおよびソートが完了したときのデータの宛先を指定します。例えば、**PLISRTA** の呼び出しは、未ソート・データ (ソートへの入力) が、ある 1 つのデータ・セット上にあることを指定し、ソート済みデータ (ソートからの出力) を別のデータ・セットに置くことを指定します。 **CALL PLISRTx** ステートメントに含めなければならないものは、ソートしようとするデータ・セットに関するソート・プログラム情報を示した引数リスト、ソートを行うフィールド、使用可能なスペースの大きさ、ソートが成功したかまたは失敗したかを示すための戻りコードをソート・プログラムが入れる変数の名前、および使用可能な出力または入力の処理プロシージャがあればその名前です。

ソート・インターフェース・ルーチンは、ソート・プログラム用の引数リストを、**PLISRTx** 引数リストが提供する情報と、選択した **PLISRTx** エントリー・ポイントから作成します。次に、制御はソート・プログラムに移されます。出力または入力の処理ルーチンを指定していれば、それぞれの未ソートまたはソート済みレコードを処理するのに必要な回数だけ、処理ルーチンがソート・プログラムによって呼び出されます。ソート操作が完了するとソート・プログラムは、戻りコードでソートが成功したか失敗したかを知らせ、**PL/I** 呼び出しプロシージャに戻ります。なお、その戻りコードは、インターフェース・ルーチンに渡される引数のうちの 1 つに入れられています。次に戻りコードは、処理を継続すべきかどうかを判別するために、**PL/I** ルーチン中でテストすることができます。330 ページの図 46 は、この操作を示す単純化されたフローチャートです。

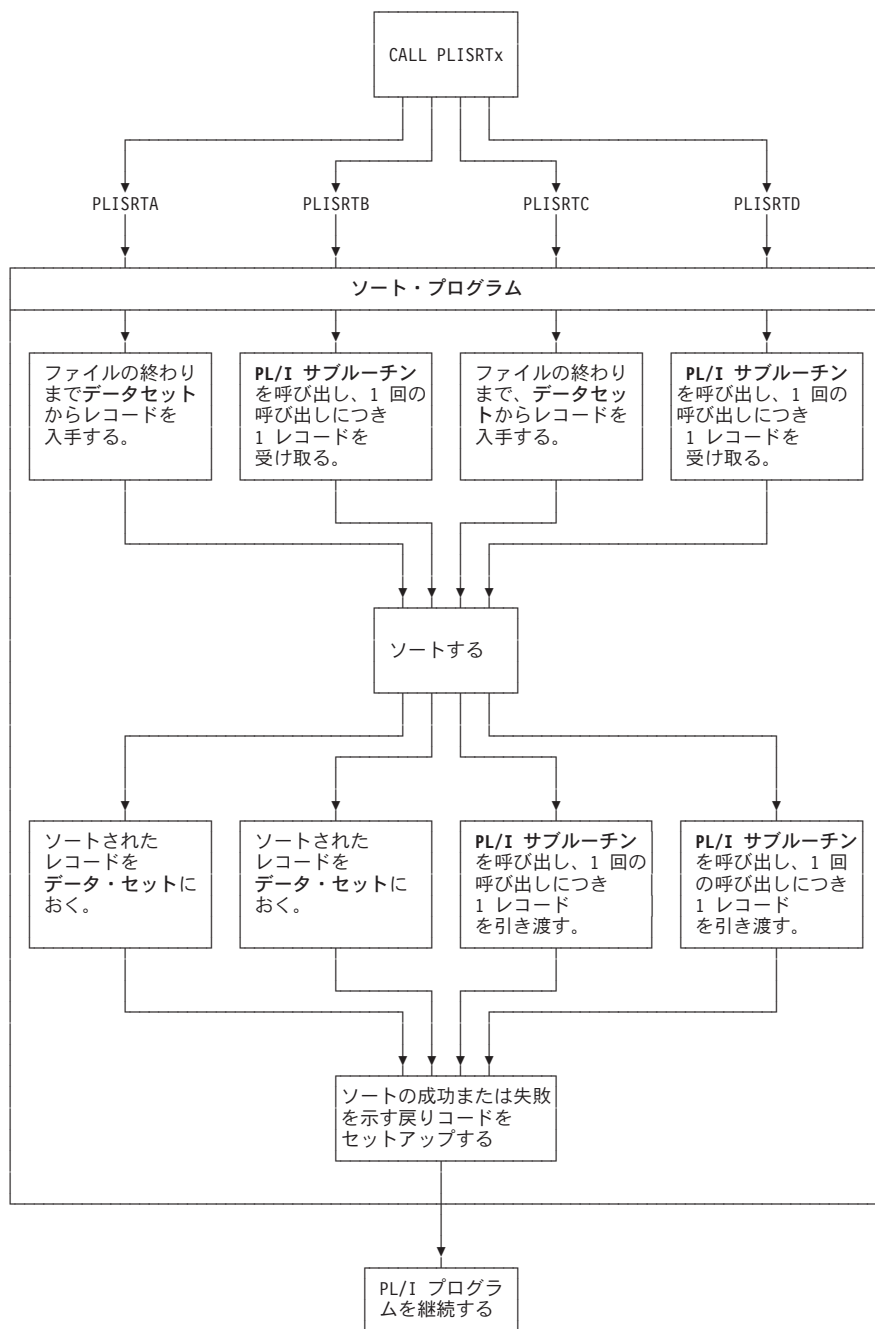


図 46. ソート・プログラムの制御の流れ

ソート・プログラムそのものにおいては、ソート・プログラムと入力および出力処理ルーチンとの間の制御の流れは、戻りコードで制御されます。ソート・プログラムは、その処理の途中で、適切な時点でこれらのルーチン呼び出します。(ソート・プログラムとそれに関連した資料では、これらのルーチンはユーザー出口と呼ばれます。ソートされる入力を渡すルーチンは、E15 ソート・ユーザー出口です。ソート済み出力を処理するルーチンは、E35 ソート・ユーザー出口です。)これらのルーチンから、ソート・プログラムは、そのルーチンをもう一度呼び出すべきか、または次の処理段階に進むべきかを示す戻りコードがくることを予期します。

ソート・プログラムに関して覚えておくべき重要点は次の 2 つです。(1) ソート・プログラムは、完全なソート操作を処理する自己完結型プログラムである。(2) ソート・プログラムは、呼び出し側との連絡、および呼び出しを行うユーザー出口との連絡を、戻りコードを使って行う。

この章の後半では、PL/I からどのようにソート・プログラムを使用するかについて詳しく説明します。まず、必要な PL/I ステートメントについて説明し、次にデータ・セット要件について説明します。この章の終わりには、ソート・インターフェース・ルーチンの 4 つのエントリー・ポイントを示す一連の例を挙げてあります。

ソート・フィールドの指定

`SORT` ステートメントは、`PLISRTx` への最初の引数です。`SORT` ステートメントの構文は、次の形式の文字ストリング式でなければなりません。

```
'bSORTbFIELDS=(start1,length1,form1,seq1,
...startn,lengthn,formn,seqn)[,other options]b'
```

次に例を示します。

```
' SORT FIELDS=(1,10,CH,A) '
```

b 1 つ以上のブランクを表します。ここに示されているブランクは必須です。これ以外のブランクは認められません。

start,length,form,seq

ソート・フィールドを定義します。指定するソート・フィールドの数は任意ですが、フィールドの全長には限度があります。複数のフィールドがソートされる場合は、レコードはまず最初のフィールドに従ってソートされ、次に、等しい値を持つレコードが 2 番目のフィールドに従ってソートされ、というようになります。ソート値がすべて等しければ、`EQUALS` オプションを使わない限り、等しいレコードの順序は任意になります。(この定義リストの後半を参照してください。) フィールドは互いにオーバーレイしてもかまいません。

`DFSORT (5740-SM1)` の場合、ソート・フィールドの最大全長は 4092 バイトに制限され、全ソート・フィールドはレコードの始まりから 4092 バイト以内でなくてはなりません。他のソート製品は異なる制限を持つ場合があります。

start レコード内の開始位置です。値はバイト単位で与えます (2 進数データの場合は、「バイト.ビット」表記を使うことができます)。ストリング内の最初のバイトはバイト 1 と見なされ、最初のビットはビット 0 と見なされます。(したがって、バイト 2 の 2 番目のビットは 2.1 と呼ばれます。) 可変長レコードの場合、4 バイト長の接頭部を入れて、5 をデータの最初のバイトにします。

length ソート・フィールドの長さです。値はバイト単位で指定します (ただし、2 進数データの場合は、「バイト.ビット」表記を使うことができます)。ソート・フィールドの長さには、それぞれのデータ・タイプ別に制約があります。

form データのフォーマットです。これは、ソートの目的で用いられるフォーマットです。`PL/I` ルーチンとソート・プログラム間でやり取りされるデータはすべて、文字ストリングの形式でなければなりません。主なデータ・タイプとその長さに対する制約事項を次に示します。これ以外のデータ・タイプも特殊目的のソート用に用意されています。これに関し

ては、「*DFSORT Application Programming Guide*」または、ご使用のソート製品用の資料を参照してください。

コード データ・タイプと長さ

CH 文字 1-4096
ZD ゾーン 10 進数符号付き 1 から 32 まで
PD パック 10 進符号付き 1 から 32 まで
FI 固定小数点、符号付き 1 から 256 まで
BI 2 進数、符号なし 1 ビットから 4092 バイトまで
FL 浮動小数点、符号付き 1 から 256 まで

全フィールドの合計長は、4092 バイトを超えてはなりません。

seq データが次のようにソートされる順序です。

A 昇順 (つまり、1,2,3,...)
D 降順 (つまり、...,3,2,1)

注: E を指定することはできません。その理由は、PL/I は、ユーザーが提供した順序を渡す手段を備えていないからです。

他のオプション

使用するソート・プログラムに応じて、ほかにいくつかのオプションを指定できます。それらのオプションは **FIELDS** オペランドとの間、およびオプション同士の間をコンマで区切らなければなりません。オペランドとオペランドの間にブランクを入れないでください。

FILSZ=y

ソートのレコード数を指定し、ソート・プログラムによる最適化を可能にします。y が単に近似値であれば、y の前に E を付けなければなりません。

SKIPREC=y

入力ファイルの始めにある y 個のレコードを無視して、残りのレコードをソートすることを指定します。

CKPT または CHKPT

チェックポイントをとることを指定します。このオプションを使用する場合は、**SORTCKPT** データ・セットを提供しなくてはならず、また **DFSORT** の **16NCKPT=NO** のインストール・オプションが指定されなくてはなりません。

EQUALS **NOEQUALS**

等しいレコードの順序を、入力したときと同じままにするか (**EQUALS**)、または任意にするか (**NOEQUALS**) を指定します。**NOEQUALS** を使えば、ソート・パフォーマンスを上げることができます。デフォルト・オプションは、ソート・プログラムがインストールされるときに選択されます。IBM 提供のデフォルトは、**NOEQUALS** です。

DYNALLOC=(d,n)

(OS/VS ソート・プログラムの場合のみ) プログラムは中間記憶装置を動的に割り振ることを指定します。

d 装置タイプ (3380 など)

ソートするレコードの指定

PLISRTx への 2 番目の引数として、RECORD ステートメントを使用してください。RECORD ステートメントの構文は、評価時には次に示す構文をとる文字ストリング式でなければなりません。

```
'bRECORDbTYPE=rectype[,LENGTH=(L1,[,,L4,L5])]b'
```

次に例を示します。

```
' RECORD TYPE=F,LENGTH=(80) '
```

- b** 1 つ以上のブランクを表します。ここに示されているブランクは必須です。これ以外のブランクは認められません。

TYPE

次のように、レコード・タイプを指定します。

- F** 固定長
- V** 可変長 EBCDIC
- D** 可変長 ASCII

未ソート・データとソート済みデータを処理するのに入力ルーチンと出力ルーチンを使用するときでも、ソート・プログラムが使用する作業データ・セットに適用されるレコード・タイプを指定しなければなりません。

可変長ストリングを入力ルーチン (E15 出口) からソート・プログラムに渡すときには、通常はレコード・フォーマットとして **V** を指定すべきです。ただし、**F** を指定すると、最大長になるまでレコードにブランクが埋め込まれます。

LENGTH

ソートするレコードの長さを指定します。PLISRTA や PLISRTC を使う場合、レコード長は入力データ・セットからとられるので、LENGTH は省略することができます。ソートできるレコードの最大長と最小長には、制限があることに注意してください。可変長のレコードの場合は、4 バイトの接頭部を含める必要があります。

- L1** ソートされるレコードの長さです。VSAM データ・セットを可変長レコードとしてソートする場合は、長さは最大レコード・サイズ +4 になります。
- „** PL/I から呼び出しを行うとき、ソート・プログラムに適用されない 2 つの引数を表します。その後に引数を使用する場合は、コンマを入れなければなりません。
- L4** 可変長レコードを使用するときの、最小レコード長を指定します。これを指定すると、この値は、ソート・プログラムによって最適化のために使用されます。
- L5** 可変長レコードを使用するときの、形式指定上の (最も一般的な) レコード長を指定します。これを指定すると、この値は、ソート・プログラムによって最適化のために使用されます。

最大レコード長

レコードの長さは、ユーザーが指定する最大長を超えることはありません。可変長レコードの最大レコード長は 32756 バイトで、固定長レコードの場合は 32760 バイトです。

ソート・プログラムに必要なストレージの決定

主記憶域

ソートには、主記憶域と補助記憶域の両方が必要です。DFSORT の最小主記憶域は 88K バイトですが、最大限のパフォーマンスのためにはこれより多くのストレージ (1 メガバイト強程度) をお勧めします。DFSORT は 16M より上の仮想記憶域あるいは拡張アーキテクチャー・プロセッサを利用します。z/OS のもとでは、DFSORT は拡張ストレージも利用できます。次のようにストレージ・パラメーターを渡せば、ソート・プログラムが使用可能なストレージを最大限使用するように指定することができます。

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B||UNSPEC('MAX');
CALL PLISRTA
  (' SORT FIELDS=(1,80,CH,A) ',
   ' RECORD TYPE=F,LENGTH=(80) ',
   MAXSTOR,
   RETCODE,
   'TASK');
```

ファイルを E15 または E35 出口ルーチン内でオープンする場合には、ファイルを正常にオープンできるよう、十分な残余ストレージを確保しておいてください。

補助記憶域

ある特定のソート操作の最小補助記憶域を計算するのは、複雑な作業です。補助記憶域によって最大限に効率を上げるには、できるかぎり直接アクセス記憶装置 (DASD) を使用します。プログラム効率向上に関する詳細は、「*DFSORT Application Programming Guide*」、特に DFSORT が必要な補助記憶域を決定して割り振ることを可能とする、動的ワークスペース割り振りに関する情報を参照してください。

ソートが確実に行われるように十分なストレージを提供することだけが目的の場合は、SORTWK データ・セットの合計サイズを、ソートするレコードを 3 セット保持できる大きさにします。(3 つのデータ・セット内に十分なスペースがあれば、3 より大きい数を指定してもとくに利益は得られません。)

ただし、この推奨値は近似値であるため、うまくいかないこともあるので、その場合には、ソートに関する資料を参照してください。この推奨値でうまくいった場合でも、スペースを無駄に使っている可能性があります。

ソート・プログラムの呼び出し

上記の各事項の決定が済めば、CALL PLISRTx ステートメントを作成する準備が整っています。この作成は注意して行う必要があります。エントリー・ポイントおよび使用する引数に関しては 335 ページの表 30 を参照してください。

表 30. 各エントリー・ポイントと *PLISRTx* ($x = A, B, C$, または D) への引数

エントリー・ポイント	引数
PLISRTA ソート入力: データ・セット ソート出力: データ・セット	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード [, データ・セット接頭部、メッセージ・レベル、ソート手法])
PLISRTB ソート入力: PL/I サブルーチン ソート出力: データ・セット	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード、 入力ルーチン [, データ・セット接頭部、メッセージ・レベル、ソート手法])
PLISRTC ソート入力: データ・セット ソート出力: PL/I サブルーチン	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード、 出力ルーチン [, データ・セット接頭部、メッセージ・レベル、ソート手法])
PLISRTD ソート入力: PL/I サブルーチン ソート出力: PL/I サブルーチン	(SORT ステートメント、RECORD ステートメント、ストレージ、戻りコード、入 力ルーチン、出力ルーチン[, データ・セット接頭部、メッセージ・レベル、ソート 手法])
SORT ステートメント	ソート・プログラムの SORT ステートメントが含まれた文字ストリング式。ソー ト・フィールドとフォーマットを記述する。331 ページの『ソート・フィールドの 指定』を参照。
RECORD ステートメント	ソート・プログラム RECORD ステートメントが含まれた文字ストリング式。デー タの長さレコード・フォーマットを記述する。333 ページの『ソートするレコー ドの指定』を参照。
ストレージ	ソート・プログラムで使用される主記憶域の最大値を示す固定 2 進式。DFSORT では、>88K バイトでなければならない。334 ページの『ソート・プログラムに必要な ストレージの決定』も参照。
戻りコード	精度 (31,0) の固定 2 進変数。ソート・プログラムが完了するとこの中に戻りコード が入る。戻りコードの意味は次のとおり。 0= ソート・プログラムは正常に完了 16= ソート・プログラムは失敗 20= ソート・プログラム・メッセージ・データ・セットが欠落
入力ルーチン	(PLISRTB および PLISRTD の場合のみ。) ソート・プログラムにレコードをソート 出口 E15 で渡すのに使用する PL/I 外部または内部プロシージャの名前。
出力ルーチン	(PLISRTC および PLISRTD の場合のみ。) ソートがソート出口 E35 でソート済み レコードを渡す PL/I 外部または内部プロシージャの名前。
データ・セット接頭部	データ・セット SORTIN、SORTOUT、SORTWKnn、または SORTCNTL を使用し た場合に、そのデータ・セット名内の 'SORT' デフォルト接頭部と置き換わる 4 文 字の文字ストリング式。このため、引数が "TASK" であれば、データ・セット TASKIN、TASKOUT、TASKWKnn、および TASKCNTL を使用することができる。 この機能により、同一ジョブ・ステップ内でソート・プログラムを複数呼び出すこ とができる。この 4 文字は英字で始まらなければならない、予約名 PEER、BALN、 CRCX、OSCL、POLY、DIAG、SYSC、または LIST のいずれであってもならない。 後続引数のうちのいずれかが必要だが、この引数は必要でない場合、この引数 にはヌル・ストリングをコーディングしなければならない。

表 30. 各エントリー・ポイントと *PLISRTx* ($x = A, B, C$, または D) への引数 (続き)

エントリー・ポイント	引数
メッセージ・レベル	<p>ソート・プログラムの診断メッセージの処理方法を示す次のような 2 文字の文字ストリング式。</p> <p>NO メッセージを SYSOUT へ出さない</p> <p>AP すべてのメッセージを SYSOUT へ送る</p> <p>CP 重大メッセージを SYSOUT へ送る</p> <p>SYSOUT は通常、プリンターに割り振られるため、簡略記号文字 “P” を使用。ソート・プログラムによっては、他のコードを使用することもできる。このようなコードに関する詳細は、「<i>DFSORT Application Programming Guide</i>」を参照。後続引数が必要だが、この引数は必要でない場合、この引数にはヌル・ストリングをコーディングしなければならない。</p>
ソート手法	<p>(これは DFSORT では使用されない。互換性のためだけに存在。) 次のように、行おうとするソートのタイプを示す長さ 4 の文字ストリング。</p> <p>PEER 対等機能ソート</p> <p>BALN 平衡</p> <p>CRCX 交差ソート</p> <p>OSCL 振動</p> <p>POLY 多相ソート</p> <p>通常、ソート・プログラムは、ユーザーの処置を必要としないで、使用可能なスペースの大きさを分析して最も効率のよい手法を選択する。この引数を使用するのは、ソート上の問題で迂回を行うときか、または、別の手法を使えばパフォーマンスが向上するのが確実なときに限らなければならない。詳細については「<i>DFSORT Application Programming Guide</i>」を参照。</p>

次の例は、CALL PLISRTx ステートメントが通常とる形式を示しています。

例 1

1048576 (1 メガバイト) のストレージと FIXED BINARY (31,0) と宣言された戻りコードの RETCODE を使って 80 バイトのレコードを SORTIN から SORTOUT へソートする PLISRTA への呼び出し。

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE);
```

例 2

この例は、入力、出力、および作業データ・セットが TASKIN、TASKOUT、TASKWK01 などという名前に変わっている点を除けば、例 1 と同じです。ソート・プログラムを 1 つのジョブ・ステップの中で 2 回呼び出したときに、このようになることがあります。

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE,
              'TASK');
```

例 3

ソートが 2 つのフィールドで行われることを除けばこの例も例 1 と同じです。最初に、文字であるバイト 1 から 10 までと、次に、それらが等しい場合は、2 進数フィールドが含まれているバイト 11 と 12 が昇順にソートされます。

```
CALL PLISRTA (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE);
```

例 4

この例は、PLISRTB の呼び出しを示しています。PL/I ルーチン PUTIN によって入力がソート・プログラムに渡され、80 バイト固定長レコードの文字 1 から 10 までに対してソートが実行されます。その他の詳細については上記と同様です。

```
CALL PLISRTB (' SORT FIELDS=(1,10,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE,
              PUTIN);
```

例 5

この例は、PLISRTD の呼び出しを示しています。PL/I ルーチン PUTIN によって入力が提供され、PL/I ルーチンの PUTOUT に出力が渡されます。ソートされるレコードは 84 バイト可変 (長さの接頭部を含む) です。データのバイト 1 から 5 までを昇順でソートし、次にこれらのフィールドが等しい場合は、バイト 6 から 10 までを降順でソートします。(4 バイト長さ接頭部が含まれているので、実際に開始点で使う値は 5 と 10 であることに注意してください。) 両方のフィールドが同じである場合は、入力の順序は保持されます。(これは EQUALS オプションによって行われます。)

```
CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(84) ',
              1048576,
              RETCODE,
              PUTIN,          /*input routine (sort exit E15)*/
              PUTOUT);       /*output routine (sort exit E35)*/
```

ソートが成功したかどうかの判別

ソート・プログラムはソートが完了すると、PLISRTx の呼び出しの 4 番目の引数内で指定した変数内に、戻りコードを設定します。次に、CALL PLISRTx ステートメントの後のステートメントに制御を戻します。戻された値は、次のようにソートが成功または失敗のどちらであったかを示します。

- 0 ソート・プログラムは正常に完了した
- 16 ソート・プログラムは失敗した
- 20 ソート・メッセージ・データ・セットが欠落している

戻りコードを渡す先の変数は FIXED BINARY (31,0) と宣言する必要があります。通常の作業では、CALL PLISRTx ステートメントの後で戻りコードの値をテストし、操作が成功したか失敗したかに応じて適切な処置をとります。

例を示します (戻りコードが RETCODE という名前であると仮定して)。

```

IF RETCODE≠0 THEN DO;
  PUT DATA(RETCODE);
  SIGNAL ERROR;
END;

```

ソート後のジョブ・ステップが、ソートの成功、失敗に依存している場合は、ソート・プログラム内で戻された値を、PL/I プログラムからの戻りコードとして設定する必要があります。これで、その戻りコードを次のジョブ・ステップに使用することができます。PL/I 戻りコードは、PLIRETC への呼び出して設定されます。PLIRETC を呼び出すには、次のようにソート・プログラムから戻された値を使うことができます。

```
CALL PLIRETC(RETCODE);
```

この PLIRETC の呼び出しを、ソート・プログラムへ制御情報を渡すのに戻りコードが使用される入出力ルーチン内での呼び出しと混同しないでください。

ソート・プログラム用のデータ・セットの確立

DFSORT が、システムが認識していないライブラリーへインストールされた場合は、DFSORT ライブラリーを JOBLIB または STEPLIB DD ステートメントで指定しなくてはなりません。

ソート・プログラムを呼び出すときには、特定のソート・データ・セットはオープンしてはなりません。それらは次のとおりです。

SYSOUT

ソート・プログラムからのメッセージが書き込まれるデータ・セット (通常はプリンター)

ソート作業データ・セット

SORTWK01-SORTWK32

注: 32 より多いソート作業データ・セットを指定した場合、DFSORT は最初の 32 個しか使用しません。

****WK01-****WK32

ソート処理で使用される 1 から 32 個までのデータ・セット。これらは直接アクセスでなければなりません。必要なスペースとデータ・セットの数については、334 ページの『ソート・プログラムに必要なストレージの決定』を参照してください。

**** は、PLISRTx への呼び出しにおいてデータ・セット接頭部引数として指定できる 4 文字を表します。この文字を使えば、SORT 以外の接頭部を使ってデータ・セットを使用することもできます。これは英字で始まらなければならず、名前は PEER、BALN、CRCX、OSCL、POLY、SYSC、LIST、または DIAG であってはなりません。

入力データ・セット

SORTIN

****IN

PLISRTA および PLISRTC を呼び出すときに使用する入力データ・セット。

詳細は前記 ****WK01-****WK32 を参照してください。

出力データ・セット

SORTOUT

****OUT

PLISRTA および PLISRTB を呼び出すときに使用する出力データ・セット。

詳細は前記 ****WK01-****WK32 を参照してください。

チェックポイント・データ・セット

SORTCKPT

CKPT または CHKPT オプションが SORT ステートメント引数中で使用され、DFSORT の 16NCKPT=NO のインストール・オプションが指定された場合は、チェックポイント・データを保持するために使用されるデータ・セット。このプログラム DD ステートメントに関する詳細は、「*DFSORT Application Programming Guide*」を参照してください。

DFSPARM SORTCNTL

追加の制御ステートメントまたは変更された制御ステートメントが入った (オプションの) データ・セット。このプログラム DD ステートメントに関する詳細は、「*DFSORT Application Programming Guide*」を参照してください。

詳細は前記 ****WK01-****WK32 を参照してください。

ソート・データの入出力

ソートするデータのソースは、データ・セットから直接提供される場合と、ユーザーが作成したルーチン (ソート出口 E15) によって間接的に提供される場合とがあります。同様に、ソートされた出力の宛先も、データ・セットである場合と、ユーザー提供のルーチン (ソート出口 E35) である場合とがあります。

PLISRTA は、データ・セットからデータ・セットにソートするものであるため、すべてのインターフェースの中で最も単純なインターフェースです。PLISRTA プログラムの例を 345 ページの図 50 に示します。ほかのインターフェースには、入力処理ルーチンまたは出力処理ルーチン、あるいはこの両方が必要です。

データ入出力処理ルーチン

PLISRTB、PLISRTC、または PLISRTD を使用するとき、ソート・プログラムはいくつかの入力処理ルーチンと出力処理ルーチンを呼び出します。これらのルーチンは、PL/I で作成する必要がある、内部プロシージャまたは外部プロシージャのいずれにもすることができます。入出力処理ルーチンが、PLISRTx を呼び出すルーチンに対して内部であれば、名前の有効範囲については、通常の内部プロシージャと同様に動作します。入力および出力プロシージャ名自体が、PLISRTx を呼び出すプロシージャ内で認識されていなければなりません。

これらのルーチンは、各レコードが、ソート・プログラムで必要になるか、ソート・プログラムから渡されるたびに個別に呼び出されます。したがって、各ルーチンは一度に 1 つのレコードを処理できるように作成しなければなりません。プロシージャ内で AUTOMATIC と宣言される変数は、呼び出しから次の呼び出しの間ではその値を保持しません。したがって、1 つの呼び出しから次の呼び出しへ保持されなければならないカウンタのような項目は、STATIC として宣言するか、収容ブロック内で宣言する必要があります。

E15 と E35 のソート出口は、MAIN プロシージャであってはなりません。

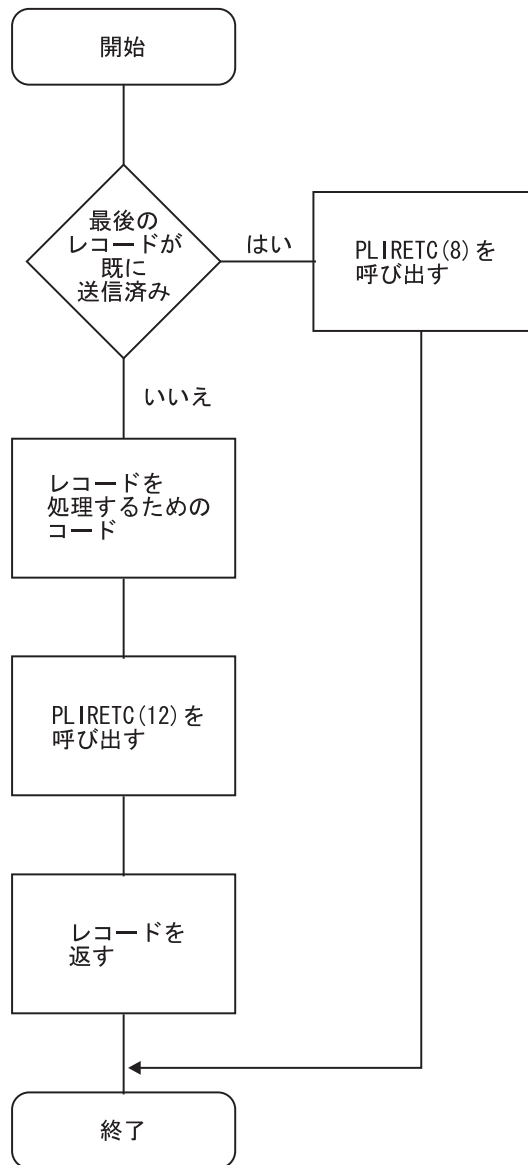
E15 - 入力処理ルーチン (ソート出口 E15)

入力ルーチンは、通常、データがソートされる前に、データに対して何らかの処理を加えるのに使用されます。入力ルーチンは、346 ページの図 51 および 348 ページの図 53 に示してあるとおり、データを印刷するのに使用するか、正しい結果を出すためにソート・フィールドを生成または操作するのに使用することができます。

入力処理ルーチンは、PLISRTB または PLISRTD を呼び出すときにソート・プログラムが使用します。ソート・プログラムがレコードを必要とする場合、戻りコード 12 と一緒に、文字ストリング・フォーマットのレコードを戻す入力ルーチン呼び出します。この戻りコードの意味は、渡したレコードはソートの対象になるということです。ソート・プログラムは、戻りコード 8 が渡されるまでこのルーチン呼び出し続けます。戻りコード 8 の意味は、すべてのレコードを渡し終えたので、ソート・プログラムがそのルーチンをもう呼び出す必要はないということです。戻りコードが 8 のときにレコードが戻された場合、そのレコードはソート・プログラムによって無視されます。

ルーチンによって戻されるデータは、文字ストリングでなければなりません。この文字ストリングは、固定でも可変でも構いません。可変の場合、PLISRTx への呼び出し内の 2 番目の引数である RECORD ステートメント内のレコード・フォーマットとして、通常は、V を指定する必要があります。しかし、F を指定することも可能で、その場合は、ストリングが最大長になるようブランクが埋め込まれます。レコードは RETURN ステートメントを使って戻されるため、PROCEDURE ステートメント内で RETURNS 属性を指定しなければなりません。戻りコードは、PLIRETC の呼び出し内で設定されます。代表的な入力ルーチンのフローチャートを、341 ページの図 47 に示します。

入力処理サブルーチン



出力処理サブルーチン



図 47. 入力および出力処理サブルーチンのフローチャート

代表的な入力ルーチンの骨組みコードを、342 ページの図 48 に示します。

```

E15: PROC RETURNS (CHAR(80));
      /*-----*/
      /*RETURNS attribute must be used specifying length of data to be */
      /* sorted, maximum length if varying strings are passed to Sort. */
      /*-----*/
      DCL STRING CHAR(80); /*-----*/
                           /*A character string variable will normally be*/
                           /* required to return the data to Sort          */
                           /*-----*/

      IF LAST_RECORD_SENT THEN
      DO;
      /*-----*/
      /*A test must be made to see if all the records have been sent, */
      /*if they have, a return code of 8 is set up and control returned*/
      /*to Sort                                                         */
      /*-----*/

      CALL PLIRETC(8); /*-----*/
                      /* Set return code of 8, meaning last record */
                      /* already sent.                               */
                      /*-----*/

      RETURN('');
      END;

      ELSE
      DO;
      /*-----*/
      /* If another record is to be sent to Sort, do the*/
      /* necessary processing, set a return code of 12 */
      /* by calling PLIRETC, and return the data as a */
      /* character string to Sort                      */
      /*-----*/

      ****(The code to do your processing goes here)

      CALL PLIRETC (12); /*-----*/
                        /* Set return code of 12, meaning this */
                        /* record is to be included in the sort */
                        /*-----*/

      RETURN (STRING); /*Return data with RETURN statement*/
      END;

      END; /*End of the input procedure*/

```

図 48. 入力プロシージャ用の骨組みコード

入力ルーチンの例は、346 ページの図 51 および 348 ページの図 53 に示してあります。

戻りコード 12 (ソートに現行レコードを組み込む) と戻りコード 8 (すべてのレコードを送付済み) 以外に、ソート・プログラムでは戻りコード 16 を使うこともできます。これは、ソートを終了させ、戻りコード 16 (ソート失敗) とともに、ソート・プログラムを PL/I プログラムに戻します。

注: PLIRETC への呼び出しは、PL/I プログラムから渡され、それ以降の任意のジョブ・ステップで 사용할 ことができる戻りコードを設定します。出力処理ルーチンを使用した後は、PLISRTx を呼び出してから PLIRETC を呼び出し、戻りコードをリセットして、ゼロ以外の完了コードが出ないようにすることをお勧めします。ソート・プログラムからの戻りコードを引数として使用して

PLIRETC を呼び出せば、PL/I 戻りコードにソートの成功または失敗を反映させることができます。この方法は、347 ページの図 52 に示してあります。

E35 - 出力処理ルーチン (ソート出口 E35)

出力処理ルーチンは通常、ソート後に必要なすべての処理に使われます。この処理は、347 ページの図 52 および 348 ページの図 53 に示してあるとおり、ソート済みデータを印刷することである場合も、そのソート済みデータを使ってさらに情報を生成することである場合もあります。出力処理ルーチンは、ソート・プログラムが PLISRTC または PLISRTD を呼び出すときに使用します。レコードのソートが終われば、ソート・プログラムはそれを一度に 1 つずつ、出力処理ルーチンに渡します。次に、出力ルーチンは、必要に応じてそれを処理します。すべてのレコードを渡し終わると、ソート・プログラムはその戻りコードをセットアップし、CALL PLISRTx ステートメントの後のステートメントに戻ります。ソート・プログラムから、最終レコードに達したことを出力処理ルーチンに知らせる指示はありません。したがって、データの終わり処理 (EOD) は、PLISRTx を呼び出すプロシージャで行わなければなりません。

レコードはソート・プログラムから出力ルーチンへ文字ストリングとして渡されるため、そのデータを受け取るには、出力処理サブルーチン内で文字ストリング・パラメーターを宣言しなければなりません。また、出力処理サブルーチンは、ソート・プログラムへ戻りコード 4 を渡して、別のレコードを処理する準備ができたことを通知する必要があります。この戻りコードは、PLIRETC への呼び出しによって設定します。

戻りコード 16 をソート・プログラムに渡せば、ソートを停止することができます。この場合、ソート・プログラムは戻りコード 16 (ソート失敗) とともに、呼び出し側プログラムに戻ることになります。

ソート・プログラムからルーチンに渡されるレコードは、文字ストリング・パラメーターです。PLISRTx への呼び出し内の 2 番目の引数でレコード・タイプを F と指定するときには、レコード長を持つパラメーターを宣言しなければなりません。レコード・タイプを V と指定するときには、次の例のように、パラメーターを調整可能と宣言する必要があります。

```
DCL STRING CHAR(*);
```

350 ページの図 54 は、可変長レコードをソートするためのプログラムを示しています。

典型的な出力処理ルーチンのフローチャートを、341 ページの図 47 に示してあります。典型的な出力処理ルーチンの骨組みコードは、344 ページの図 49 に示してあります。

```
E35: PROC(String);      /*The procedure must have a character string
                          parameter to receive the record from Sort*/

      DCL String CHAR(80); /*Declaration of parameter*/

      (Your code goes here)

      CALL PLIRETC(4);    /*Pass return code to Sort indicating that the next
                          sorted record is to be passed to this procedure.*/
      END E35;           /*End of procedure returns control to Sort*/
```

図 49. 出力処理プロシージャ用の骨組みコード

PLIRETC を呼び出すと、PL/I プログラムから渡され、その後の任意のジョブ・ステップで利用できる戻りコードが設定されることに注意しなければなりません。出力処理ルーチンを使用した後は、PLISRTx を呼び出してから PLIRETC を呼び出し、戻りコードをリセットして、ゼロ以外の完了コードが出ないようにすることをお勧めします。ソート・プログラムからの戻りコードを引数として使用して PLIRETC を呼び出せば、PL/I 戻りコードにソートの成功または失敗を反映させることができます。この方法は、この章の終わりにある例に示してあります。

PLISRTA の呼び出し例

PL/I 入力および出力処理ルーチンが、戻りコード情報をソート・プログラムに通知した後には、常に戻りコードのフィールドはゼロにリセットされます。したがって、ソート・プログラムの特定用途以外の通常の戻りコードとしては使用されません。

処理条件に関する詳細、特に入力および出力処理ルーチン時に生じる条件に関しては、「z/OS 言語環境プログラム プログラミングの手引き」を参照してください。

```

//OPT14#7 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
EX106: PROC OPTIONS(MAIN);
    DCL RETURN_CODE FIXED BIN(31,0);

    CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE);
    SELECT (RETURN_CODE);
    WHEN(0) PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
    WHEN(16) PUT SKIP EDIT
              ('SORT FAILED, RETURN_CODE 16') (A);
    WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
    OTHER PUT SKIP EDIT (
              'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
    END /* select */;
    CALL PLIRETC(RETURN_CODE);
    /*set PL/I return code to reflect success of sort*/
    END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*

```

図 50. *PLISRTA* - 入力データ・セットから出力データ・セットへのソート

PLISRTB の呼び出し例

```
//OPT14#8 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
EX107: PROC OPTIONS(MAIN);
      DCL RETURN_CODE FIXED BIN(31,0);

      CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X);
      SELECT(RETURN_CODE);
      WHEN(0) PUT SKIP EDIT
        ('SORT COMPLETE RETURN_CODE 0') (A);
      WHEN(16) PUT SKIP EDIT
        ('SORT FAILED, RETURN_CODE 16') (A);
      WHEN(20) PUT SKIP EDIT
        ('SORT MESSAGE DATASET MISSING ') (A);
      OTHER PUT SKIP EDIT
        ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
      END /* select */;
      CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

E15X:  /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT
      STREAM AND PUTS THEM BEFORE THEY ARE SORTED*/
      PROC RETURNS (CHAR(80));
      DCL SYSIN FILE RECORD INPUT,
        INFIELD CHAR(80);

      ON ENDFILE(SYSIN) BEGIN;
        PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
        CALL PLIRETC(8); /* signal that last record has
                           already been sent to sort*/

        INFIELD = '';
        GOTO ENDE15;
      END;

      READ FILE (SYSIN) INTO (INFIELD);
      PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
      CALL PLIRETC(12); /* request sort to include current
                           record and return for more*/

      ENDE15:
        RETURN(INFIELD);
      END E15X;
      END EX107;
/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
/*
//GO.SORTCNTL DD *
      OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

図 51. PLISRTB - 入力処理ルーチンから出力データ・セットへのソート

PLISRTC の呼び出し例

```
//OPT14#9 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
EX108: PROC OPTIONS(MAIN);

    DCL RETURN_CODE FIXED BIN(31,0);

    CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                  ' RECORD TYPE=F,LENGTH=(80) ',
                  1048576
                  RETURN_CODE,
                  E35X);
    SELECT(RETURN_CODE);
    WHEN(0) PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
    WHEN(16) PUT SKIP EDIT
              ('SORT FAILED, RETURN_CODE 16') (A);
    WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
    OTHER PUT SKIP EDIT
           ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
    END /* select */;
    CALL PLIRETC (RETURN_CODE);
    /*set PL/I return code to reflect success of sort*/

E35X: /* output handling routine prints sorted records*/
PROC (INREC);
    DCL INREC CHAR(80);
    PUT SKIP EDIT (INREC) (A);
    CALL PLIRETC(4); /*request next record from sort*/
END E35X;
END EX108;

/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

図 52. PLISRTC - 入力データ・セットから出力処理ルーチンへのソート

PLISRTD の呼び出し例

```
//OPT14#10 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
EX109: PROC OPTIONS(MAIN);
      DCL RETURN_CODE FIXED BIN(31,0);
      CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   1048576
                   RETURN_CODE,
                   E15X,
                   E35X);

      SELECT(RETURN_CODE);
      WHEN(0) PUT SKIP EDIT
              ('SORT COMPLETE RETURN_CODE 0') (A);
      WHEN(20) PUT SKIP EDIT
              ('SORT MESSAGE DATASET MISSING ') (A);
      OTHER PUT SKIP EDIT
              ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
      END /* select */;

      CALL PLIRETC(RETURN_CODE);
      /*set PL/I return code to reflect success of sort*/

E15X:  /* Input handling routine prints input before sorting*/
      PROC RETURNS(CHAR(80));
      DCL INFIELD CHAR(80);

      ON ENDFILE(SYSIN) BEGIN;
      PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT. ',
                      'SORTED OUTPUT SHOULD FOLLOW')(A);
      CALL PLIRETC(8); /* Signal end of input to sort*/
      INFIELD = '';
      GOTO ENDE15;
      END;

      GET FILE (SYSIN) EDIT (INFIELD) (A(80));
      PUT SKIP EDIT (INFIELD)(A);
      CALL PLIRETC(12); /*Input to sort continues*/
ENDE15:
      RETURN(INFIELD);
      END E15X;

E35X:  /* Output handling routine prints the sorted records*/
      PROC (INREC);

      DCL INREC CHAR(80);
      PUT SKIP EDIT (INREC) (A);
      NEXT: CALL PLIRETC(4); /* Request next record from sort*/
      END E35X;
END EX109;
```

図 53. PLISRTD - 入力処理ルーチンから出力処理ルーチンへのソート (1/2)

```
/*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
```

図 53. *PLISRTD* - 入力処理ルーチンから出力処理ルーチンへのソート (2/2)

可変長レコードのソートの例

```
//OPT14#11 JOB ...
//STEP1 EXEC IBMZCBG
//PLI.SYSIN DD *
/* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH
   RECORDS */

EX1306: PROC OPTIONS(MAIN);
  DCL RETURN_CODE FIXED BIN(31,0);
  CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
    ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
    /*NOTE THAT LENGTH IS MAX AND INCLUDES
      4 BYTE LENGTH PREFIX*/
    1048576
    RETURN_CODE,
    PUTIN,
    PUTOUT);

  SELECT(RETURN_CODE);
  WHEN(0) PUT SKIP EDIT (
    'SORT COMPLETE RETURN_CODE 0') (A);
  WHEN(16) PUT SKIP EDIT (
    'SORT FAILED, RETURN_CODE 16') (A);
  WHEN(20) PUT SKIP EDIT (
    'SORT MESSAGE DATASET MISSING ') (A);
  OTHER PUT SKIP EDIT (
    'INVALID RETURN_CODE = ', RETURN_CODE)
    (A,F(2));
  END /* SELECT */;

  CALL PLIRETC(RETURN_CODE);
  /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
  PUTIN: PROC RETURNS (CHAR(80) VARYING);
    /*OUTPUT HANDLING ROUTINE*/
    /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE
      WHEN USING VARYING LENGTH RECORDS*/
    DCL STRING CHAR(80) VAR;

    ON ENDFILE(SYSIN) BEGIN;
      PUT SKIP EDIT ('END OF INPUT')(A);
      CALL PLIRETC(8);
      STRING = '';
      GOTO ENDPUT;
    END;

    GET EDIT(STRING)(A(80));
    I=INDEX(STRING||' ',' ')-1; /*RESET LENGTH OF THE*/
    STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO */
                                /* LENGTH OF TEXT IN */
                                /* EACH INPUT RECORD.*/
```

図 54. 入出力処理ルーチンを使った可変長レコードのソート (1/2)

```

        PUT SKIP EDIT(I,STRING) (F(2),X(3),A);
        CALL PLIRETC(12);
ENDPUT: RETURN(STRING);
        END;
PUTOUT:PROC(STRING);
/*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
DCL STRING CHAR (*);
/*NOTE THAT FOR VARYING RECORDS THE STRING
PARAMETER FOR THE OUTPUT-HANDLING ROUTINE
SHOULD BE DECLARED ADJUSTABLE BUT CANNOT BE
DECLARED VARYING*/
PUT SKIP EDIT(STRING)(A); /*PRINT THE SORTED DATA*/
CALL PLIRETC(4);
END; /*ENDS PUTOUT*/
END;

/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//*/

```

図 54. 入出力処理ルーチンを使った可変長レコードのソート (2/2)

第 13 章 C との ILC

この章では、PL/I と C との間で行う言語間通信 (ILC) のいくつかの局面について説明します。両方の言語に共通するさまざまなデータ・タイプの使用法が例示されており、これらの例を参照して C との間で相互に呼び出しを行う PL/I コードを作成できます。

同等なデータ・タイプ

表 31 は、C と PL/I に共通する同等なデータ・タイプの一覧です。

表 31. C と PL/I の同等なタイプ

C のタイプ	同等な PL/I のタイプ
char[...]	char(...) varyingz
wchar[...]	wchar(...) varyingz
signed char	fixed bin(7)
unsigned char	unsigned fixed bin(8)
short	fixed bin(15)
unsigned short	unsigned fixed bin(16)
int	fixed bin(31)
unsigned int	unsigned fixed bin(32)
long long	fixed bin(63)
unsigned long long	unsigned fixed bin(64)
float	float bin(21)
double	float bin(53)
long double	float bin(p) (p >= 54)
enum	ordinal
typedef	define alias
struct	define struct
union	define union
struct *	handle

単純なタイプの一致

次の例は、C ヘッダー・ファイル *time.h* から抜粋した、単純な *time_t* の typedef の変換を示しています。

```
typedef long time_t;

define alias time_t fixed bin(31);
```

図 55. 単純なタイプの一致

struct タイプの一致

次の例は、C ヘッダー・ファイル *time.h* から抜粋した、単純な *tm* の struct の変換を示しています。

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};

define structure
1 tm
,2 tm_sec    fixed bin(31)
,2 tm_min    fixed bin(31)
,2 tm_hour   fixed bin(31)
,2 tm_mday   fixed bin(31)
,2 tm_mon     fixed bin(31)
,2 tm_year   fixed bin(31)
,2 tm_wday   fixed bin(31)
,2 tm_yday   fixed bin(31)
,2 tm_isdst  fixed bin(31)
;
```

図 56. struct タイプの一致の例

enum タイプの一致

次の例は、C ヘッダー・ファイル *stdio.h* から抜粋した、単純な enum *__device_t* の変換を示しています。

```

typedef enum {
    __disk      = 0,
    __terminal  = 1,
    __printer   = 2,
    __tape      = 3,
    __tdq       = 5,
    __dummy     = 6,
    __memory    = 8,
    __hfs       = 9,
    __hiperspace = 10
} __device_t;

#define ordinal __device_t (
    __disk      value(0)
    , __terminal value(1)
    , __printer  value(2)
    , __tape     value(3)
    , __tdq      value(4)
    , __dummy    value(5)
    , __memory   value(8)
    , __hfs      value(9)
    , __hiperspace value(10)
);

```

図 57. *enum* タイプの一致の例

ファイル・タイプの一致

C のファイル宣言はプラットフォームによって異なりますが、通常は次のように始まります。

```

struct __file {
    unsigned char * __bufPtr;
    ... } FILE;

```

図 58. *FILE* タイプの C 宣言の開始

必要なものはファイルのポインター (トークン) だけなので、この変換は次のようにうまく簡略化できます。

```

#define struct    1 file;
#define alias     file_Handle handle file;

```

図 59. C ファイルと一致する *PL/I*

C 関数の使用

C 関数 `fopen` と `fread` を使用して、ファイルを読み取ってフォーマット済み 16 進数としてダンプするプログラムを書きたいとしましょう。

このプログラムのコードは、次のように簡単なものです。

```
filedump:
  proc(fn) options(noexecops main);

  dcl fn          char(*) var;

  %include filedump;

  file = fopen( fn, 'rb' );

  if file = sysnull() then
    do;
      display( 'file could not be opened' );
      return;
    end;

  do forever;
    unspec(buffer) = 'b;

    read_In = fread( addr(buffer), 1, stg(buffer), file );

    if read_In = 0 then
      leave;

    display(   heximage(addr(buffer),16,' ') || ' '
              || translate(buffer,(32)'.',unprintable) );

    if read_In < stg(buffer) then
      leave;
    end;

    call fclose( file );
  end filedump;
```

図 60. *fopen* と *fread* を使用してファイルをダンプするコードの例

次のように、INCLUDE ファイル *filedump* の宣言のほとんどは自明なものです。

```
define struct      1 file;
define alias       file_Handle  handle file;

define alias       size_t unsigned fixed bin(32);
define alias       int signed fixed bin(31);

dcl file           type(file_Handle);
dcl read_In        fixed bin(31);
dcl buffer         char(16);

dcl unprintable    char(32) value( substr(collate(),1,32) );
```

図 61. *filedump* プログラムの宣言

一致する単純パラメーター・タイプ

C 関数の宣言の変換は間違いを起こしやすいかもしれません。例えば、次に示す C 関数 *fread* の宣言を考えてみましょう。

```
size_t  fread( void *,
               size_t,
               size_t,
               FILE *);
```

図 62. *fread* の C 宣言

これを次のように変換します。

```
dcl fread      ext
               entry( pointer,
                      type size_t,
                      type size_t,
                      type file_Handle )
               returns( type size_t );
```

図 63. *fread* の誤った宣言 (その 1)

プラットフォームによっては、C の名前に大文字小文字の区別があるため、これではリンクが正常に行われない場合があります。この種のリンカーの問題を防ぐために最適な方法は、*external* 属性の拡張形式を使用して、大文字小文字混合の名前を指定することです。したがって、例えば *fread* の宣言は次のように改良できます。

```
dcl fread      ext('fread')
               entry( pointer,
                      type size_t,
                      type size_t,
                      type file_Handle )
               returns( type size_t );
```

図 64. *fread* の誤った宣言 (その 2)

ただし、これは正しく実行されません。この原因は、PL/I パラメーターはデフォルトで *byaddr* になり、一方で C パラメーターはデフォルトで *byvalue* になるからです。このため、パラメーターに *byvalue* 属性を追加してこの問題を修正します。

```
dcl fread      ext('fread')
               entry( pointer byvalue,
                      type size_t byvalue,
                      type size_t byvalue,
                      type file_Handle byvalue )
               returns( type size_t );
```

図 65. *fread* の誤った宣言 (その 3)

ただし、図 66 で戻り値がどのように設定されているかに注意してください。4 番目のパラメーター（一時 `_temp5` のアドレス）が関数 `fread` に渡され、この関数はそのアドレスに整数の戻りコードを置くことになります。これは、`returns` に `byaddr` 属性が適用されている場合に値を戻す方法の規則で、PL/I はデフォルトでこの規則を使用します。

```

*      read_In = fread( addr(buffer), 1, stg(buffer), file );
*
L      r4,FILE(,r13,176)
L      r1,fread(,r5,12)
LA     r2,_temp5(,r13,420)
LA     r8,BUFFER(,r13,184)
L      r15,&EPA_&WSA(,r1,8)
L      r0,&EPA_&WSA(,r1,12)
ST     r0,_CEECAA_(,r12,500)
LA     r1,#MX_TEMP1(,r13,152)
ST     r8,#MX_TEMP1(,r13,152)
LA     r8,1
ST     r8,#MX_TEMP1(,r13,156)
ST     r7,#MX_TEMP1(,r13,160)
ST     r4,#MX_TEMP1(,r13,164)
ST     r2,#MX_TEMP1(,r13,168)
BALR   r14,r15
L      r0,_temp5(,r13,420)
ST     r0,READ_IN(,r13,180)

```

図 66. `RETURNS BYADDR` に対して生成されるコード

C の戻り値は `byvalue` なので、これは正しく実行されません。このため、`byvalue` 属性をもう 1 つ追加してこの問題を修正します。

```

dcl fread      ext('fread')
               entry( pointer byvalue,
                      type size_t byvalue,
                      type size_t byvalue,
                      type file_Handle byvalue )
               returns( type size_t byvalue );

```

図 67. `fread` の正しい宣言

359 ページの図 68 で戻り値がどのように設定されているかに注目してください。追加のアドレスは渡されず、戻り値がレジスター 15 に戻されるだけです。

```

*      read_In = fread( addr(buffer), 1, stg(buffer), file );
*
L      r2,FILE(,r13,176)
L      r1,fread(,r5,12)
LA     r7,BUFFER(,r13,184)
L      r15,&EPA_WSA(,r1,8)
L      r0,&EPA_WSA(,r1,12)
ST     r0,_CEECAA_(,r12,500)
LA     r1,#MX_TEMP1(,r13,152)
ST     r7,#MX_TEMP1(,r13,152)
LA     r7,1
ST     r7,#MX_TEMP1(,r13,156)
ST     r4,#MX_TEMP1(,r13,160)
ST     r2,#MX_TEMP1(,r13,164)
BALR   r14,r15
LR     r0,r15
ST     r0,READ_IN(,r13,180)

```

図 68. RETURNS BYVALUE に対して生成されるコード

一致するストリング・パラメーター・タイプ

fread を正しく変換したので、*fopen* に対してこの変換を試してみましょう。

```

dcl fopen    ext('fopen')
              entry( char(*) varyingz byvalue,
                    char(*) varyingz byvalue )
              returns( byvalue type file_handle );

```

図 69. *fopen* の誤った宣言 (その 1)

しかし、実際には C にストリングはない (ポインターだけ) ので、これらのポインターは *byvalue* として渡されます。このため、ストリングは次のように *byaddr* でなければなりません。

```

dcl fopen    ext('fopen')
              entry( char(*) varyingz byaddr,
                    char(*) varyingz byaddr )
              returns( byvalue type file_handle );

```

図 70. *fopen* の誤った宣言 (その 2)

しかし、PL/I はストリングとともに記述子を渡しますが、C は記述子を解釈できないので、記述子は抑止する必要があります。このためには、宣言に *options(nodescriptor)* を追加します。

```
dc1 fopen      ext('fopen')
               entry( char(*) varyingz byaddr,
                     char(*) varyingz byaddr )
               returns( byvalue type file_handle )
               options ( nodestructor );
```

図 71. *fopen* の正しい宣言

これで正しく働きますが、パラメーターは入力専用なので、この変換は最適ではありません。パラメーターが定数の場合は、*nonassignable* 属性を指定すれば、コピーの実行と引き渡しを防ぐことができます。したがって、*fopen* の宣言の最適な変換は次のとおりです。

```
dc1 fopen      ext('fopen')
               entry( char(*) varyingz nonasgn byaddr,
                     char(*) varyingz nonasgn byaddr )
               returns( byvalue type file_handle )
               options ( nodestructor );
```

図 72. *fopen* の正しく最適な宣言

この時点で、*fclose* 関数の宣言には、おそらく *returns* 指定の *optional* 属性を除けば、意外なものはほとんどありません。この属性を指定することで、CALL ステートメントで *fclose* 関数を呼び出せるようになり、戻りコードを処理する必要がなくなります。しかし、ここで注意しなければならないのが、ファイルが出力ファイルだった場合は、*fclose* の戻りコードを常に検査する必要があるという点です。最終バッファが書き出されるのがファイルがクローズされるときのみの場合があり、その書き込みがスペース不足のために失敗する可能性があるからです。

```
dc1 fclose     ext('fclose')
               entry( type file_handle byvalue )
               returns( optional type int byvalue )
               options ( nodestructor );
```

図 73. *fclose* の宣言

これで、z/OS UNIX 環境で、次のコマンドを使用してプログラムをコンパイルして実行できるようになりました。

```
pli -qdisplay=std filedump.pli

filedump filedump.pli
```

図 74. *filedump* をコンパイルして実行するためのコマンド

次の出力が生成されます。

```
15408689 938584A4 94977A40 97999683 . filedump: proc
4D86955D 409697A3 899695A2 4D959685 (fn) options(noe
A7858396 97A24094 8189955D 5E151540 xecops main);..
```

図 75. *filedump* の実行結果の出力

ENTRY を戻す関数

C のクイック・ソート関数 *qsort* は、比較ルーチンを使用します。例えば、整数の配列をソートするには、次の関数（前述の理由から、*byvalue* 属性を 2 回使用する）を使用できます。

```
comp2:
  proc( key, element )
  returns( fixed bin(31) byvalue );

  dcl (key, element) pointer byvalue;
  dcl word based fixed bin(31);

  select;
    when( key->word < element->word )
      return( -1 );
    when( key->word = element->word )
      return( 0 );
    otherwise
      return( +1 );
  end;
end;
```

図 76. C *qsort* 関数の比較ルーチンの例

次のコード・フラグメントに示すように、C *qsort* 関数とこの比較ルーチンを組み合わせて使用して、整数の配列をソートできます。

```
dcl a(1:4) fixed bin(31) init(19,17,13,11);

put skip data( a );

call qsort( addr(a), dim(a), stg(a(1)), comp2 );

put skip data( a );
```

図 77. C *qsort* 関数を使用するためのコード例

ただし、C 関数ポインターは PL/I ENTRY 変数と同じではないので、C *qsort* 関数を単純に次のように宣言してはなりません。

```
decl qsort      ext('qsort')
                entry( pointer,
                      fixed bin(31),
                      fixed bin(31),
                      entry returns( byvalue fixed bin(31) )
                )
                options( byvalue nodestructor );
```

図 78. *qsort* の誤った宣言

PL/I ENTRY 変数がネスト関数を指す場合があることに注意してください (このため、エントリー・ポイント・アドレスだけでなく逆チェーン・アドレスが必要です)。しかし、C 関数ポインターが指す先は非ネスト関数だけに限定されるので、PL/I ENTRY 変数と C 関数ポインターはストレージ容量を使用することすらありません。

ただし、C 関数ポインターは新しい PL/I タイプ LIMITED ENTRY と同等です。このため、C *qsort* 関数は次のように宣言できます。

```
decl qsort      ext('qsort')
                entry( pointer,
                      fixed bin(31),
                      fixed bin(31),
                      limited entry
                      returns( byvalue fixed bin(31) )
                )
                options( byvalue nodestructor );
```

図 79. *qsort* の正しい宣言

リンケージ

z/OS 上では、リンケージに関して次の 2 つの重要な事実があります。

- IBM C、JAVA、および Enterprise PL/I は、デフォルトで同じリンケージを使用します。
- このリンケージはシステム・リンケージではありません。

パラメーターがすべて *byaddr* である従来の PL/I アプリケーションの場合、関数にデフォルト・リンケージが使用される場合に生成されるコードと、システム・リンケージが使用される場合に生成されるコードの違いは、通常は問題になりませんでした。ただし、パラメーターが *byvalue* である場合 (C および JAVA ではこれが通常) は、この違いによってコードが使えなくなる可能性があります。

実際には、パラメーターが *byaddr* である場合の違いはほんの小さなものです。363 ページの図 80 では、デフォルト・リンケージを使用する関数に対して生成されたコードと、システム・リンケージを使用する関数に対して生成されたコードの違いは、システム・リンケージ呼び出しの最終パラメーターの高位ビットがオンになっていることです。

この違いは、ほとんどのプログラムには透過的です。

```
dc1 dftv  ext entry( fixed bin(31) byaddr
                    ,fixed bin(31) byaddr );
dc1 sysv  ext entry( fixed bin(31) byaddr
                    ,fixed bin(31) byaddr )
                    options( linkage(system) );

*      call dfta( n, m );
*
*      LA    r0,M(,r13,172)
*      LA    r2,N(,r13,168)
*      L     r15,V(DFTV)(,r3,126)
*      LA    r1,#MX_TEMP1(,r13,152)
*      ST    r2,#MX_TEMP1(,r13,152)
*      ST    r0,#MX_TEMP1(,r13,156)
*      BALR  r14,r15
*
*      call sysa( n, m );
*
*      LA    r0,M(,r13,172)
*      LA    r2,N(,r13,168)
*      O     r0,=X'80000000'
*      L     r15,V(SYSV)(,r3,130)
*      LA    r1,#MX_TEMP1(,r13,152)
*      ST    r2,#MX_TEMP1(,r13,152)
*      ST    r0,#MX_TEMP1(,r13,156)
*      BALR  r14,r15
```

図 80. パラメーターが *BYADDR* である場合のコード

ただし、パラメーターが *byaddr* でなく *byvalue* の場合は、大きな違いがあります。364 ページの図 81 で、デフォルト・リンケージを使用する関数の場合は、レジスター 1 は渡される整数の値を指します。一方、システム・リンケージを使用する関数の場合は、レジスター 1 はこれらの値のアドレスを指します。

この違いは、ほとんどのプログラムに透過的ではありません。

```

dcl dftv  ext entry( fixed bin(31) byvalue
                    ,fixed bin(31) byvalue );
dcl sysv  ext entry( fixed bin(31) byvalue
                    ,fixed bin(31) byvalue )
           options( linkage(system) );

*      call dftv( n, m );
*
*      L      r2,N(,r13,168)
*      L      r0,M(,r13,172)
*      L      r15,V(DFTV)(,r3,174)
*      LA     r1,#MX_TEMP1(,r13,152)
*      ST     r2,#MX_TEMP1(,r13,152)
*      ST     r0,#MX_TEMP1(,r13,156)
*      BALR   r14,r15
*
*      call sysv( n, m );
*
*      L      r1,N(,r13,168)
*      L      r0,M(,r13,172)
*      ST     r0,#wtemp_1(,r13,176)
*      LA     r0,#wtemp_1(,r13,176)
*      ST     r1,#wtemp_2(,r13,180)
*      LA     r2,#wtemp_2(,r13,180)
*      O      r0,=X'80000000'
*      L      r15,V(SYSV)(,r3,178)
*      LA     r1,#MX_TEMP1(,r13,152)
*      ST     r2,#MX_TEMP1(,r13,152)
*      ST     r0,#MX_TEMP1(,r13,156)
*      BALR   r14,r15

```

図 81. パラメーターが *BYVALUE* である場合のコード

出力の共用

C プログラムと *SYSPRINT* を共用したい場合は、*STDSYS* オプションを指定して *PL/I* コードをコンパイルする必要があります。

デフォルトでは、*DISPLAY* ステートメントは、出力を表示するために *WTO* を使用します。*DISPLAY(STD)* コンパイラー・オプションを指定すると、*DISPLAY* ステートメントは、C の *puts* 関数を使用して、出力を表示します。これは、*z/OS* UNIX 環境では特に便利です。

要約

この章で説明したことは、次のとおりです。

- C は大/小文字の区別がある。
- パラメーターは *BYVALUE* にする必要がある。
- 戻り値は *BYVALUE* にする必要がある。
- ストリング・パラメーターは *BYADDR* にする必要がある。
- 配列と構造体も *BYADDR* にする必要がある。
- 記述子を渡してはならない。

- 入力専用のストリングは NONASSIGNABLE にする必要がある。
- C 関数ポインターは LIMITED ENTRY にマップする。
- IBM C コンパイラーと IBM PL/I コンパイラーは、同じデフォルト・リンケージを使用する (このことが問題になる)。

第 14 章 Java とのインターフェース

この章では、Java と Java Native Interface (JNI) の概要を示し、JNI を PL/I と組み合わせて使用する場合の利点について説明します。単純な Java - PL/I アプリケーションを紹介し、また 2 つの言語間の互換性についても説明します。Java - PL/I サンプル・アプリケーションを構築して実行する方法の説明では、z/OS の z/OS UNIX System Services 環境で作業を行うことを前提にしています。

PL/I から Java とのやり取りを行うには、事前に z/OS システムに Java をインストールしておく必要があります。z/OS Java 環境のセットアップ方法の詳細については、該当のシステム管理者に問い合わせてください。

このサンプル・プログラムは、Java 2 バージョン 1.4.2 でコンパイルされ、テストされました。ご使用の z/OS UNIX System Services 環境の Java のレベルを判別するには、コマンド行から次のコマンドから入力してください。

```
java -version
```

アクティブな Java のバージョンが次のように表示されます。

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM z/OS Persistent Reusable VM build cm142)
```

Java Native Interface (JNI) の概要

Java は、Sun Microsystems によって開発されたオブジェクト指向プログラミング言語で、インターネット文書を対話式に作成するための強力な手段です。

Java Native Interface (JNI) は、ネイティブ・プログラミング言語に対する Java インターフェースで、Java Development Kit の一部です。JNI を利用するプログラムを作成すれば、さまざまなプラットフォーム間でコードを移植できるようになります。

JNI によって、Java 仮想マシン (JVM) 内で稼働する Java コードは、PL/I などの他言語で書かれたアプリケーションやライブラリーと相互運用できます。さらに、*Invocation API* を使用すれば、Java 仮想マシンをネイティブ PL/I アプリケーションに組み込むことができます。

Java は完成度の高いプログラム言語ですが、状況によっては他のプログラミング言語で書かれたプログラムを呼び出す必要も生じます。Java からこの呼び出しを行うには、ネイティブ・メソッドと呼ばれる、ネイティブ言語へのメソッド呼び出しを使用します。

ネイティブ・メソッドを使用する理由のいくつかを挙げます。

- アプリケーションのニーズを満たす Java クラス・ライブラリーにはない特殊な機能がネイティブ言語に備わっている。
- ネイティブ言語で書かれたアプリケーションが既に多数存在し、Java アプリケーションからこれらにアクセスできるようにしたい。

- ネイティブ言語で一連の複雑な計算を集中的にインプリメントし、これらの関数を Java アプリケーションから呼び出したい。
- ユーザーまたはプログラマーがネイティブ言語の幅広いスキルを持っていて、この利点を失いたくない。

JNI を介したプログラミングを行うことにより、ネイティブ・メソッドを使用してさまざまな操作を実行できます。ネイティブ・メソッドは次の操作を実行できます。

- Java メソッドがオブジェクトを使用する場合と同じ方法で Java オブジェクトを使用する。
- Java オブジェクト (配列やストリングなど) を作成し、これらのオブジェクトを検査して作業の実行に使用する。
- Java アプリケーション・コードによって作成されたオブジェクトを検査して使用する。
- 自身が作成した、または渡された Java オブジェクトを更新し、この更新済みオブジェクトを Java アプリケーションに提供する。

最後に、ネイティブ・メソッドは Java プログラミング・フレームワークに既に組み込まれている機能を利用して、既存の Java メソッドを呼び出すことも簡単にできます。このように、アプリケーションのネイティブ言語側と Java 側の両方で Java オブジェクトを作成、更新、および使用でき、さらにこれらのオブジェクトを相互間で共用できます。

JNI サンプル・プログラム #1 - 'Hello World'

Java サンプル・プログラム #1 の作成

最初のサンプル・プログラムは、"Hello World!" プログラムの一種です。

"Hello World!" プログラムには、1 つの Java クラス *callingPLI.java* があります。PL/I で書かれたネイティブ・メソッドは、*hiFromPLI.pli* に含まれています。このサンプル・プログラムを作成するための手順を簡単に概説します。

1. ネイティブ・メソッドを含むクラスを定義し、ネイティブ・ロード・ライブラリーをロードし、ネイティブ・メソッドを呼び出す Java プログラムを作成する。
2. Java プログラムをコンパイルして Java クラスを作成する。
3. ネイティブ・メソッドをインプリメントして "Hello!" テキストを表示する PL/I プログラムを作成する。
4. PL/I プログラムをコンパイルしてリンクする。
5. PL/I プログラム内のネイティブ・メソッドを呼び出す Java プログラムを実行する。

ステップ 1: Java プログラムの作成

ネイティブ・メソッドの宣言: Java メソッドであるかネイティブ・メソッドであるかに関係なく、メソッドはすべて Java クラス内で宣言する必要があります。Java メソッドとネイティブ・メソッドの宣言の違いは、キーワード *native* だけです。*native* キーワードは、このメソッドのインプリメンテーションのある場所が、プロ

グラムの実行時にロードされるネイティブ・ライブラリー内であることを Java に指示します。ネイティブ・メソッドの宣言は、次のとおりです。

```
public native void callToPLI();
```

上記のステートメントの中で、*void* はこのネイティブ・メソッド呼び出しから予期される戻り値がないことを示しています。メソッド名 *callToPLI()* の空括弧は、ネイティブ・メソッドの呼び出し時に渡すパラメーターがないことを示しています。

ネイティブ・ライブラリーのロード: ネイティブ・ライブラリーが実行時にロードされるように、ネイティブ・ライブラリーをロードするステップを組み込む必要があります。ネイティブ・ライブラリーをロードする Java ステートメントは、次のとおりです。

```
static {  
    System.loadLibrary("hiFromPLI");  
}
```

上記のステートメントでは、ネイティブ・ライブラリーを検索してロードするために、Java システム・メソッド *System.loadLibrary(...)* が呼び出されています。PL/I プログラムをコンパイルしてリンクするステップの実行中に、PL/I 共用ライブラリー *libhiFromPLI.so* が作成されます。

Java main メソッドの作成: *callingPLI* クラスには、クラスのインスタンスを生成してネイティブ・メソッドを呼び出す *main* メソッドも含まれています。*main* メソッドは *callingPLI* のインスタンスを生成し、*callToPLI()* ネイティブ・メソッドを呼び出します。

このセクションで前述した点をすべて含む *callingPLI* クラスの完全な定義は、次のとおりです。

```
public class callingPLI {  
    public native void callToPLI();  
    static {  
        System.loadLibrary("hiFromPLI");  
    }  
    public static void main(String[] argv) {  
        callingPLI callPLI = new callingPLI();  
        callPLI.callToPLI();  
        System.out.println("And Hello from Java, too!");  
    }  
}
```

ステップ 2: Java プログラムのコンパイル

Java コンパイラーを使用して *callingPLI* クラスをコンパイルし、実行可能形式にします。コマンドは次のとおりです。

```
javac callingPLI.java
```

ステップ 3: PL/I プログラムの作成

ネイティブ・メソッドの PL/I インプリメンテーションは、他の PL/I サブルーチンとほぼ同じようなものです。

便利な PL/I コンパイラー・オプション: サンプル・プログラムには、重要なコンパイラー・オプションを定義する一連の **PROCESS* ステートメントが含まれています。

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;  
*Process Display(Std) Dllinit Extrn(Short);  
*Process Rent Default( ASCII IEEE );
```

次に、これらのオプションの概要と利点を説明します。

Extname(100)

Java スタイルの長い外部名を許可します。

Margins(1,100)

マージンを拡張して、Java スタイルの名前と ID が入る場所を確保します。

Display(Std)

WTO を介さずに、"Hello World" テキストを stdout に書き込みます。 z/OS UNIX 環境では、WTO はユーザーによって意識されることはありません。

Dllinit

DLL の作成に必要な初期化コードをインクルードします。

Extrn(Short)

EXTRN は、参照された定数に対してだけ発行されます。このオプションは、Enterprise PL/I V3R3 以上が必要です。

Default(ASCII IEEE);

ASCII は、CHARACTER と PICTURE のデータを ASCII 形式 (JAVA での保持形式) で保持するように指定します。

IEEE は、FLOAT データを IEEE フォーマット (JAVA での保持形式) で保持するように指定します。

RENT

RENT オプションは、コードが静的変数に対して書き込みを行う場合にも、コードの再入可能性を確保します。

PL/I プロシージャ名とプロシージャ・ステートメントの正しい形式: PL/I プロシージャ名は、実行時に Java クラス・ローダーによって検出されるために、Java 命名規則に準拠している必要があります。Java 命名体系は 3 つの部分で構成されます。最初の部分は Java 環境に対してルーチンを識別し、2 番目の部分はネイティブ・メソッドを定義する Java クラスの名前、3 番目の部分はネイティブ・メソッド自体の名前です。

次に、サンプル・プログラムにある PL/I プロシージャ名 *Java_callingPLI_callToPLI* を分解して説明します。

Java

動的ライブラリー内にあるネイティブ・メソッドはすべて、*Java* を最初に指定する必要があります。

_callingPLI

ネイティブ・メソッドを宣言する Java クラスの名前。

_callToPLI

ネイティブ・メソッド自体の名前。

注: PL/I と C の間では、ネイティブ・メソッドのコーディングに重要な違いがあります。JDK に付属する *javah* ツールは、C プログラムに必要な外部参照形式

を生成します。ネイティブ・メソッドを PL/I で書き、前述した PL/I 外部参照の命名規則に準拠する場合は、PL/I ネイティブ・メソッドに対して *javah* ステップを実行する必要はありません。

また、PROCEDURE ステートメントの OPTIONS オプションに、以下のオプションを指定する必要があります。

- FromAlien
- NoDescriptor
- ByValue

サンプル・プログラムの完全なプロシージャ・ステートメントは、次のとおりです。

```
Java_callingPLI_callToPLI:
Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( FromAlien NoDescriptor ByValue );
```

JNI インクルード・ファイル: Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、*ibmzjni.inc* およびこれにインクルードされる *ibmzjnim.inc* です。これらのインクルード・ファイルは以下のステートメントでインクルードされます。

```
%include ibmzjni;
```

ibmzjni および *ibmzjnim* インクルード・ファイルは、PL/I **SIBMZSAM** データ・セットの中に提供されています。

完全な PL/I プロシージャ: 最後にまとめとして、ネイティブ・メソッドを定義する PL/I プログラム全体を示します。

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 );
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
PliJava_Demo: Package Exports(*);

Java_callingPLI_callToPLI:
Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( FromAlien NoDescriptor ByValue );

%include ibmzjni;
Dcl myJObject          Type jobject;

Display('Hello from Enterprise PL/I!');

End;
```

ステップ 4: PL/I プログラムのコンパイルとリンク

PL/I プログラムのコンパイル: 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

```
pli -c hiFromPLI.pli
```

共用ライブラリーのリンク: 次のコマンドを使用して、生成された PL/I オブジェクト・デックを共用ライブラリーにリンクします。

```
c89 -o libhiFromPLI.so hiFromPLI.o
```

PL/I 共用ライブラリーの名前には必ず *lib* 接頭部を付けてください。そうしないと、Java クラス・ローダーはライブラリーを検出できません。

ステップ 5: サンプル・プログラムの実行

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
java callingPLI
```

サンプル・プログラムの出力は次のとおりです。

```
Hello from Enterprise PL/I!  
And Hello from Java, too!
```

最初の行は PL/I ネイティブ・メソッドから書き込まれたものです。2 行目は、PL/I ネイティブ・メソッド呼び出しから戻った後、呼び出し側の Java クラスから書き込まれたものです。

JNI サンプル・プログラム #2 - スtringの引き渡し

Java サンプル・プログラム #2 の作成

このサンプル・プログラムは、Java と PL/I の間で双方向にStringの受け渡しを行います。*jPassString.java* プログラムの完全なリストは、373 ページの図 82 を参照してください。Java 部分には、1 つの Java クラス *jPassString.java* があります。PL/I で書かれたネイティブ・メソッドは、*passString.pli* に含まれています。最初のサンプル・プログラムで説明したこと多くが、このサンプル・プログラムにも当てはまります。このサンプル・プログラムについては、新しい面と異なる面だけを説明します。

ステップ 1: Java プログラムの作成

ネイティブ・メソッドの宣言: このサンプル・プログラムのネイティブ・メソッドは、次のとおりです。

```
public native void pliShowString();
```

ネイティブ・ライブラリーのロード: このサンプル・プログラムのネイティブ・ライブラリーをロードする Java ステートメントは、次のとおりです。

```
static {  
    System.loadLibrary("passString");  
}
```

Java main メソッドの作成: *jPassString* クラスには、クラスのインスタンスを生成してネイティブ・メソッドを呼び出す *main* メソッドも含まれています。*main* メソッドは *jPassString* のインスタンスを生成し、*pliShowString()* ネイティブ・メソッドを呼び出します。

このサンプル・プログラムは、Stringの入力をユーザーに促し、コマンド行からその値を読み込みます。この作業は、373 ページの図 82 に示す *try/catch* ステートメント内で行われます。

```

// Read a string, call PL/I, display new string upon return
import java.io.*;

public class jPassString{

    /* Field to hold Java string */
    String myString;

    /* Load the PL/I native library */
    static {
        System.loadLibrary("passString");
    }

    /* Declare the PL/I native method */
    public native void pliShowString();

    /* Main Java class */
    public static void main(String[] arg) {

        System.out.println(" ");

        /* Instantiate Java class and initialize string */
        jPassString myPassString = new jPassString();
        myPassString.myString = " ";

        /* Prompt user for a string */
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));

            /* Process until 'quit' received */
            while (!myPassString.myString.equalsIgnoreCase("quit")) {
                System.out.println(
                    "From Java: Enter a string or 'quit' to quit.");
                System.out.print("Java Prompt > ");
                /* Get string from command line */
                myPassString.myString = in.readLine();
                if (!myPassString.myString.equalsIgnoreCase("quit"))
                {
                    /* Call PL/I native method */
                    myPassString.pliShowString();
                    /* Return from PL/I and display new string */
                    System.out.println(" ");
                    System.out.println(
                        "From Java: String set by PL/I is: "
                        + myPassString.myString );
                }
            }
        } catch (IOException e) {
        }
    }
}

```

図 82. Java サンプル・プログラム #2 - スtringの引き渡し

ステップ 2: Java プログラムのコンパイル

Java コードをコンパイルするコマンドは、次のとおりです。

```
javac jPassString.java
```

ステップ 3: PL/I プログラムの作成

PL/I "Hello World" サンプル・プログラムの作成についての説明が、このプログラムにもすべて当てはまります。

PL/I プロシージャ名とプロシージャ・ステートメントの正しい形式: このプログラムの PL/I プロシージャ名は、*Java_jPassString_pliShowString* です。

サンプル・プログラムの完全なプロシージャ・ステートメントは、次のとおりです。

```
Java_jPassString_pliShowString:
  Proc( JNIEnv , myjobject )
    external( "Java_jPassString_pliShowString" )
    Options( FromAlien NoDescriptor ByValue );
```

JNI インクルード・ファイル: Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、*ibmzjni.inc* およびこれにインクルードされる *ibmzjnim.inc* です。これらのインクルード・ファイルは以下のステートメントでインクルードされます。

```
%include ibmzjni;
```

ibmzjni および *ibmzjnim* インクルード・ファイルは、PL/I **SIBMZSAM** データ・セットの中に提供されています。

完全な PL/I プロシージャ: 完全な PL/I プログラムは、375 ページの図 83 に示してあります。このサンプル PL/I プログラムは、JNI を介していくつかの呼び出しを行います。

開始時に、呼び出し側 Java オブジェクト *myObject* への参照が PL/I プロシージャに渡されます。PL/I プログラムはこの参照を使用して、呼び出し側からの情報を取得します。最初の情報は、*GetObjectClass* JNI 関数を使用して検索される呼び出し側オブジェクトのクラスです。このクラス値は、対象となる Java オブジェクト内の Java スtring・フィールドの ID を取得するために、*GetFieldID* JNI 関数によって使用されます。この Java フィールドは、フィールド名 *myString*、および JNI フィールド記述子 *Ljava/lang/String;* (フィールドが Java スtring・フィールドであることを示す) の指定によってさらに詳細に識別されます。その後、Java スtring・フィールドの値が *GetObjectField* JNI 関数を使用して検索されます。PL/I が Java スtring値を使用するには、事前にこの値をアンパックして PL/I が解釈できる形式にする必要があります。*GetStringUTFChars* JNI 関数を使用して、Java スtringが PL/I *varyingz* スtringに変換され、このスtringが PL/I プログラムによって表示されます。

取得した Java スtringを表示した後、PL/I プログラムは、呼び出し側 Java オブジェクト内のスtring・フィールドの更新に使用する PL/I スtringの入力をユーザーに促します。PL/I スtringの値は、*NewString* JNI 関数を使用して Java スtringに変換されます。この新しい Java スtringを使用して、*SetObjectField* JNI 関数によって呼び出し側 Java オブジェクト内のスtring・フィールドが更新されます。

PL/I プログラムが終了すると Java に制御が戻され、新しく更新された Java スtringが Java プログラムによって表示されます。

```

*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extn(Short);
*Process Rent Default( ASCII IEEE );
plijava_demo: package exports(*);

Java_passString_pliShowString:
Proc( JNIEnv , myJObject )
    external( "Java_jPassString_pliShowString" )
    Options( FromAlien NoDescriptor ByVal );

%include ibmzjni;

Dcl myBool          Type jBoolean;
Dcl myClazz         Type jclass;
Dcl myFID           Type jFieldID;
Dcl myJObject       Type jobject;
Dcl myJString       Type jString;
Dcl newJString      Type jString;
Dcl myID            Char(9)  Varz static init( 'myString' );
Dcl mySig           Char(18) Varz static
                    init( 'Ljava/lang/String;' );
Dcl pliStr          Char(132) Varz Based(pliStrPtr);
Dcl pliReply        Char(132) Varz;
Dcl pliStrPtr       Pointer;
Dcl nullPtr         Pointer;

Display(' ');

/* Get information about the calling Class */
myClazz = GetObjectClass(JNIEnv, myJObject);

/* Get Field ID for String field from Java */
myFID = GetFieldID(JNIEnv, myClazz, myID, mySig );

/* Get the Java String in the string field */
myJString = GetObjectField(JNIEnv, myJObject, myFID );

/* Convert the Java String to a PL/I string */
pliStrPtr = GetStringUTFChars(JNIEnv, myJString, myBool );

Display('From PLI: String retrieved from Java is: ' || pliStr );
Display('From PLI: Enter a string to be returned to Java:')
    reply(pliReply);

/* Convert the new PL/I string to a Java String */
newJString = NewString(JNIEnv, trim(pliReply), length(pliReply) );

/* Change the Java String field to the new string value */
nullPtr = SetObjectField(JNIEnv, myJObject, myFID, newJString);

End;

end;

```

図 83. PL/I サンプル・プログラム #2 - スtringの引き渡し

ステップ 4: PL/I プログラムのコンパイルとリンク

PL/I プログラムのコンパイル: 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

```
pli -c passString.pli
```

共用ライブラリーのリンク: 次のコマンドを使用して、生成された PL/I オブジェクト・デックを共用ライブラリーにリンクします。

```
c89 -o libpassString.so passString.o
```

PL/I 共用ライブラリーの名前には必ず *lib* 接頭部を付けてください。そうしないと、Java クラス・ローダーはライブラリーを検出できません。

ステップ 5: サンプル・プログラムの実行

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
java jPassString
```

Java と PL/I の両方からのユーザー入力プロンプトを含めた、サンプル・プログラムの出力は次のとおりです。

```
>java jPassString
```

```
From Java: Enter a string or 'quit' to quit.  
Java Prompt > A string entered in Java
```

```
From PLI: String retrieved from Java is: A string entered in Java  
From PLI: Enter a string to be returned to Java:  
A string entered in PL/I
```

```
From Java: String set by PL/I is: A string entered in PL/I  
From Java: Enter a string or 'quit' to quit.  
Java Prompt > quit  
>
```

JNI サンプル・プログラム #3 - 整数の引き渡し

Java サンプル・プログラム #3 の作成

このサンプル・プログラムは、Java と PL/I の間で双方向に整数の受け渡しを行います。*jPassInt.java* プログラムの完全なリストは、378 ページの図 84 を参照してください。Java 部分には、1 つの Java クラス *jPassInt.java* があります。PL/I で書かれたネイティブ・メソッドは、*passInt.pli* に含まれています。最初のサンプル・プログラムで説明したこと多くが、このサンプル・プログラムにも当てはまります。このサンプル・プログラムについては、新しい面と異なる面だけを説明します。

ステップ 1: Java プログラムの作成

ネイティブ・メソッドの宣言: このサンプル・プログラムのネイティブ・メソッドは、次のとおりです。

```
public native void pliShowInt();
```

ネイティブ・ライブラリーのロード: このサンプル・プログラムのネイティブ・ライブラリーをロードする Java ステートメントは、次のとおりです。

```
static {  
    System.loadLibrary("passInt");  
}
```

Java main メソッドの作成: *jPassInt* クラスには、クラスのインスタンスを生成してネイティブ・メソッドを呼び出す *main* メソッドも含まれています。 *main* メソッドは *jPassInt* のインスタンスを生成し、 *pliShowInt()* ネイティブ・メソッドを呼び出します。

このサンプル・プログラムは、整数の入力をユーザーに促し、コマンド行からその値を読み込みます。この作業は、378 ページの図 84 に示す *try/catch* ステートメント内で行われます。

```

// Read an integer, call PL/I, display new integer upon return
import java.io.*;
import java.lang.*;

public class jPassInt{

    /* Fields to hold Java string and int */
    int myInt;
    String myString;

    /* Load the PL/I native library */
    static {
        System.loadLibrary("passInt");
    }

    /* Declare the PL/I native method */
    public native void pliShowInt();

    /* Main Java class */
    public static void main(String[] arg) {

        System.out.println(" ");

        /* Instantiate Java class and initialize string */
        jPassInt pInt = new jPassInt();
        pInt.myInt = 1024;
        pInt.myString = " ";

        /* Prompt user for an integer */
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));

            /* Process until 'quit' received */
            while (!pInt.myString.equalsIgnoreCase("quit")) {
                System.out.println
                    ("From Java: Enter an Integer or 'quit' to quit.");
                System.out.print("Java Prompt > ");
                /* Get string from command line */
                pInt.myString = in.readLine();
                if (!pInt.myString.equalsIgnoreCase("quit"))
                {
                    /* Set int to integer value of String */
                    pInt.myInt = Integer.parseInt( pInt.myString );
                    /* Call PL/I native method */
                    pInt.pliShowInt();
                    /* Return from PL/I and display new string */
                    System.out.println(" ");
                    System.out.println
                        ("From Java: Integer set by PL/I is: " + pInt.myInt );
                }
            }
        } catch (IOException e) {
        }
    }
}

```

図 84. Java サンプル・プログラム #3 - 整数の引き渡し

ステップ 2: Java プログラムのコンパイル

Java コードをコンパイルするコマンドは、次のとおりです。

```
javac jPassInt.java
```

ステップ 3: PL/I プログラムの作成

PL/I "Hello World" サンプル・プログラムの作成についての説明が、このプログラムにもすべて当てはまります。

PL/I プロシージャ名とプロシージャ・ステートメントの正しい形式: このプログラムの PL/I プロシージャ名は、*Java_jPassInt_pliShowInt* です。

サンプル・プログラムの完全なプロシージャ・ステートメントは、次のとおりです。

```
Java_passNum_pliShowInt:
Proc( JNIEnv , myjobject )
    external( "Java_jPassInt_pliShowInt" )
    Options( FromAlien NoDescriptor ByValue );
```

JNI インクルード・ファイル: Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、*ibmzjni.inc* およびこれにインクルードされる *ibmzjnim.inc* です。これらのインクルード・ファイルは以下のステートメントでインクルードされます。

```
%include ibmzjni;
```

ibmzjni および *ibmzjnim* インクルード・ファイルは、PL/I SIBMZSAM データ・セットの中に提供されています。

完全な PL/I プロシージャ: 完全な PL/I プログラムは、380 ページの図 85 に示してあります。このサンプル PL/I プログラムは、JNI を介していくつかの呼び出しを行います。

開始時に、呼び出し側 Java オブジェクト *myObject* への参照が PL/I プロシージャに渡されます。PL/I プログラムはこの参照を使用して、呼び出し側からの情報を取得します。最初の情報は、*GetObjectClass* JNI 関数を使用して検索される呼び出し側オブジェクトのクラスです。このクラス値は、対象となる Java オブジェクト内の Java 整数フィールドの ID を取得するために、*GetFieldID* JNI 関数によって使用されます。この Java フィールドは、フィールド名 *myInt*、および JNI フィールド記述子 *I* (フィールドが整数フィールドであることを示す) の指定によってさらに詳細に識別されます。その後、Java 整数フィールドの値が *GetIntField* JNI 関数を使用して検索され、PL/I プログラムによって表示されます。

取得した Java 整数を表示した後、PL/I プログラムは、呼び出し側 Java オブジェクト内の整数フィールドの更新に使用する PL/I 整数の入力をユーザーに促します。この PL/I 整数値を使用して、*SetIntField* JNI 関数によって呼び出し側 Java オブジェクトの整数フィールドが更新されます。

PL/I プログラムが終了すると Java に制御が戻され、新しく更新された Java 整数が Java プログラムによって表示されます。

```

*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
*Process Display(Std) Dllinit Extrn(Short);
*Process Rent Default( ASCII IEEE );
plijava_demo: package exports(*);

Java_passNum_pliShowInt:
Proc( JNIEnv , myjobject )
  external( "Java_jPassInt_pliShowInt" )
  Options( FromAlien NoDescriptor ByValue );

%include ibmzjni;

Dcl myClazz          Type jclass;
Dcl myFID             Type jFieldID;
Dcl myJInt            Type jint;
dcl rtnJInt           Type jint;
Dcl myJObject         Type jobject;
Dcl pliReply          Char(132) Varz;
Dcl nullPtr           Pointer;

Display(' ');

/* Get information about the calling Class */
myClazz = GetObjectClass(JNIEnv, myJObject);

/* Get Field ID for int field from Java */
myFID = GetFieldID(JNIEnv, myClazz, "myInt", "I");

/* Get Integer value from Java */
myJInt = GetIntField(JNIEnv, myJObject, myFID);

display('From PLI: Integer retrieved from Java is: ' || trim(myJInt) );
display('From PLI: Enter an integer to be returned to Java:')
  reply(pliReply);

rtnJInt = pliReply;

/* Set Integer value in Java from PL/I */
nullPtr = SetIntField(JNIEnv, myJObject, myFID, rtnJInt);

End;

end;

```

図 85. PL/I サンプル・プログラム #3 - 整数の引き渡し

ステップ 4: PL/I プログラムのコンパイルとリンク

PL/I プログラムのコンパイル: 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

```
pli -c passInt.pli
```

共用ライブラリーのリンク: 次のコマンドを使用して、生成された PL/I オブジェクト・デックを共用ライブラリーにリンクします。

```
c89 -o libpassInt.so passInt.o
```

PL/I 共用ライブラリーの名前には必ず *lib* 接頭部を付けてください。そうしないと、Java クラス・ローダーはライブラリーを検出できません。

ステップ 5: サンプル・プログラムの実行

次のコマンドを使用して、Java - PL/I サンプル・プログラムを実行します。

```
java jPassInt

Java と PL/I の両方からのユーザー入力プロンプトを含めた、サンプル・プログラムの出力は次のとおりです。

>java jPassInt

From Java: Enter an Integer or 'quit' to quit.
Java Prompt > 12345

From PLI: Integer retrieved from Java is: 12345
From PLI: Enter an integer to be returned to Java:
54321

From Java: Integer set by PL/I is: 54321
From Java: Enter an Integer or 'quit' to quit.
Java Prompt > quit
>
```

JNI サンプル・プログラム #4 - Java 呼び出し API

Java サンプル・プログラム #4 の作成

このサンプル・プログラムは、今までのサンプルと若干異なります。このサンプルでは、まず PL/I が Java 呼び出し API を介して Java を呼び出し、組み込み Java 仮想マシン (JVM) を作成します。次に PL/I が、ストリングを渡して Java メソッドを呼び出します。そして、Java メソッドはそのストリングを表示します。

PL/I サンプル・プログラムは *createJVM.pli* という名前で、このサンプル・プログラムによって呼び出される Java メソッドは *javaPart.java* に含まれています。

ステップ 1: Java プログラムの作成

このサンプルは PL/I ネイティブ・メソッドを使用しないため、それを宣言する必要がありません。その代わり、このサンプルの Java 部分には単純な Java メソッドが 1 つ含まれています。

Java main メソッドの作成: *javaPart* クラスには、ステートメントが 1 つだけ含まれます。このステートメントは、Java から「Hello World...」という短いメッセージを出力し、PL/I プログラムから渡されたストリングを付け足します。クラス全体は、図 86 に示されています。

```
// Receive a string from PL/I then display it after saying "Hello"
public class javaPart {
    public static void main(String[] args) {
        System.out.println("From Java - Hello World... " + args[0]);
    }
}
```

図 86. Java サンプル・プログラム #4 - スtringの受け取りおよび出力

ステップ 2: Java プログラムのコンパイル

Java コードをコンパイルするコマンドは、次のようになります。

```
javac javaPart.java
```

ステップ 3: PL/I プログラムの作成

PL/I "Hello World" サンプル・プログラムの作成についての説明の大部分が、このプログラムにも当てはまります。しかし、このサンプルでは PL/I は Java を呼び出すため、考慮する必要のある点が他にもあります。

PL/I プロシージャ名とプロシージャ・ステートメントの正しい形式: このサンプルでは PL/I プログラムが Java を呼び出すため、PL/I プログラムは MAIN になります。この PL/I プログラムの外部名は参照されないため、考慮する必要はありません。

サンプル・プログラムの完全なプロシージャ・ステートメントは、次のとおりです。

```
createdJVM: Proc Options(Main);
```

JNI インクルード・ファイル: Java ネイティブ・インターフェースの PL/I 定義を含む 2 つの PL/I インクルード・ファイルは、*ibmzjni.inc* およびこれにインクルードされる *ibmzjnim.inc* です。このサンプルで PL/I が Java を呼び出すとしても、これらのインクルード・ファイルはやはり必要です。これらのインクルード・ファイルは、次のステートメントとともに組み込まれています。

```
%include ibmzjni;
```

ibmzjni および *ibmzjnim* インクルード・ファイルは、PL/I SIBMZSAM データ・セットの中に提供されています。

PL/I プログラムと Java ライブラリーとのリンク: この PL/I サンプル・プログラムは Java を呼び出すため、Java ライブラリーにリンクできるようにする必要があります。Java ライブラリーは XPLINK とリンクしていますが、PL/I モジュールはリンクしていません。PL/I は現在でも、XPLINK ライブラリーとリンクしてそれら呼び出すことができますが、PLIXOPT 変数を使用して、**XPLINK=ON** ランタイム・オプションを指定する必要があります。PLIXOPT 変数の宣言は、以下のようになります。

```
Dcl PliXOpt Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );
```

PLIXOPT の使用法については、「*z/OS 言語環境プログラム プログラミングの手引き*」を参照してください。

Java 呼び出し API の使用: この PL/I サンプル・プログラムは、Java 呼び出し API *JNI_CreateJavaVM* を呼び出して、自身の組み込み JVM を作成します。この API には、384 ページの図 87 に示されているような、セットアップおよび初期化を正しく行うための特定の構造体が必要です。まず、*JNI_GetDefaultJavaVMInitArgs* が、デフォルトの初期化オプションを取得するために呼び出されます。次に、これらのデフォルト・オプションが、*java.class.path* 情報の追加により変更されます。最後に、*JNI_CreateJavaVM* が組み込み JVM を作成するために呼び出されます。

完全な PL/I プログラム: 完全な PL/I プログラムは、384 ページの図 87 に示してあります。このサンプル PL/I プログラムは、JNI を介していくつかの呼び出しを行います。

このサンプルでは、Java オブジェクト (この場合は新しく作成された JVM) への参照は渡されませんが、代わりに *JNI_CreateJavaVM* API への呼び出しから戻されます。PL/I プログラムはこの参照を使用して、JVM から情報を取得します。情報の最初の部分は、呼び出す Java メソッドを含むクラスです。このクラスは、*FindClass* JNI 関数を使用して検出されます。クラス値が *GetStaticMethodID* JNI 関数によって使用され、呼び出される Java メソッドの ID が取得されます。

この Java メソッドを呼び出す前に、PL/I スtring を Java が認識できるフォーマットに変換する必要があります。PL/I プログラムでは、String は ASCII フォーマットで保持されます。Java String は UTF フォーマットで保管されます。さらに、Java String は、PL/I プログラマーが認識するような意味では真に String ではありませんが、それ自体 Java クラスであり、メソッドを通じてのみ変更できます。 *NewStringUTF* JNI 関数を使用することにより、Java String 配列を作成します。この関数は、UTF に変換された PL/I String を含む *myJString* という Java オブジェクトを戻します。次に、*myJString* オブジェクトへの参照を渡して、*NewObjectArray* JNI 関数を呼び出すことにより、Java オブジェクト配列を作成します。この関数は、Java メソッドに表示させる String を含む Java オブジェクト配列への参照を返します。

これで、*CallStaticVoidMethod* JNI 関数を使用して Java メソッドを呼び出すことができました。次に、Java メソッドは、渡された String を表示します。String を表示した後、PL/I プログラムは、*DestroyJavaVM* JNI 関数を使用して組み込み JVM を破棄し、PL/I プログラムが完了します。

PL/I プログラムの完全なソースは、384 ページの図 87 にあります。

```

*Process Limits( Extname( 100 ) ) Margins( 1, 100 );
*Process LangLVL( SAA2 ) Margins( 1, 100 ) ;
*Process Display(STD) Rent;
*Process Default( ASCII ) Or('|');
createJVM: Proc Options(Main);

    %include ibmzjni;

    Dcl myJObjArray Type jobjectArray;
    Dcl myClass      Type jclass;
    Dcl myMethodID   Type jmethodID;
    Dcl myJString    Type jstring;
    Dcl myRC         Fixed Bin(31) Init(0);
    Dcl myPLIStr     Char(50) Varz
                    Init('... a PLI string in a Java Virtual Machine!');

    Dcl OptStr       Char(1024) Varz;
    Dcl myNull       Pointer;
    Dcl VM_Args      Like JavaVMInitArgs;
    Dcl myOptions     Like JavaVMOption;
    Dcl PLIXOPT       Char(40) Varying Ext Static Init( 'XPLINK(ON)'e );

    Display('From PL/I - Beginning execution of createJVM...');
    /* Important difference between J1.3 and J1.4: */
    /* J1.3 requires VM_Args.version = JNI_VERSION_1_2; */
    /* J1.4 requires VM_Args.version = JNI_VERSION_1_4; */
    VM_Args.version   = JNI_VERSION_1_4;

    myRC = JNI_GetDefaultJavaVMInitArgs( addr(VM_Args) );
    OptStr = "-Djava.class.path=.";
    myOptions.theOptions = addr(OptStr);
    VM_Args.nOptions = 1;
    VM_Args.JavaVMOption = addr(myOptions);

    /* Create the Java VM */
    myrc = JNI_CreateJavaVM(
        addr(JavaVM),
        addr(JNIEnv),
        addr(VM_Args) );

    /* Get the Java Class for the javaPart class */
    myClass = FindClass(JNIEnv, "javaPart");
    /* Get static method ID */
    myMethodID = GetStaticMethodID(JNIEnv, myClass, "main",
        "([Ljava/lang/String;)V" );
    /* Create a Java String Object from the PL/I string. */
    myJString = NewStringUTF(JNIEnv, myPLIStr);
    myJObjArray = NewObjectArray(JNIEnv, 1,
        FindClass(JNIEnv, "java/lang/String"), myJString);
    Display('From PL/I - Calling Java method in new JVM from PL/I...');
    Display(' ');
    myNull = CallStaticVoidMethod(JNIEnv, myClass,
        myMethodID, myJObjArray );
    /* destroy the Java VM */
    Display(' ');
    Display('From PL/I - Destroying the new JVM from PL/I...');
    myRC = DestroyJavaVM( JavaVM ); /* rc = -1 is OK */
end;

```

図 87. PL/I サンプル・プログラム #4 - Java 呼び出し API の呼び出し

ステップ 4: PL/I プログラムのコンパイルとリンク

PL/I プログラムのコンパイル: 次のコマンドを使用して、PL/I サンプル・プログラムをコンパイルします。

```
pli -c createJVM.pli
```

PL/I プログラムの Java ライブラリーへのリンク: 次のコマンドを使用して、生成された PL/I オブジェクト・デックを Java ライブラリーにリンクします。

```
c89 -o createJVM createJVM.o $JAVA_HOME/bin/classic/libjvm.x
```

\$JAVA_HOME 環境変数への参照に注意してください。この変数は、ご使用の Java 1.4 製品がインストールされているディレクトリーを指す必要があります。例えば、ご使用の環境でこの変数をセットアップするには、次のコマンドを使用します。

```
export JAVA_HOME="/usr/lpp/java/J1.4"
```

このケースでは、Java 1.4 製品が `/usr/lpp/java/J1.4` にインストールされていることが前提となっています。

ステップ 5: サンプル・プログラムの実行

次のコマンドを使用して、PL/I - Java サンプル・プログラムを実行します。

```
createJVM
```

サンプル・プログラムの出力は次のとおりです。

```
From PL/I - Beginning execution of createJVM...
From PL/I - Calling Java method in new JVM from PL/I...
From Java - Hello World... ... a PLI string in a Java Virtual Machine!
From PL/I - Destroying the new JVM from PL/I...
```

Java および PL/I の同等なデータ・タイプの判別

PL/I から Java とのやり取りを行う際には、2 つのプログラム言語間でデータ・タイプを一致させる必要があります。次の表に、Java の基本的なタイプと PL/I の同等なタイプを示します。

表 32. Java の基本的なタイプと同等な PL/I ネイティブのタイプ

Java のタイプ	PL/I のタイプ	サイズ (ビット)
boolean	jboolean	8、符号なし
byte	jbyte	8
char	jchar	16、符号なし
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	21
double	jdouble	53
void	jvoid	n/a

第 5 部 特殊プログラミング・タスク

第 15 章 PLISAXA および PLISAXB XML パー

サーの使用	389
概要	389
PLISAXA 組み込みサブルーチン	390
PLISAXB 組み込みサブルーチン	390
SAX イベント構造体	391
start_of_document	392
version_information	392
encoding_declaration	392
standalone_declaration	392
document_type_declaration	392
end_of_document	392
start_of_element	392
attribute_name	392
attribute_characters	393
attribute_predefined_reference	393
attribute_character_reference	393
end_of_element	393
start_of_CDATA_section	393
end_of_CDATA_section	393
content_characters	394
content_predefined_reference	394
content_character_reference	394
processing_instruction	394
comment	394
unknown_attribute_reference	395
unknown_content_reference	395
start_of_prefix_mapping	395
end_of_prefix_mapping	395
exception	395
イベント関数に渡されるパラメーター	395
XML 文書のコード化文字セット	396
サポートされる EBCDIC コード・ページ	396
サポートされる ASCII コード・ページ	397
コード・ページの指定	397
番号の使用	397
別名の使用	398
例外	398
例	399
継続可能な例外コード	411
例外コードの終了	415

第 16 章 PLISAXC XML パーサーの使用

概要	419
PLISAXC 組み込みサブルーチン	420
SAX イベント構造体	420
start_of_document	421
version_information	421
encoding_declaration	421
standalone_declaration	421
document_type_declaration	421

end_of_document	422
start_of_element	422
attribute_name	422
attribute_characters	422
end_of_element	422
start_of_CDATA_section	422
end_of_CDATA_section	423
content_characters	423
processing_instruction	423
comment	423
namespace_declare	423
end_of_input	423
unresolved_reference	424
exception	424
イベント関数に渡されるパラメーター	424
イベントにおける差異	426
XML 文書のコード化文字セット	426
サポートされるコード・ページ	427
コード・ページの指定	427
番号の使用	428
別名の使用	428
例外	428
単純な文書での例	428

第 17 章 PLIDUMP の用法

PLIDUMP の使用上の注意	440
PLIDUMP 出力内の変数の検出	441
AUTOMATIC 変数の検出	441
STATIC 変数の検出	442
CONTROLLED 変数の検出	444
NORENT WRITABLE を指定した場合	444
NORENT NOWRITABLE(FWS) を指定した場 合	445
NORENT NOWRITABLE(PRV) を指定した場 合	446
保存されたコンパイル・データ	447
タイム・スタンプ	447
保存されたオプション・ストリング	448

第 18 章 割り込みとアテンションの処理

ATTENTION ON ユニットの使用	454
デバッグ・ツールとの対話	454

第 19 章 チェックポイント/再始動機能の使用

チェックポイント・レコードの要求	455
チェックポイント・データ・セットの定義	456
再始動の要求	457
システム障害後の自動再始動	457
プログラム内の自動再始動	458
据え置き再始動	458
チェックポイント/再始動活動の変更	458

第 20 章 ユーザー出口の用法	459
コンパイラー・ユーザー出口によって実行されるプ ロシージャー	459
コンパイラー・ユーザー出口の活動化	460
IBM 提供のコンパイラー出口、IBMUEXIT	460
コンパイラー・ユーザー出口のカスタマイズ	461
SYSUEXIT の変更	461
独自のコンパイラー出口の作成	461
グローバル制御ブロックの構造	462
初期化プロシージャーの作成	463
メッセージ・フィルタ操作プロシージャーの作 成	463
終了プロシージャーの作成	465
第 21 章 PL/I 記述子	467
引数の引き渡し	467
記述子リストによる引数の引き渡し	467
記述子ロケーターによる引数の引き渡し	468
CMPAT(V*) 記述子	468
ストリング記述子	468
配列記述子	470
CMPAT(LE) 記述子	470
ストリング記述子	471
配列記述子	471

第 15 章 PLISAXA および PLISAXB XML パーサーの使用

PLISAXx (x = A または B) 組み込みサブルーチンは、プログラムがインバウンド XML 文書を取り込んで、適格性を検査して、その内容を処理できるようにする、基本的な XML 構文解析機能を提供します。

これらのサブルーチンでは XML の生成は提供しておらず、代わりに、PL/I プログラム・ロジックによって行うか、または XMLCHAR 組み込み関数を使用する必要があります。

PLISAXA および PLISAXB には、特別な環境要件はありません。

CICS、IMS、MQ Series、さらに z/OS のバッチと TSO など、主要なすべてのランタイム環境で実行できます。

PLISAXA および PLISAXB には一部の重要な制限があります。XML 名前空間のサポートおよび Unicode UTF-8 文書のサポートがありません。また、XML 文書の構文解析前に、その文書全体を (バッファまたはファイルで) 渡す必要があります。PLISAXC 組み込みサブルーチンにはこれらの制限はありません。このサブルーチンは、PLISAXA および PLISAXB 組み込みサブルーチンと多くの点で類似していますが、大分異なるところもあり、この章では説明しません。PLISAXC の詳細については、次の章で説明します。

概要

XML 構文解析用のインターフェースには、大きく分けてイベント・ベースとツリー・ベースの 2 種類があります。

イベント・ベース API の場合、パーサーはコールバックによってアプリケーションにイベントを報告します。報告されるイベントは、文書の開始、エレメントの開始などです。アプリケーションは、パーサーによって報告されるイベントを処理するためのハンドラーを備えています。Simple API for XML (SAX) は、業界標準のイベント・ベース API の一例です。

ツリー・ベース API (文書オブジェクト・モデル (DOM) など) の場合、パーサーは XML をツリー・ベースの内部表現に変換します。ツリーをナビゲートするためのインターフェースが提供されています。

IBM PL/I は、XML 文書の構文解析用に SAX のようなイベント・ベースのインターフェースを提供します。パーサーは、対応する文書フラグメントへの参照を渡し、アプリケーション提供のパーサー・イベント用のハンドラーを呼び出します。

パーサーには次の特性があります。

- ・ 高性能であるが非標準のインターフェースを提供します。
- ・ ユニコード UTF-16、または後述の 1 バイト・コード・ページのいずれかでエンコードされた XML ファイルをサポートします。
- ・ パーサーは有効性検査を行いませんが、適格性を部分的に検査します。セクション 2.5.10 を参照してください。


XML 文書の準拠レベルには適格性と有効性の 2 つがあり、どちらのレベルも XML 標準に定義されています。XML 標準は、<http://www.w3c.org/XML/> に掲載されています。これらの定義を要約すると、XML 文書が基本的な XML 文法と、いくつかの特定の規則（開始エレメントと終了エレメントのタグが一致していることなどの要件）に準拠していれば、XML 文書は整形形式です。さらに、整形形式 XML 文書に文書タイプ宣言 (DTD) が関連していて、文書が DTD に表された制約に準拠している場合、その文書は有効です。

XML パーサーは有効性検査を行いませんが、適格性のエラーを部分的に検査し、エラーを発見した場合は例外イベントを生成します。

それぞれのパーサー・イベントごとに、下記のコード例に示すように、適切なパラメーターを受け入れて適切な戻り値を戻す PL/I 関数を用意する必要があります。特に、戻り値は BYVALUE で戻す必要があることに注意してください。また、これらの関数は、すべて OPTLINK リンケージを使用する必要があります。DEFAULT(LINKAGE(OPTLINK)) オプションを使用してこのリンケージを指定するか、または OPTIONS(LINKAGE(OPTLINK)) 属性を使用して、個々の PROCEDURE および ENTRY でこのリンケージを指定することができます。

PLISAXA 組み込みサブルーチン

PLISAXA 組み込みサブルーチンを使用すると、プログラムのバッファ内にある XML 文書に対して XML パーサーを起動することができます。

▶▶—PLISAXA(*e,p,x,n*—)—▶▶

- e** イベント構造体
- p** パーサーがイベント関数に戻すポインター値または「トークン」
- x** 入力 XML が入っているバッファのアドレス
- n** そのバッファにあるデータのバイト数
- c** その XML のコード・ページの名称を指定する数値表現

XML が CHARACTER VARYING スtringまたは WIDECHAR VARYING スtringに含まれている場合は、ADDRDATA 組み込み関数を使用して、最初のデータ・バイトのアドレスを取得する必要があります。

また、XML が WIDECHAR スtringに含まれている場合、バイト数の値は LENGTH 組み込み関数によって戻される値の 2 倍になることに注意してください。

PLISAXB 組み込みサブルーチン

PLISAXB 組み込みサブルーチンを使用すると、ファイル内にある XML 文書に対して XML パーサーを起動することができます。

▶▶—PLISAXB(*e,p,x*—)—▶▶

- e イベント構造体
- p パーサーがイベント関数に戻すポインター値または「トークン」
- x 入力ファイルを指定する文字ストリング式
- c その XML のコード・ページの名称を指定する数値表現

バッチのもとでは、入力ファイルを指定する文字ストリングは 'file://dd:ddname' のフォームであり、ddname は、そのファイルを指定している DD ステートメントの名前です。

z/OS UNIX のもとでは、入力ファイルを指定する文字ストリングは、'file://filename' の書式になっている必要があります。filename は、z/OS UNIX ファイルの名前です。

バッチと z/OS UNIX の両方の環境とも、入力ファイルを指定する文字ストリングには、先行ブランクも末尾ブランクも含めません。

入力 XML ファイルのサイズは、2G 未満でなくてはなりません。また、パーサーはファイル全体をメモリーに読み取るため、ご使用のプログラムの REGION は、文書すべてを含めることができるストレージの部分をパーサーが取得できるほどに大きくなければなりません。

SAX イベント構造体

イベント構造体は、24 個の LIMITED ENTRY 変数で構成される構造体です。これらの変数は、さまざまな「イベント」に対してパーサーが起動する機能を指しています。

これらすべての ENTRY は、OPTLINK リンケージを使用する必要があります。

次に示す各イベントの説明は、図 88 の XML 文書の例に対応しています。この説明にある「XML テキスト」という用語は、イベントに渡されるポインターと長さに基づくストリングを意味しています。

```
xmlDocument =
  '<?xml version="1.0" standalone="yes"?>'
  '<!--This document is just an example-->'
  '<sandwich>'
  '<bread type="baker&quot;s best"/>'
  '<?spread please use real mayonnaise ?>'
  '<meat>Ham & turkey</meat>'
  '<filling>Cheese, lettuce, tomato, etc.</filling>'
  '<![CDATA[We should add a <relish> element in future!]]>'
  '</sandwich>'
  'junk';
```

図 88. サンプル XML 文書

この構造体での出現順に、パーサーは次のイベントを認識することができます。

start_of_document

このイベントは、文書の構文解析が開始されるときに 1 回発生します。パーサーは、LF (改行) や NL (改行) などの行制御文字を含む、文書全体のアドレスと長さを渡します。上記の例では、文書の長さは 305 文字です。

version_information

このイベントは、オプショナルの XML 宣言内のバージョン情報に対して発生します。パーサーは、バージョン値 (上記の例では "1.0") が入ったテキストのアドレスと長さを渡します。

encoding_declaration

このイベントは、XML 宣言内でオプショナルのエンコード宣言に対して発生します。パーサーは、エンコード方式値が入ったテキストのアドレスと長さを渡します。

standalone_declaration

このイベントは、XML 宣言内でオプショナルのスタンドアロン宣言に対して発生します。パーサーは、スタンドアロン値 (上記の例では "yes") が入ったテキストのアドレスと長さを渡します。

document_type_declaration

このイベントは、パーサーが文書タイプ宣言を検出したときに発生します。文書タイプ宣言は、文字シーケンス "<!DOCTYPE" から始まって ">" 文字で終わるもので、その間には内容を記述するやや複雑な文法規則が入ります。パーサーは、開始と終了の文字シーケンスを含む宣言全体が入ったテキストのアドレスと長さを渡します。このイベントは、XML テキストに区切り文字が含まれる唯一のイベントです。上記の例には、文書タイプ宣言はありません。

end_of_document

このイベントは、文書の構文解析が完了したときに 1 回発生します。

start_of_element

このイベントは、エレメント開始タグ、または空エレメント・タグごとに 1 回発生します。パーサーは、エレメント名が入ったテキストのアドレスと長さを渡します。例の構文解析中に最初に発生する start_of_element イベントの場合、このテキストはストリング "sandwich" です。

attribute_name

このイベントは、エレメント開始タグ、または空エレメント・タグ内の属性ごとに、有効な名前を認識した後に発生します。パーサーは、属性名が入ったテキストのアドレスと長さを渡します。例にある属性名は "type" だけです。

attribute_characters

このイベントは、属性値のフラグメントごとに発生します。パーサーは、フラグメントが入ったテキストのアドレスと長さを渡します。属性値は通常、次のように複数の行に分割されている場合でもただ 1 つのストリングで構成されます。

```
<element attribute="This attribute value is  
split across two lines"/>
```

ただし、属性値は複数の部分で構成されている場合があります。例えば、"sandwich" の例でセクションの最初にある "type" 属性の値は、ストリング "baker"、単一文字 "'", およびストリング "s best" の 3 つのフラグメントで構成されています。パーサーは、これらのフラグメントを 3 つの別々のイベントとして渡します。ストリング (例の "baker" と "s best") はそれぞれ attribute_characters イベントとして渡され、単一文字 "'" は次に説明する attribute_predefined_reference イベントとして渡されます。

attribute_predefined_reference

このイベントは、属性値の中で 5 つの定義済みエンティティー参照 "&"、"'", ">", "<", および "'" に対して発生します。パーサーは、"&"、"'", ">", "<", または "'" のうちの 1 つがそれぞれ入った CHAR(1) または WIDECHAR(1) の値を渡します。

attribute_character_reference

このイベントは、属性値の中で "&#dd;" または "&#xhh;" の形式の数字参照 (ユニコード・コード・ポイント、または「スカラー値」) に対して発生します。ただし、"d" と "h" はそれぞれ 10 進数字と 16 進数字を表します。パーサーは、対応する整数値が入った FIXED BIN(31) 値を渡します。

end_of_element

このイベントは、エレメント終了タグ、または空エレメント・タグごとに、パーサーがタグの終了不等号括弧を認識したときに 1 回発生します。パーサーは、エレメント名が入ったテキストのアドレスと長さを渡します。

start_of_CDATA_section

このイベントは、CDATA セクションが開始されると発生します。CDATA セクションはストリング "<![CDATA[" で始まり、ストリング "]" で終わるもので、他の場合には XML マークアップとして認識される文字を含むテキストのブロックを「エスケープ」するために使用されます。パーサーは、開始文字 "<![CDATA[" が入ったテキストのアドレスと長さを渡します。パーサーは、これらの区切り文字間にある CDATA セクションの内容を単一の content-characters イベントとして渡します。例えば上記の例では、content-characters イベントとしてテキスト "We should add a <relish> element in future!" が渡されます。

end_of_CDATA_section

このイベントは、パーサーが CDATA セクションの終了を認識したときに発生します。パーサーは、終了文字シーケンス "]" が入ったテキストのアドレスと長さを渡します。

content_characters

このイベントは、XML 文書の「本体」である、エレメントの開始タグと終了タグの間にある文字データを表します。パーサーは、このデータが入ったテキストのアドレスと長さを渡します。テキストは通常、次のように複数の行に分割されている場合でもただ 1 つのストリングで構成されます。

```
<element1>This character content is  
split across two lines</element1>
```

エレメント内容に参照や他のエレメントが含まれている場合、内容全体が複数のセグメントで構成されることがあります。例えば、例の "meat" エレメントの内容は、ストリング "Ham "、文字 "&"、およびストリング " turkey" で構成されます。これら 2 つのストリング・フラグメントのそれぞれ先頭と末尾にあるスペースに注意してください。パーサーは、これら 3 つの内容フラグメントを別々のイベントとして渡します。ストリングの内容フラグメント ("Ham " と " turkey") は content_characters イベントとして渡され、単一の "&" 文字は content_predefined_reference イベントとして渡されます。またパーサーは、content_characters イベントを使用して CDATA セクションのテキストをアプリケーションに渡します。

content_predefined_reference

このイベントは、エレメント内容にある 5 つの定義済みエンティティー参照 "&"、"'"、">"、"<"、および '"' に対して発生します。パーサーは、"&"、"'"、">"、"<"、または '"' のうちの 1 つがそれぞれ入った CHAR(1) または WIDECHAR(1) の値を渡します。

content_character_reference

このイベントは、エレメント内容にある "&#dd;" または "&#xhh;" の形式の数字参照 (ユニコード・コード・ポイント、または「スカラー値」) に対して発生します。ただし、"d" と "h" はそれぞれ 10 進数字と 16 進数字を表します。パーサーは、対応する整数値が入った FIXED BIN(31) 値を渡します。

processing_instruction

処理命令 (PI) を使用すると、XML 文書にアプリケーション用の特別な命令を含めることができます。このイベントは、パーサーが PI 開始文字シーケンス "<?" に続く名前を認識したときに発生します。さらにこのイベントは、処理命令 (PI) ターゲットに続く、PI 終了文字シーケンス ">" の直前までのデータを対象とします。データの末尾にある空白文字は含まれますが、先頭にある空白文字は含まれません。パーサーは、ターゲットが入ったテキスト (例では "spread") のアドレスと長さ、およびデータが入ったテキストのアドレスと長さ (例では "please use real mayonnaise ") を渡します。

comment

このイベントは、XML 文書内のコメントに対して発生します。パーサーは、開始および終了コメント区切り文字 (それぞれ "<!--" と "-->") の間にあるテキストのアドレスと長さを渡します。例では、唯一のコメントのテキストは "This document is just an example" です。

unknown_attribute_reference

このイベントは、属性値の中で、5 つの定義済みエンティティ参照 (イベント `attribute_predefined_character` の項に示した) 以外のエンティティ参照に対して発生します。パーサーは、エンティティ名が入ったテキストのアドレスと長さを渡します。

unknown_content_reference

このイベントは、エレメント内容の中で、5 つの定義済みエンティティ参照 (`content_predefined_character` イベントの項に示した) 以外のエンティティ参照に対して発生します。パーサーは、エンティティ名が入ったテキストのアドレスと長さを渡します。

start_of_prefix_mapping

このイベントは、現在は生成されません。

end_of_prefix_mapping

このイベントは、現在は生成されません。

exception

XML 文書の処理中にエラーを検出すると、パーサーはこのイベントを生成します。

イベント関数に渡されるパラメーター

イベント関数はすべて、パーサーへの戻りコードである `BYVALUE FIXED BIN(31)` 値を返す必要があります。パーサーを正常に継続するためには、この値がゼロでなければなりません。

これらの関数すべてに、最初の引数として `BYVALUE POINTER` が渡されます。この値は、元は組み込み関数への 2 番目の引数として渡されたトークン値です。

次に示す例外を除き、イベントのテキスト・エレメントのアドレスと長さを提供する `BYVALUE POINTER` と `BYVALUE FIXED BIN(31)` も、すべての関数に渡されます。例外の関数/イベントは、次のとおりです。

end_of_document

ユーザー・トークン以外の引数は渡されません。

attribute_predefined_reference

ユーザー・トークンに加えて、定義済み文字の値を保持する `BYVALUE CHAR(1)`、または (UTF-16 文書の場合) `BYVALUE WIDECHAR(1)` がもう 1 つの引数として渡されます。

content_predefined_reference

ユーザー・トークンに加えて、定義済み文字の値を保持する `BYVALUE CHAR(1)`、または (UTF-16 文書の場合) `BYVALUE WIDECHAR(1)` がもう 1 つの引数として渡されます。

attribute_character_reference

ユーザー・トークンに加えて、数値参照の値を保持する `BYVALUE FIXED BIN(31)` がもう 1 つの引数として渡されます。

content_character_reference

ユーザー・トークンに加えて、数値参照の値を保持する BYVALUE FIXED BIN(31) がもう 1 つの引数として渡されます。

processing_instruction

ユーザー・トークンに加えて、次の 4 つの引数が渡されます。

1. ターゲット・テキストのアドレスを示す BYVALUE POINTER
2. ターゲット・テキストの長さを示す BYVALUE FIXED BIN(31)
3. データ・テキストのアドレスを示す BYVALUE POINTER
4. データ・テキストの長さを示す BYVALUE FIXED BIN(31)

exception

ユーザー・トークンに加えて、次の 3 つの引数が渡されます。

1. 問題のテキストのアドレスを示す BYVALUE POINTER
2. 文書内での問題のテキストのバイト・オフセットを示す BYVALUE FIXED BIN(31)
3. 例外コードの値を示す BYVALUE FIXED BIN(31)

XML 文書のコード化文字セット

PLISAX 組み込みサブルーチンがサポートする XML 文書は、ユニコード UTF-16 を使用してエンコードされた WIDECHAR か、または後述の明示的にサポートされる 1 バイト文字セットを使用してエンコードされた CHARACTER の文書だけです。パーサーは、XML 文書のエンコード方式に関する情報ソースを 3 つまで使用し、これらのソース間で矛盾を検出した場合は、次のように例外 XML イベントをシグナル通知します。

1. パーサーは、文書の最初の文字を検査することによって文書の基本エンコードを判別します。
2. ステップ 1 が正常に完了した場合、パーサーはエンコード宣言を検索します。
3. 最後に、パーサーは PLISAX 組み込みサブルーチン呼び出しのコード・ページ値を参照します。このパラメーターが省略された場合、デフォルトで 사용되는値は、明示指定またはデフォルトの CODEPAGE コンパイラー・オプションの値です。

XML 文書の最初に、後述のサポート対象のコード・ページを指定した XML 宣言がある場合は、その宣言が基本文書エンコード、または PLISAX 組み込みサブルーチンからのエンコード情報と矛盾しなければ、パーサーはエンコード宣言を受け入れます。XML 文書に XML 宣言自体がない場合、または XML 宣言がエンコード宣言を省略している場合は、基本文書エンコードと矛盾しなければ、パーサーは PLISAX 組み込みサブルーチンからのエンコード情報を使用して文書进行处理します。

サポートされる EBCDIC コード・ページ

次の表で、最初の番号はユーロ国別拡張コード・ページ (ECECP)、2 番目の番号は国別拡張コード・ページ (CECP) のものです。

CCSID	説明
01047	Latin 1/オープン・システム
01140、00037	米国、カナダなど
01141、00273	オーストリア、ドイツ
01142、00277	デンマーク、ノルウェー
01143、00278	フィンランド、スウェーデン
01144、00280	イタリア
01145、00284	スペイン、ラテンアメリカ (スペイン語)
01146、00285	英国
01147、00297	フランス
01148、00500	国際
01149、00871	アイスランド

サポートされる ASCII コード・ページ

CCSID	説明
00813	ISO 8859-7 ギリシャ語/ラテン語
00819	ISO 8859-1 Latin 1/オープン・システム
00920	ISO 8859-9 Latin 5 (ECMA-128、トルコ TS-5881)

コード・ページの指定

文書の XML 宣言にエンコード宣言がない場合、または XML 宣言自体がない場合は、パーサーは、PLISAX 組み込みサブルーチン呼び出しで提供されたエンコード情報を、文書の基本エンコードと組み合わせて使用します。

ほとんどの XML 文書の最初にある XML 宣言の中で、文書のエンコード情報を指定することもできます。エンコード宣言を含む XML 宣言の一例を次に示します。

```
<?xml version="1.0" encoding="ibm-1140"?>
```

XML 文書にエンコード宣言がある場合は、PLISAX 組み込みサブルーチンから提供されるエンコード情報、および文書の基本エンコードと、エンコード宣言が整合していることを確認してください。エンコード宣言、PLISAX 組み込みサブルーチンから提供されるエンコード情報、および文書の基本エンコードの間に矛盾がある場合、パーサーは例外 XML イベントをシグナル通知します。

エンコード宣言は、次のように宣言します。

番号の使用

次のいずれかの接頭部を付けて (大文字と小文字の組み合わせは自由)、CCSID 番号を指定できます (先行ゼロなし、または任意数の先行ゼロを付けて)。

IBM_	CP	CCSID_
IBM-	CP_	CCSID-
	CP-	

別名の使用

次に示す別名がサポートされており、任意に使用できます (大文字と小文字の組み合わせは自由)。

コード・ページ	サポートされる別名
037	EBCDIC-CP-US、EBCDIC-CP-CA、EBCDIC-CP-WT、 EBCDIC-CP-NL
500	EBCDIC-CP-BE、EBCDIC-CP-CH
813	ISO-8859-7、ISO_8859-7
819	ISO-8859-1、ISO_8859-1
920	ISO-8859-9、ISO_8859-9
1200	UTF-16

例外

ほとんどの例外の場合、XML テキストには、文書の、例外が検出されたポイントまで (そのポイントを含む) の構文解析済み部分が入ります。構文解析の開始前にシグナル通知される、エンコード方式の競合に関する例外の場合は、XML テキストの長さはゼロか、または文書からのエンコード宣言値だけが XML テキストに入ります。前述の例では、例外イベントを引き起こす項目が 1 つあります。"sandwich" エlement 終了タグの後ろにある余分な "junk" がその項目です。

例外には次の 2 種類があります。

1. 構文解析を任意で継続できる例外。継続可能な例外の例外コードは、1 から 99 まで、100,001 から 165,535 まで、または 200,001 から 265,535 までの範囲です。前述の例にある例外イベントの例外番号は 1 であり、したがって継続可能です。
2. 継続が不可能でない致命的な例外。致命的な例外の例外コードは、99 より大きい (ただし 100,000 より小さい) 値です。

非ゼロの戻りコードを出した例外イベント関数から戻ると、通常パーサーは文書の処理を停止し、PLISAXA または PLISAXB 組み込みサブルーチンを呼び出したプログラムに制御を戻します。

継続可能な例外の場合は、ゼロの戻りコードを指定して例外イベント関数から戻ることにより、パーサーに文書の処理継続を要求します。ただし、その後でさらに例外が発生する場合があります。継続を要求したときにパーサーがとる処置の詳細については、セクション 2.5.6.1、「継続可能な例外」を参照してください。

範囲 100,001 から 165,535 まで、および 200,001 から 265,535 までの例外番号の例外については、特殊なケースが適用されます。これらの範囲の例外コードは、文書の CCSID (エンコード宣言など、文書の先頭を検査することによって決定される) が、PLISAXA または PLISAXB 組み込みサブルーチンによって指定 (明示的または暗黙的に) された CCSID 値と同一でないことを示しています。このことは、両方の CCSID が同じ基本エンコード (EBCDIC または ASCII) を示すものであっても起こります。

これらの例外の場合、例外イベントに渡される例外コードは、EBCDIC CCSID の場合は文書の CCSID に 100,000 を加算した値、ASCII CCSID の場合は 200,000 を加算した値になります。例えば、例外コードが 101,140 の場合、文書の CCSID は 01140 です。PLISAXA または PLISAXB 組み込みサブルーチンによって提供される CCSID 値は、呼び出しの最後の引数として明示的に設定されるか、また最後の引数が省略された場合は、CODEPAGE コンパイラー・オプションの値を使用して暗黙設定されます。

このような CCSID の矛盾によって起こった例外に対する例外イベント関数から戻った後、戻りコードの値に応じて、パーサーは次の 3 つのうちいずれかの処置を実行します。

1. 戻りコードがゼロの場合、パーサーは組み込みサブルーチンによって提供された CCSID を使用して処理を続行します。
2. 戻りコードに文書の CCSID (つまり、元の例外コード値から 100,000 または 200,000 を引いた値) が入っている場合、パーサーは文書の CCSID を使用して処理を続行します。これは、構文解析イベントのいずれかから非ゼロの値が戻された後、パーサーが処理を継続する唯一のケースです。
3. そうでなければ、パーサーは文書の処理を停止し、制御を PLISAXA または PLISAXB 組み込みサブルーチンに戻します。サブルーチンは ERROR 条件を発生させます。

例

次の例は、PLISAXA 組み込みサブルーチンの使用を示すもので、前述の XML の文書を使用しています。

```
saxtest: package exports(saxtest);

define alias event
  limited entry( pointer, pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_end_of_document
  limited entry( pointer )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_predefined_ref
  limited entry( pointer, char(1) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) nodestructor );

define alias event_character_ref
  limited entry( pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_pi
  limited entry( pointer, pointer, fixed bin(31),
                pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_exception
  limited entry( pointer, pointer, fixed bin(31),
                fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );
```

図 89. PLISAXA のコーディング例 - 型宣言

```

saxtest: proc options( main );

dc1
  1 eventHandler static

    ,2 e01 type event
      init( start_of_document )
    ,2 e02 type event
      init( version_information )
    ,2 e03 type event
      init( encoding_declaration )
    ,2 e04 type event
      init( standalone_declaration )
    ,2 e05 type event
      init( document_type_declaration )
    ,2 e06 type event_end_of_document
      init( end_of_document )
    ,2 e07 type event
      init( start_of_element )
    ,2 e08 type event
      init( attribute_name )
    ,2 e09 type event
      init( attribute_characters )
    ,2 e10 type event_predefined_ref
      init( attribute_predefined_reference )
    ,2 e11 type event_character_ref
      init( attribute_character_reference )
    ,2 e12 type event
      init( end_of_element )
    ,2 e13 type event
      init( start_of_CDATA )
    ,2 e14 type event
      init( end_of_CDATA )
    ,2 e15 type event
      init( content_characters )
    ,2 e16 type event_predefined_ref
      init( content_predefined_reference )
    ,2 e17 type event_character_ref
      init( content_character_reference )
    ,2 e18 type event_pi
      init( processing_instruction )
    ,2 e19 type event
      init( comment )
    ,2 e20 type event
      init( unknown_attribute_reference )
    ,2 e21 type event
      init( unknown_content_reference )
    ,2 e22 type event
      init( start_of_prefix_mapping )
    ,2 e23 type event
      init( end_of_prefix_mapping )
    ,2 e24 type event_exception
      init( exception )
  ;

```

図 90. PLISAXA のコーディング例 - イベント構造体

```

dc1 token      char(8);

dc1 xmlDocument char(4000) var;

xmlDocument =
  '<?xml version="1.0" standalone="yes"?>'
  '<!--This document is just an example-->'
  '<sandwich>'
  '<bread type="baker&quot;s best"/>'
  '<?spread please use real mayonnaise ?>'
  '<meat>Ham & turkey</meat>'
  '<filling>Cheese, lettuce, tomato, etc.</filling>'
  '<![CDATA[We should add a <relish> element in future!]]>'.
  '</sandwich>'
  'junk';

call plisaxa( eventHandler,
              addr(token),
              addrrdata(xmlDocument),
              length(xmlDocument) );

end;

```

図 91. PLISAXA のコーディング例 - メインルーチン

```

dcl chars char(32000) based;

start_of_document:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' length=' || tokenlength );

return(0);
end;

version_information:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

return(0);
end;

encoding_declaration:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

return(0);
end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (1/8)

```

standalone_declaration:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dc1 userToken      pointer;
    dc1 xmlToken        pointer;
    dc1 tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

document_type_declaration:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dc1 userToken      pointer;
    dc1 xmlToken        pointer;
    dc1 tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

end_of_document:
  proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dc1 userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (2/8)

```

start_of_element:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken        pointer;
    dcl tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

attribute_name:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken        pointer;
    dcl tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

attribute_characters:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken        pointer;
    dcl tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (3/8)

```

attribute_predefined_reference:
  proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) nodestructor );

    dcl userToken      pointer;
    dcl reference      char(1);

    put skip list( lowercase( procname() )
      || ' ' || hex(reference) );

    return(0);
  end;

attribute_character_reference:
  proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl reference      fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || hex(reference) );

    return(0);
  end;

end_of_element:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (4/8)

```

start_of_CDATA:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

return(0);
end;

end_of_CDATA:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

return(0);
end;

content_characters:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

return(0);
end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (5/8)

```

content_predefined_reference:
  proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) nodestructor );

    dcl userToken      pointer;
    dcl reference      char(1);

    put skip list( lowercase( procname() )
      || ' ' || hex(reference) );

    return(0);
  end;

content_character_reference:
  proc( userToken, reference )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl reference      fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || hex(reference) );

    return(0);
  end;

processing_instruction:
  proc( userToken, piTarget, piTargetLength,
        piData, piDataLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl piTarget        pointer;
    dcl piTargetLength  fixed bin(31);
    dcl piData          pointer;
    dcl piDataLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(piTarget->chars,1,piTargetLength) || ' >' );

    return(0);
  end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (6/8)

```

comment:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

unknown_attribute_reference:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

unknown_content_reference:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (7/8)

```

start_of_prefix_mapping:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken        pointer;
    dcl tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

end_of_prefix_mapping:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken        pointer;
    dcl tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

exception:
  proc( userToken, xmlToken, currentOffset, errorID )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken        pointer;
    dcl currentOffset   fixed bin(31);
    dcl errorID         fixed bin(31);

    put skip list( lowercase( procname() )
      || ' errorid =' || errorid );

    return(0);
  end;
end;

```

図 92. PLISAXA のコーディング例 - イベント・ルーチン (8/8)

前のプログラムから生成される出力は、次のとおりです。

```

start_of_document length=          305
version_information <1.0>
standalone_declaration <yes>
comment <This document is just an example>
start_of_element <sandwich>
start_of_element <bread>
attribute_name <type>
attribute_characters <baker>
attribute_predefined_reference 7D
attribute_characters <s best>
end_of_element <bread>
processing_instruction <spread>
start_of_element <meat>
content_characters <Ham >
content_predefined_reference 50
content_characters < turkey>
end_of_element <meat>
start_of_element <filling>
content_characters <Cheese, lettuce, tomato, etc.>
end_of_element <filling>
start_of_cdata <<![CDATA[>
content_characters <We should add a <relish> element in future!>
end_of_cdata <]]>
end_of_element <sandwich>
exception errorid =          1
content_characters <j>
exception errorid =          1
content_characters <u>
exception errorid =          1
content_characters <n>
exception errorid =          1
content_characters <k>
end_of_document

```

図 93. PLISAXA のコーディング例 - プログラム出力

継続可能な例外コード

次の表では、例外イベント（「番号」という見出しの下にリストされている）に渡される例外コード・パラメーターの値ごとに、例外の説明と、例外の発生後にユーザーが継続を要求した場合にパーサーが行う処置を示します。この説明にある「XML テキスト」という用語は、イベントに渡されるポインターと長さに基づくストリングを意味しています。

表 33. 継続可能な例外

番号	説明	継続時のパーサーの処置
1	エレメント内容の外側で、空白文字をスキャン中にパーサーが無効文字を検出した。	パーサーは <code>content_characters</code> イベントを生成し、XML テキストには (単一の) 無効文字が入る。構文解析は無効文字の後の文字から継続する。
2	エレメント内容の外側で、処理命令、エレメント、コメント、または文書タイプ宣言の無効な開始をパーサーが検出した。	パーサーは <code>content_characters</code> イベントを生成し、XML テキストには 2 から 3 文字までの無効な先頭文字シーケンスが入る。構文解析は無効シーケンスの後の文字から継続する。

表 33. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
3	属性名の重複をパーサーが検出した。	パーサーは <code>attribute_name</code> イベントを生成し、XML テキストには重複した属性名が入る。
4	属性値の中にマークアップ文字 "<" をパーサーが検出した。	例外イベントを生成する前に、"<" 文字の前にある属性値の部分に対して、パーサーは <code>attribute_characters</code> イベントを生成する。例外イベントの後、パーサーは <code>attribute_characters</code> イベントを生成し、XML テキストには "<" が入る。構文解析は "<" の後の文字から継続する。
5	エレメントの開始タグと終了タグの名前が一致しない。	パーサーは <code>end_of_element</code> イベントを生成し、XML テキストには一致しなかった終了名が入る。
6	エレメント内容の中で無効文字をパーサーが検出した。	パーサーは、後続の <code>content_characters</code> イベントの XML テキストに無効文字を入れる。
7	エレメント内容の中で、エレメント、コメント、処理命令、または CDATA セクションの無効な開始をパーサーが検出した。	例外イベントを生成する前に、"<" マークアップ文字の前にある内容の部分に対して、パーサーは <code>content_characters</code> イベントを生成する。例外イベントの後、パーサーは <code>content_characters</code> イベントを生成し、XML テキストには "<" と無効文字の 2 つの文字が入る。構文解析は無効文字の後の文字から継続する。
8	エレメント内容の中で、一致する開始文字シーケンス "<![CDATA[" がない CDATA 終了文字シーケンス "]" をパーサーが検出した。	例外イベントを生成する前に、"]]" 文字シーケンスの前にある内容の部分に対して、パーサーは <code>content_characters</code> イベントを生成する。例外イベントの後、パーサーは <code>content_characters</code> イベントを生成し、XML テキストには 3 文字のシーケンス "]" が入る。構文解析はこのシーケンスの後の文字から継続する。
9	コメントの中で無効文字をパーサーが検出した。	パーサーは、後続の <code>comment</code> イベントの XML テキストに無効文字を入れる。
10	コメントの中で、文字シーケンス "--" の後に ">" が付いていないことをパーサーが検出した。	パーサーは "--" 文字シーケンスによってコメントが終了したと想定し、 <code>comment</code> イベントを生成する。構文解析は "--" シーケンスの後の文字から継続する。
11	処理命令データ・セグメントの中で、無効文字をパーサーが検出した。	パーサーは、後続の <code>processing_instruction</code> イベントの XML テキストに無効文字を入れる。
12	処理命令ターゲット名が、小文字、大文字、または大/小文字混合の "xml" である。	パーサーは <code>processing_instruction</code> イベントを生成し、XML テキストには元の大文字小文字を使用した "xml" が入る。
13	16 進文字参照 (形式 ෝ) の中で、無効数字をパーサーが検出した。	パーサーは <code>attribute_characters</code> イベントまたは <code>content_characters</code> イベントを生成し、XML テキストには無効数字が入る。参照の構文解析は、この無効数字の後から継続する。
14	10 進文字参照 (形式 &#ddd;) の中で、無効数字をパーサーが検出した。	パーサーは <code>attribute_characters</code> イベントまたは <code>content_characters</code> イベントを生成し、XML テキストには無効数字が入る。参照の構文解析は、この無効数字の後から継続する。

表 33. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
15	XML 宣言のエンコード宣言値が、小文字または大文字の A から Z から始まっていない。	パーサーは <code>encoding</code> イベントを生成し、XML テキストには指定されたとおりのエンコード宣言値が入る。
16	文字参照が正しい XML 文字を参照していない。	パーサーは <code>attribute_character_reference</code> イベントまたは <code>content_character_reference</code> イベントを生成し、XML-NTEXT には文字参照に指定された単一のユニコード文字が入る。
17	エンティティー参照名の中で、無効文字をパーサーが検出した。	パーサーは、後続の <code>unknown_attribute_reference</code> イベント、または <code>unknown_content_reference</code> イベントの XML テキストに無効文字を入れる。
18	属性値の中で、無効文字をパーサーが検出した。	パーサーは、後続の <code>attribute_characters</code> イベントの XML テキストに無効文字を入れる。注: PL/I XML パーサーは、標準から逸脱し、「<」を属性ストリングにおいて有効であると受け入れます。
50	文書は EBCDIC でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されているが、文書のエンコード宣言に認識可能なエンコード方式が指定されていない。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
51	文書は EBCDIC でエンコードされ、文書のエンコード宣言にはサポートされる EBCDIC エンコード方式が指定されているが、CODEPAGE コンパイラー・オプションに指定されたコード・ページをパーサーがサポートしていない。	パーサーは、文書のエンコード宣言に指定されたエンコード方式を使用する。
52	文書は EBCDIC でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されているが、文書のエンコード宣言には ASCII エンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
53	文書は EBCDIC でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されているが、文書のエンコード宣言にはサポートされるユニコード・エンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
54	文書は EBCDIC でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されているが、文書のエンコード宣言には、パーサーがサポートしないユニコード・エンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。

表 33. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
55	文書は EBCDIC でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されているが、文書のエンコード宣言には、パーサーがサポートしないエンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
56	文書は ASCII でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されているが、文書のエンコード宣言に認識可能なエンコード方式が指定されていない。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
57	文書は ASCII でエンコードされ、文書のエンコード宣言にはサポートされる ASCII エンコード方式が指定されているが、CODEPAGE コンパイラー・オプションに指定されたコード・ページをパーサーがサポートしていない。	パーサーは、文書のエンコード宣言に指定されたエンコード方式を使用する。
58	文書は ASCII でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されているが、文書のエンコード宣言にはサポートされる EBCDIC エンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
59	文書は ASCII でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されているが、文書のエンコード宣言にはサポートされるユニコード・エンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
60	文書は ASCII でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されているが、文書のエンコード宣言には、パーサーがサポートしないユニコード・エンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
61	文書は ASCII でエンコードされ、CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されているが、文書のエンコード宣言には、パーサーがサポートしないエンコード方式が指定されている。	パーサーは、CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。
100,001 から 165,535	文書は EBCDIC でエンコードされ、CODEPAGE コンパイラー・オプションと文書のエンコード宣言に指定されたエンコード方式は、両方ともサポートされる EBCDIC コード・ページだが、一致していない。例外コードには、エンコード宣言の CCSID に 100,000 を加算した値が入る。	ユーザーが例外イベントからゼロを戻した場合、パーサーは CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。文書のエンコード宣言からの CCSID を戻した場合 (例外コードから 100,000 を減算して)、パーサーはこのエンコード方式を使用する。

表 33. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
200,001 から 265,535	文書は ASCII でエンコードされ、CODEPAGE コンパイラー・オプションと文書のエンコード宣言に指定されたエンコード方式は、両方ともサポートされる ASCII コード・ページだが、一致していない。例外コードには、エンコード宣言の CCSID に 200,000 を加算した値が入る。	ユーザーが例外イベントからゼロを戻した場合、パーサーは CODEPAGE コンパイラー・オプションに指定されたエンコード方式を使用する。文書のエンコード宣言からの CCSID を戻した場合 (例外コードから 200,000 を減算して)、パーサーはこのエンコード方式を使用する。

例外コードの終了

表 34. 終了例外

番号	説明
100	XML 宣言の開始のスキャン中に、パーサーが文書の終わりに達した。
101	XML 宣言の終了の検索中に、パーサーが文書の終わりに達した。
102	ルート・エレメントの検索中に、パーサーが文書の終わりに達した。
103	XML 宣言のバージョン情報の検索中に、パーサーが文書の終わりに達した。
104	XML 宣言のバージョン情報値の検索中に、パーサーが文書の終わりに達した。
106	XML 宣言のエンコード宣言値の検索中に、パーサーが文書の終わりに達した。
108	XML 宣言のスタンドアロン宣言値の検索中に、パーサーが文書の終わりに達した。
109	属性名のスキャン中に、パーサーが文書の終わりに達した。
110	属性値のスキャン中に、パーサーが文書の終わりに達した。
111	属性値の文字参照またはエンティティー参照のスキャン中に、パーサーが文書の終わりに達した。
112	空エレメント・タグのスキャン中に、パーサーが文書の終わりに達した。
113	ルート・エレメント名のスキャン中に、パーサーが文書の終わりに達した。
114	エレメント名のスキャン中に、パーサーが文書の終わりに達した。
115	エレメント内容の文字データのスキャン中に、パーサーが文書の終わりに達した。
116	エレメント内容の処理命令のスキャン中に、パーサーが文書の終わりに達した。
117	エレメント内容のコメントまたは CDATA セクションのスキャン中に、パーサーが文書の終わりに達した。
118	エレメント内容のコメントのスキャン中に、パーサーが文書の終わりに達した。
119	エレメント内容の CDATA セクションのスキャン中に、パーサーが文書の終わりに達した。
120	エレメント内容の文字参照またはエンティティー参照のスキャン中に、パーサーが文書の終わりに達した。
121	ルート・エレメントの終了後のスキャン中に、パーサーが文書の終わりに達した。
122	文書タイプ宣言の開始が無効である可能性があることをパーサーが検出した。
123	2 番目の文書タイプ宣言をパーサーが検出した。
124	ルート・エレメント名の先頭文字が、文字、'_'、または ':' でない。
125	エレメントの最初の属性名の実頭文字が、文字、'_'、または ':' でない。
126	エレメント名の内部または後に、パーサーが無効文字を検出した。
127	属性名の後に '=' 以外の文字が続いていることをパーサーが検出した。
128	無効な属性値区切り文字をパーサーが検出した。

表 34. 終了例外 (続き)

番号	説明
130	属性名の先頭文字が、文字、'_'、または ':' でない。
131	属性名の内部または後に無効文字をパーサーが検出した。
132	空エレメント・タグが、'/' とそれに続く '>' で終わっていない。
133	エレメント終了タグ名の先頭文字が、文字、'_'、または ':' でない。
134	エレメント終了タグ名が '>' で終わっていない。
135	エレメント名の先頭文字が、文字、'_'、または ':' でない。
136	エレメント内容の中で、コメントまたは CDATA セクションの無効な開始をパーサーが検出した。
137	コメントの無効な開始をパーサーが検出した。
138	処理命令の先頭文字が、文字、'_'、または ':' でない。
139	処理命令ターゲット名の内部または後に、無効文字をパーサーが検出した。
140	処理命令が終了文字シーケンス '?>' で終わっていない。
141	文字参照またはエンティティ参照名の中で、'&' の後に無効文字をパーサーが検出した。
142	XML 宣言の中にバージョン情報がない。
143	XML 宣言の中で、'version' の後に '=' が付いていない。
144	XML 宣言の中で、バージョン宣言値が欠落しているか、誤って区切られている。
145	XML 宣言の中で、バージョン情報値に不正な文字が指定されているか、または開始と終了の区切り文字が一致しない。
146	XML 宣言の中で、バージョン情報値の終了区切り文字の後に無効文字をパーサーが検出した。
147	XML 宣言の中で、オプショナルのエンコード宣言があるべき個所に無効な属性をパーサーが検出した。
148	XML 宣言の中で、'encoding' の後に '=' が付いていない。
149	XML 宣言の中で、エンコード宣言値が欠落しているか、誤って区切られている。
150	XML 宣言の中で、エンコード宣言値に不正な文字が指定されているか、または開始と終了の区切り文字が一致しない。
151	XML 宣言の中で、エンコード宣言値の終了区切り文字の後に無効文字をパーサーが検出した。
152	XML 宣言の中で、オプショナルのスタンドアロン宣言があるべき個所に無効な属性をパーサーが検出した。
153	XML 宣言の中で、'standalone' の後に '=' が付いていない。
154	XML 宣言の中で、スタンドアロン宣言値が欠落しているか、誤って区切られている。
155	スタンドアロン宣言値が 'yes' または 'no' のどちらでもない。
156	XML 宣言の中で、スタンドアロン宣言値に不正な文字が指定されているか、または開始と終了の区切り文字が一致しない。
157	XML 宣言の中で、スタンドアロン宣言値の終了区切り文字の後に無効文字をパーサーが検出した。
158	XML 宣言が正しい文字シーケンス '?>' で終わっていないか、無効な属性を含んでいる。
159	ルート・エレメントの終了後、文書タイプ宣言の開始をパーサーが検出した。
160	ルート・エレメントの終了後、エレメントの開始をパーサーが検出した。
300	文書は EBCDIC でエンコードされているが、CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されている。

表 34. 終了例外 (続き)

番号	説明
301	文書は EBCDIC でエンコードされているが、CODEPAGE コンパイラー・オプションにはユニコードが指定されている。
302	文書は EBCDIC でエンコードされているが、CODEPAGE コンパイラー・オプションにはサポートされないコード・ページが指定されている。
303	文書は EBCDIC でエンコードされているが、CODEPAGE コンパイラー・オプションがサポートされておらず、文書エンコード宣言は空であるか、サポートされない英字のエンコード方式の別名が指定されている。
304	文書は EBCDIC でエンコードされているが、CODEPAGE コンパイラー・オプションがサポートされておらず、文書にはエンコード宣言が含まれていない。
305	文書は EBCDIC でエンコードされているが、CODEPAGE コンパイラー・オプションがサポートされておらず、文書のエンコード宣言にはサポートされる EBCDIC コード方式が指定されていない。
306	文書は ASCII でエンコードされているが、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されている。
307	文書は ASCII でエンコードされているが、CODEPAGE コンパイラー・オプションにはユニコードが指定されている。
308	文書は ASCII でエンコードされているが、CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページ、ASCII、またはユニコードが指定されていない。
309	CODEPAGE コンパイラー・オプションにはサポートされる ASCII コード・ページが指定されているが、文書はユニコードでエンコードされている。
310	CODEPAGE コンパイラー・オプションにはサポートされる EBCDIC コード・ページが指定されているが、文書はユニコードでエンコードされている。
311	CODEPAGE コンパイラー・オプションにはサポートされないコード・ページが指定されており、文書はユニコードでエンコードされている。
312	文書は ASCII でエンコードされているが、外部から指定されたエンコード方式と、エンコード宣言の中で指定されたエンコード方式が両方ともサポートされていない。
313	文書は ASCII でエンコードされているが、CODEPAGE コンパイラー・オプションがサポートされておらず、文書にはエンコード宣言が含まれていない。
314	文書は ASCII でエンコードされているが、CODEPAGE コンパイラー・オプションがサポートされておらず、文書のエンコード宣言にはサポートされる ASCII コード方式が指定されていない。
315	文書は UTF-16 リトル・エンディアンでエンコードされているが、パーサーはこのプラットフォーム上でその方式をサポートしない。
316	文書は UCS4 でエンコードされているが、パーサーはその方式をサポートしない。
317	文書のエンコード方式をパーサーが判別できない。文書は損傷している可能性がある。
318	文書は UTF-8 でエンコードされているが、パーサーはその方式をサポートしない。
319	文書は UTF-16 ビッグ・エンディアンでエンコードされているが、パーサーはこのプラットフォーム上でその方式をサポートしない。
501, 502, 503	PLISAX(AIB) で内部エラーが発生した。IBM サポート部門に連絡してください。
500	PLISAXA 内部データ構造へのメモリー割り振りが失敗した。アプリケーション・プログラムが使用できるストレージ量を増やしてください。
520	PLISAXB 内部データ構造へのメモリー割り振りが失敗した。アプリケーション・プログラムが使用できるストレージ量を増やしてください。
521	PLISAX(AIB) で内部エラーが発生した。IBM サポート部門に連絡してください。

表 34. 終了例外 (続き)

番号	説明
523	PLISAXB がファイル入出力エラーを検出した。
524	ファイル・システムから XML 文書をキャッシュに入れようとして PLISAXB でメモリ割り振りに失敗した。アプリケーション・プログラムが使用できるストレージ量を増やしてください。
525	サポートされない URI スキームが PLISAXB に指定された。
526	PLISAXB に提供された XML 文書の文字数が、最小限の 4 文字より少ないか多すぎた。
527, 560	PLISAX(AIB) で内部エラーが発生した。 IBM サポート部門に連絡してください。
561	PLISAX(AIB) のどちらにもイベント・ハンドラーが指定されていない。
562, 563, 580, 581	PLISAX(AIB) で内部エラーが発生した。 IBM サポート部門に連絡してください。
600 から 99,999	内部エラーこのエラーはサービス技術員に報告してください。

第 16 章 PLISAXC XML パーサーの使用

PLISAXC 組み込みサブルーチンは、プログラムがインバウンド XML 文書を取り込んで、適格性を検査して、その内容を処理できるようにする、基本的な XML 構文解析機能を提供します。

このサブルーチンでは XML の生成は提供しておらず、代わりに、PL/I プログラム・ロジックによって行うか、または XMLCHAR 組み込み関数を使用する必要があります。

PLISAXC には、AMODE 24 でサポートされていないという点を除いて、特別な環境要件はありません。CICS、IMS、MQ Series、さらに z/OS のバッチと TSO など、主要なすべてのランタイム環境で実行できます。

PLISAXC 組み込みサブルーチンは、PLISAXA および PLISAXB サブルーチンとは大幅に異なりますが、多くの点で類似しています。以下の説明の一部は、前章でも説明しています。

概要

XML 構文解析用のインターフェースには、大きく分けてイベント・ベースとツリー・ベースの 2 種類があります。

イベント・ベース API の場合、パーサーはコールバックによってアプリケーションにイベントを報告します。報告されるイベントは、文書の開始、エレメントの開始などです。アプリケーションは、パーサーによって報告されるイベントを処理するためのハンドラーを備えています。Simple API for XML (SAX) は、業界標準のイベント・ベース API の一例です。

ツリー・ベース API (文書オブジェクト・モデル (DOM) など) の場合、パーサーは XML をツリー・ベースの内部表現に変換します。ツリーをナビゲートするためのインターフェースが提供されています。

IBM PL/I は、PLISAXC を使用して、XML 文書の構文解析用に SAX のようなイベント・ベースのインターフェースを提供します。パーサーは、対応する文書フラグメントへの参照を渡して、アプリケーション提供のパーサー・イベント用のハンドラーを呼び出します。

パーサーには次の特性があります。

- ・ 高性能であるが非標準のインターフェースを提供します。
- ・ ユニコード UTF-16、UTF-8、または後述の 1 バイト・コード・ページのいずれかでエンコードされた XML ファイルをサポートします。
- ・ パーサーは有効性検査を行いませんが、適格性を部分的に検査します。セクション 2.5.10 を参照してください。

XML 文書の準拠レベルには適格性と有効性の 2 つがあり、どちらのレベルも XML 標準に定義されています。XML 標準は、<http://www.w3c.org/XML/> に掲載さ

これらのすべての ENTRY には、(唯一のパラメーターである場合もある) 最初のパラメーター (プログラムによって PLISAXC に渡されるユーザー・トークン) があります。

次に示す 19 個のイベントの説明は、図 94 の XML 文書の例に対応しています。この説明にある「XML テキスト」という用語は、イベントに渡されるポインターと長さに基づくストリングを意味しています。

```
xmlDocument =
  '<?xml version="1.0" standalone="yes"?>'
  '<!--This document is just an example-->'
  '<sandwich>'
  '<bread type="baker&quot;s best"/>'
  '<?spread please use real mayonnaise ?>'
  '<meat>Ham & turkey</meat>'
  '<filling>Cheese, lettuce, tomato, etc.</filling>'
  '<![CDATA[We should add a <relish> element in future!]]>'
  '</sandwich>';
```

図 94. サンプル XML 文書

XML 文書の内容によって、パーサーは以下のイベントを認識できます。

start_of_document

このイベントは、文書の構文解析が開始されるときに 1 回発生します。パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。

version_information

このイベントは、オプショナルの XML 宣言内のバージョン情報に対して発生します。パーサーは、バージョン値 (上記の例では "1.0") が入ったテキストのアドレスと長さを渡します。

encoding_declaration

このイベントは、XML 宣言内でオプショナルのエンコード宣言に対して発生します。パーサーは、エンコード方式値が入ったテキストのアドレスと長さを渡します。

standalone_declaration

このイベントは、XML 宣言内でオプショナルのスタンドアロン宣言に対して発生します。パーサーは、スタンドアロン値 (上記の例では "yes") が入ったテキストのアドレスと長さを渡します。

document_type_declaration

このイベントは、パーサーが文書タイプ宣言を検出したときに発生します。文書タイプ宣言は、文字シーケンス "<!DOCTYPE" から始まって ">" 文字で終わるもので、その間には内容を記述するやや複雑な文法規則が入ります。パーサーは、開始と終了の文字シーケンスを含む宣言全体が入ったテキストのアドレスと長さを渡し

ます。このイベントは、XML テキストに区切り文字が含まれる唯一のイベントです。上記の例には、文書タイプ宣言はありません。

end_of_document

このイベントは、文書の構文解析が完了したときに 1 回発生します。パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。

start_of_element

このイベントは、エレメント開始タグ、または空エレメント・タグごとに 1 回発生します。パーサーは、エレメント名が入ったテキストのアドレスと長さ、および適用される名前空間情報を渡します。例の構文解析中に最初に発生する `start_of_element` イベントの場合、このテキストはストリング "sandwich" です。

attribute_name

このイベントは、エレメント開始タグ、または空エレメント・タグ内の属性ごとに、有効な名前を認識した後に発生します。パーサーは、属性名が入ったテキストのアドレスと長さ、および適用される名前空間情報を渡します。例にある属性名は "type" だけです。

attribute_characters

このイベントは、属性値ごとに発生します。パーサーは、フラグメントが入ったテキストのアドレスと長さを渡します。属性値は通常、次のように複数の行に分割されている場合でもただ 1 つのストリングで構成されます。

```
<element attribute="This attribute value is  
split across two lines"/>
```

end_of_element

このイベントは、エレメント終了タグ、または空エレメント・タグごとに、パーサーがタグの終了不等号括弧を認識したときに 1 回発生します。パーサーは、エレメント名が入ったテキストのアドレスと長さ、および適用される名前空間情報を渡します。

start_of_CDATA_section

このイベントは、CDATA セクションが開始されると発生します。CDATA セクションはストリング "<![CDATA[" で始まり、ストリング "]" で終わるもので、他の場合には XML マークアップとして認識される文字を含むテキストのブロックを「エスケープ」するために使用されます。パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。このイベントの後、パーサーは、これらの区切り文字間にある CDATA セクションの内容を 1 つ以上の `content-characters` イベントとして渡します。例えば上記の例では、`content-characters` イベントとしてテキスト "We should add a <relish> element in future!" が渡されます。

end_of_CDATA_section

このイベントは、パーサーが CDATA セクションの終了を認識したときに発生します。パーサーは、このイベントには (ユーザー・トークンを除いて) パラメーターを渡しません。

content_characters

このイベントは、XML 文書の「本体」である、エレメントの開始タグと終了タグの間にある文字データを表します。パーサーは、このデータが入ったテキストのアドレスと長さを渡します。テキストは通常、次のように複数の行に分割されている場合でもただ 1 つのストリングで構成されます。

```
<element1>This character content is  
split across two lines</element1>
```

また、パーサーは、次のイベントが内容の一部を形成する追加文字を提供するかどうかを示す、フラグ・バイトも渡します。これは、開始タグと終了タグの間に多くのデータが存在する場合に当てはまります。

またパーサーは、content_characters イベントを使用して CDATA セクションのテキストをアプリケーションに渡します。

processing_instruction

処理命令 (PI) を使用すると、XML 文書にアプリケーション用の特別な命令を含めることができます。このイベントは、パーサーが PI 開始文字シーケンス "<?" に続く名前を認識したときに発生します。さらにこのイベントは、処理命令 (PI) ターゲットに続く、PI 終了文字シーケンス "?>" の直前までのデータを対象とします。データの末尾にある空白文字は含まれますが、先頭にある空白文字は含まれません。パーサーは、ターゲットが入ったテキスト (例では "spread") のアドレスと長さ、およびデータが入ったテキストのアドレスと長さ (例では "please use real mayonnaise") を渡します。

comment

このイベントは、XML 文書内のコメントに対して発生します。パーサーは、開始および終了コメント区切り文字 (それぞれ "<!--" と "-->") の間にあるテキストのアドレスと長さを渡します。例では、唯一のコメントのテキストは "This document is just an example" です。

namespace_declare

このイベントは、XML 文書内の名前空間宣言に対して発生します。パーサーは、名前空間接頭部のアドレスと長さ (存在する場合)、および名前空間 URI のアドレスと長さを渡します。名前空間接頭部が存在しない場合には、渡される長さはゼロであり、アドレスの値は使用されません。PLIXSAXA および PLISAXB 組み込みサブルーチンには、対応するイベントはありません。

end_of_input

このイベントは、パーサーが現在の入力バッファの最後に到達すると常に発生します。パーサーは、(BYVALUE ユーザー・トークンとともに) 2 つの BYADDR パラメーター (処理対象の次のバッファのアドレスおよび長さ) を渡します。なお、

BYADDR パラメーターをとるイベントは、このイベントおよび content-characters イベントのみですが、このイベントのみが、呼び出されるイベントが変更する必要があるパラメーターをとります。PLIXSAXA および PLISAXB 組み込みサブルーチンには、対応するイベントはありません。また、このイベントによって、PLISAXC は任意のサイズの XML 文書の構文解析を行うことができます。

unresolved_reference

このイベントは、XML 文書内の未解決の参照に対して発生します。パーサーは、未解決の参照のアドレスおよび長さを渡します。

exception

XML 文書の処理中にエラーを検出すると、パーサーはこのイベントを生成します。

イベント関数に渡されるパラメーター

イベント関数はすべて、パーサーへの戻りコードである BYVALUE FIXED BIN(31) 値を返す必要があります。ゼロ以外の値が返された場合は、パーサーは終了します。

これらの関数すべてに、最初の引数として BYVALUE POINTER が渡されます。この値は、元は組み込み関数への 2 番目の引数として渡されたトークン値です。

次に示す例外を除き、イベントのテキスト・エレメントのアドレスと長さを提供する BYVALUE POINTER と BYVALUE FIXED BIN(31) も、すべての関数に渡されます。例外の関数/イベントは、次のとおりです。

start_of_document

ユーザー・トークン以外の引数は渡されません。

end_of_document

ユーザー・トークン以外の引数は渡されません。

start_of_CDATA

ユーザー・トークン以外の引数は渡されません。

end_of_CDATA

ユーザー・トークン以外の引数は渡されません。

start_of_element

通常の 3 つのパラメーターに加えて、次の 4 つの引数が渡されます。

1. 名前空間接頭部のアドレスを示す BYVALUE POINTER
2. 名前空間接頭部の長さを示す BYVALUE FIXED BIN(31)
3. 名前空間 URI のアドレスを示す BYVALUE POINTER
4. 名前空間 URI の長さを示す BYVALUE FIXED BIN(31)

end_of_element

通常の 3 つのパラメーターに加えて、次の 4 つの引数が渡されます。

1. 名前空間接頭部のアドレスを示す BYVALUE POINTER
2. 名前空間接頭部の長さを示す BYVALUE FIXED BIN(31)
3. 名前空間 URI のアドレスを示す BYVALUE POINTER
4. 名前空間 URI の長さを示す BYVALUE FIXED BIN(31)

attribute_name

通常の 3 つのパラメーターに加えて、次の 4 つの引数が渡されます。

1. 名前空間接頭部のアドレスを示す BYVALUE POINTER
2. 名前空間接頭部の長さを示す BYVALUE FIXED BIN(31)
3. 名前空間 URI のアドレスを示す BYVALUE POINTER
4. 名前空間 URI の長さを示す BYVALUE FIXED BIN(31)

namespace_declare

ユーザー・トークンに加えて、次の 4 つの引数が渡されます。

1. 名前空間接頭部のアドレスを示す BYVALUE POINTER
2. 名前空間接頭部の長さを示す BYVALUE FIXED BIN(31)
3. 名前空間 URI のアドレスを示す BYVALUE POINTER
4. 名前空間 URI の長さを示す BYVALUE FIXED BIN(31)

content_characters

通常の 3 つのパラメーターに加えて、次の 1 つの引数が渡されます。

- 以下を示す、BYADDR ALIGNED BIT(8) フラグ・バイト
 - 次のイベントに content-characters がさらに提供されるかどうか (最初のビットがオンである場合には TRUE。つまり、このフィールドと '80'BX との AND 演算の結果が非 nul である場合に TRUE)
 - XML に変換し戻す際にエスケープする必要がある文字が存在しないかどうか (2 番目のビットがオンの場合は TRUE。つまり、このフィールドと '40'BX との AND 演算の結果が非 nul である場合に TRUE)

なお、このエントリーは、OPTIONS(NODESCRIPTOR) で宣言する必要もあります。

end_of_input

ユーザー・トークンに加えて、次の 2 つの引数が渡されます。

1. 次の入力バッファのアドレスを示す BYADDR POINTER
2. 次の入力バッファの長さを示す BYADDR FIXED BIN(31)

processing_instruction

ユーザー・トークンに加えて、次の 4 つの引数が渡されます。

1. ターゲット・テキストのアドレスを示す BYVALUE POINTER
2. ターゲット・テキストの長さを示す BYVALUE FIXED BIN(31)
3. データ・テキストのアドレスを示す BYVALUE POINTER
4. データ・テキストの長さを示す BYVALUE FIXED BIN(31)

exception

ユーザー・トークンに加えて、次の 3 つの引数が渡されます。

1. 文書内での問題のテキストのバイト・オフセットを示す BYVALUE FIXED BIN(31)
2. 例外の戻りコードを示す BYVALUE FIXED BIN(31)
3. 例外の理由コードを示す BYVALUE FIXED BIN(31)

イベントにおける差異

以下のイベントは、PLISAXA/B イベント構造体の一部ですが、PLISAXC には存在しません。

- attribute_predefined_reference
- attribute_character_reference
- content_predefined_reference
- content_character_reference
- unknown_attribute_reference
- unknown_content_reference
- start_of_prefix_mapping
- end_of_prefix_mapping

以下のイベントは、PLISAXA/B イベント構造体の一部ではありませんが、PLISAXC には存在します。

- namespace_declare
- unresolved_reference
- end_of_input

PLISAXA/B と PLISAXC の両方に共通するイベントの一部では、(通常のユーザー・トークンを除いて) 異なるパラメーターが渡されます。

start_of_document

パラメーターが渡されない

start_of_element

名前空間データも渡される

end_of_element

名前空間データも渡される

attribute_name

名前空間データも渡される

content_characters

フラグ・バイトも渡される

exception

エラー ID の代わりに戻りコードおよび理由コードが渡される

start_of_cdata

パラメーターが渡されない

end_of_cdata

パラメーターが渡されない

XML 文書のコード化文字セット

PLISAXC 組み込みサブルーチンは、Unicode UTF-16 を使用してエンコードされた WIDECHAR の XML 文書、または UTF-8 または明示的にサポートされている後述の 1 バイト文字セットのいずれかを使用してエンコードされた CHARACTER の XML 文書のみをサポートしています。パーサーは、XML 文書のエンコード方式に

関する情報ソースを 3 つまで使用し、これらのソース間で矛盾を検出した場合は、次のように例外 XML イベントをシグナル通知します。

1. パーサーは、文書の最初の文字を検査することによって文書の基本エンコードを判別します。
2. ステップ 1 が正常に完了した場合、パーサーはエンコード宣言を検索します。
3. 最後に、パーサーは PLISAXC 組み込みサブルーチン呼び出しのコード・ページ値を参照します。このパラメーターが省略された場合、デフォルトで 사용되는値は、明示指定またはデフォルトの CODEPAGE コンパイラー・オプションの値です。

XML 文書の最初に、後述のサポート対象のコード・ページを指定した XML 宣言がある場合は、その宣言が基本文書エンコード、または PLISAXC 組み込みサブルーチンからのエンコード情報と矛盾しなければ、パーサーはエンコード宣言を受け入れます。XML 文書に XML 宣言自体がない場合、または XML 宣言がエンコード宣言を省略している場合は、基本文書エンコードと矛盾しなければ、パーサーは PLISAXC 組み込みサブルーチンからのエンコード情報を使用して文書进行处理します。

サポートされるコード・ページ

次の表で、最初の番号はユーロ国別拡張コード・ページ (ECECP)、2 番目の番号は国別拡張コード・ページ (CECP) のものです。

CCSID	説明
01208	Unicode UTF-8
01047	Latin 1/オープン・システム
01140、00037	米国、カナダなど
01141、00273	オーストリア、ドイツ
01142、00277	デンマーク、ノルウェー
01143、00278	フィンランド、スウェーデン
01144、00280	イタリア
01145、00284	スペイン、ラテンアメリカ (スペイン語)
01146、00285	英国
01147、00297	フランス
01148、00500	国際
01149、00871	アイスランド

コード・ページの指定

文書の XML 宣言にエンコード宣言がない場合、または XML 宣言自体がない場合は、パーサーは、PLISAXC 組み込みサブルーチン呼び出しで提供されたエンコード情報を、文書の基本エンコードと組み合わせて使用します。

ほとんどの XML 文書の最初にある XML 宣言の中で、文書のエンコード情報を指定することもできます。エンコード宣言を含む XML 宣言の一例を次に示します。

```
<?xml version="1.0" encoding="ibm-1140"?>
```

XML 文書にエンコード宣言がある場合は、 PLISAXC 組み込みサブルーチンから提供されるエンコード情報、および文書の基本エンコードと、エンコード宣言が整合していることを確認してください。エンコード宣言、PLISAXC 組み込みサブルーチンから提供されるエンコード情報、および文書の基本エンコードの間に矛盾がある場合、パーサーは例外 XML イベントをシグナル通知します。

エンコード宣言は、次のように宣言します。

番号の使用

次のいずれかの接頭部を付けて (大文字と小文字の組み合わせは自由)、 CCSID 番号を指定できます (先行ゼロなし、または任意数の先行ゼロを付けて)。

IBM_	CP	CCSID_
IBM-	CP_	CCSID-
	CP-	

別名の使用

次に示す別名がサポートされており、任意に使用できます (大文字と小文字の組み合わせは自由)。

コード・ページ	サポートされる別名
037	EBCDIC-CP-US、EBCDIC-CP-CA、EBCDIC-CP-WT、EBCDIC-CP-NL
500	EBCDIC-CP-BE、EBCDIC-CP-CH
813	ISO-8859-7、ISO_8859-7
819	ISO-8859-1、ISO_8859-1
920	ISO-8859-9、ISO_8859-9
1200	UTF-16

例外

例外イベントが発生した場合は、そのイベントに渡される理由コードおよび戻りコードは、XML System Services パーサーからのものです。これらの戻りコードおよび理由コードの意味については、このパーサーで提供される資料で説明されています。

単純な文書での例

次の例は、PLISAXC 組み込みサブルーチンの使用を示すもので、前述の XML 文書の例を使用しています。この例は、本質的には前の章で使用した例と同じです (ただし、名前空間は使用せず、PLISAXC が最初に呼び出されたときにすべての入力渡されます (その結果、end_of_input イベントは呼び出しません))。しかし、この例を挙げることによって、これらの 2 つのパーサーの動作の対比が容易になります。

```

saxtest: package exports(saxtest);

define alias event
  limited entry( pointer, pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_with_flag
  limited entry( pointer, pointer, fixed bin(31),
                bit(8) aligned )
  returns( byvalue fixed bin(31) )
  options( nodestructor byvalue linkage(optlink) );

define alias event_with_namespace
  limited entry( pointer, pointer, fixed bin(31),
                pointer, fixed bin(31),
                pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_without_data
  limited entry( pointer )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_pi
  limited entry( pointer, pointer, fixed bin(31),
                pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_namespace_dcl
  limited entry( pointer, pointer, fixed bin(31),
                pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_exception
  limited entry( pointer, fixed bin(31),
                fixed bin(31),
                fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

define alias event_end_of_input
  limited entry( pointer,
                pointer byaddr,
                fixed bin(31) byaddr )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

```

図 95. PLISAXC のコーディング例 - 型宣言

```

saxtest: proc options( main );

dcl
1 eventHandler static

,2 e01 type event_without_data
    init( start_of_document )

,2 e02 type event
    init( version_information )

,2 e03 type event
    init( encoding_declaration )

,2 e04 type event
    init( standalone_declaration )

,2 e05 type event
    init( document_type_declaration )

,2 e06 type event_without_data
    init( end_of_document )

,2 e07 type event_with_namespace
    init( start_of_element )

,2 e08 type event_with_namespace
    init( attribute_name )

,2 e09 type event
    init( attribute_characters )

,2 e10 type event_with_namespace
    init( end_of_element )

,2 e11 type event_without_data
    init( start_of_CDATA )

,2 e12 type event_without_data
    init( end_of_CDATA )

,2 e13 type event_with_flag
    init( content_characters )

,2 e14 type event_pi
    init( processing_instruction )

,2 e15 type event
    init( comment )

,2 e16 type event_namespace_dcl
    init( namespace_declare )

,2 e17 type event_end_of_input
    init( end_of_input )

,2 e18 type event
    init( unresolved_reference )

,2 e19 type event_exception
    init( exception )
;

```

図 96. PLISAXC のコーディング例 - イベント構造体

```
dc1 token      char(8);

dc1 xmlDocument char(4000) var;

xmlDocument =
'|<?xml version="1.0" standalone="yes"?>'
'|<!--This document is just an example-->'
'|<sandwich>'
'|<bread type="baker's best"/>'
'|<?spread please use real mayonnaise ?>'
'|<meat>Ham & turkey</meat>'
'|<filling>Cheese, lettuce, tomato, etc.</filling>'
'|<![CDATA[We should add a <relish> element in future!]]>'.
'|</sandwich>'
'|';

call plisaxc( eventHandler,
              addr(token),
              addrrdata(xmlDocument),
              length(xmlDocument) );

end;
```

図 97. PLISAXC のコーディング例 - メインルーチン

```

dcl chars char(32000) based;

start_of_document:
proc( userToken )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' length=' || tokenlength );

return(0);
end;

version_information:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

return(0);
end;

encoding_declaration:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

return(0);
end;

standalone_declaration:
proc( userToken, xmlToken, TokenLength )
returns( byvalue fixed bin(31) )
options( byvalue linkage(optlink) );

dcl userToken      pointer;
dcl xmlToken       pointer;
dcl tokenLength    fixed bin(31);

put skip list( lowercase( procname() )
|| ' <' || substr(xmltoken->chars,1,tokenlength ) || '>' );

return(0);
end;

```

図 98. PLISAXC のコーディング例 - イベント・ルーチン (1/6)

```

document_type_declaration:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

namespace_declare:
  proc( userToken, nsPrefix, nsPrefixLength,
        nsUri, nsUriLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl nsPrefix       pointer;
    dcl nsPrefixLength fixed bin(31);
    dcl nsUri          pointer;
    dcl nsUriLength    fixed bin(31);

    put skip list( lowercase( procname() ) );
    put skip list( 'prefix = '
      || ' <' || substr(nsPrefix->chars,1,nsPrefixLength) || '>' );
    put skip list( 'Uri = '
      || ' <' || substr(nsUri->chars,1,nsUriLength) || '>' );

    return(0);
  end;

end_of_document:
  proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

```

図 98. PLISAXC のコーディング例 - イベント・ルーチン (2/6)

```

start_of_element:
  proc( userToken, xmlToken, TokenLength,
        nsPrefix, nsPrefixLength,
        nsUri, nsUriLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dc1 userToken      pointer;
  dc1 xmlToken       pointer;
  dc1 tokenLength    fixed bin(31);
  dc1 nsPrefix       pointer;
  dc1 nsPrefixLength fixed bin(31);
  dc1 nsUri          pointer;
  dc1 nsUriLength    fixed bin(31);

  put skip list( lowercase( procname() )
    || ' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

  return(0);
end;

attribute_name:
  proc( userToken, xmlToken, TokenLength,
        nsPrefix, nsPrefixLength,
        nsUri, nsUriLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dc1 userToken      pointer;
  dc1 xmlToken       pointer;
  dc1 tokenLength    fixed bin(31);
  dc1 nsPrefix       pointer;
  dc1 nsPrefixLength fixed bin(31);
  dc1 nsUri          pointer;
  dc1 nsUriLength    fixed bin(31);

  put skip list( lowercase( procname() )
    || ' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

  return(0);
end;

end_of_element:
  proc( userToken, xmlToken, TokenLength,
        nsPrefix, nsPrefixLength,
        nsUri, nsUriLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dc1 userToken      pointer;
  dc1 xmlToken       pointer;
  dc1 tokenLength    fixed bin(31);
  dc1 nsPrefix       pointer;
  dc1 nsPrefixLength fixed bin(31);
  dc1 nsUri          pointer;
  dc1 nsUriLength    fixed bin(31);

  put skip list( lowercase( procname() )
    || ' <' || substr(xmltoken->chars,1,tokenlength) || '>' );

  return(0);
end;

```

図 98. PLISAXC のコーディング例 - イベント・ルーチン (3/6)

```

content_characters:
  proc( userToken, xmlToken, TokenLength, flags )
    returns( byvalue fixed bin(31) )
    options( nodestructor, byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);
    dcl flags          bit(8) aligned;

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    if flags = 'b then;
    else
      put skip list( '!!flags = ' || flags );

    return(0);
  end;

attribute_characters:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;
    dcl xmlToken       pointer;
    dcl tokenLength    fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

    return(0);
  end;

start_of_CDATA:
  proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

end_of_CDATA:
  proc( userToken )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    dcl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

```

図 98. PLISAXC のコーディング例 - イベント・ルーチン (4/6)

```

processing_instruction:
  proc( userToken,
        piTarget, piTargetLength,
        piData, piDataLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl piTarget       pointer;
  dcl piTargetLength fixed bin(31);
  dcl piData         pointer;
  dcl piDataLength   fixed bin(31);

  put skip list( lowercase( procname() )
    || ' <' || substr(piTarget->chars,1,piTargetLength) || '>' );

  return(0);
end;

comment:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
    || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

  return(0);
end;

unresolved_reference:
  proc( userToken, xmlToken, TokenLength )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl tokenLength    fixed bin(31);

  put skip list( lowercase( procname() )
    || ' <' || substr(xmlToken->chars,1,tokenLength) || '>' );

  return(0);
end;

exception:
  proc( userToken, xmlToken, currentOffset, errorID )
  returns( byvalue fixed bin(31) )
  options( byvalue linkage(optlink) );

  dcl userToken      pointer;
  dcl xmlToken       pointer;
  dcl currentOffset  fixed bin(31);
  dcl errorID        fixed bin(31);

  put skip list( lowercase( procname() )
    || ' errorid =' || errorid );

  return(0);
end;

```

```
end_of_input:
  proc( userToken, addr_xml, length_xml )
    returns( byvalue fixed bin(31) )
    options( byvalue linkage(optlink) );

    decl userToken      pointer;
    decl addr_xml       byaddr pointer;
    decl length_xml     byaddr fixed bin(31);

    return(1);
  end;
end;
```

図 98. *PLISAXC* のコーディング例 - イベント・ルーチン (6/6)

前のプログラムから生成される出力は、次のとおりです。

```
start_of_document
version_information <1.0>
standalone_declaration <yes>
comment <This document is just an example>
start_of_element <sandwich>
prefix = <>
Uri = <>
start_of_element <bread>
prefix = <>
Uri = <>
attribute_name <type>
prefix = <>
Uri = <>
attribute_characters <baker's best>
end_of_element <bread>
prefix = <>
Uri = <>
processing_instruction <spread>
piData = <please use real mayonnaise >
start_of_element <meat>
prefix = <>
Uri = <>
content_characters <Ham & turkey>
end_of_element <meat>
prefix = <>
Uri = <>
start_of_element <filling>
prefix = <>
Uri = <>
content_characters <Cheese, lettuce, tomato, etc.>
!!flags = 01000000
end_of_element <filling>
prefix = <>
Uri = <>
start_of_cdata
content_characters <We should add a <relish> element in future >
end_of_cdata
end_of_element <sandwich>
prefix = <>
Uri = <>
end_of_document
```

図 99. PLISAXC のコーディング例 - プログラム出力

第 17 章 PLIDUMP の用法

このセクションでは、PLIDUMP を呼び出すのに使用されるダンプ・オプションと構文を記載し、ダンプに含まれていてユーザーがユーザー・ルーチンをデバッグする際に役立つ PL/I に固有の情報について説明します。

注: PLIDUMP は各国語サポートの標準に準拠しています。

図 100 に、z/OS 言語環境プログラム・ダンプを作成するために PLIDUMP を呼び出す PL/I ルーチンの例を示します。この例では、メインルーチン PLIDMP が PLIDMPA を呼び出し、次にそれが PLIDMPB を呼び出します。PLIDUMP の呼び出しは、ルーチン PLIDMPB 内で行います。

```
%PROCESS MAP GOSTMT SOURCE STG LIST OFFSET LC(101);
PLIDMP: PROC OPTIONS(MAIN) ;

  Declare (H,I) Fixed bin(31) Auto;
  Declare Names Char(17) Static init('Bob Teri Bo Jason');
  H = 5; I = 9;
  Put skip list('PLIDMP Starting');
  Call PLIDMPA;

  PLIDMPA: PROC;
  Declare (a,b) Fixed bin(31) Auto;
  a = 1; b = 3;
  Put skip list('PLIDMPA Starting');
  Call PLIDMPB;

  PLIDMPB: PROC;
  Declare 1 Name auto,
          2 First Char(12) Varying,
          2 Last Char(12) Varying;
  First = 'Teri';
  Last = 'Gillispy';
  Put skip list('PLIDMPB Starting');
  Call PLIDUMP('TBFC','PLIDUMP called from procedure PLIDMPB');
  Put Data;
  End PLIDMPB;

End PLIDMPA;

End PLIDMP;
```

図 100. PLIDUMP を呼び出す PL/I ルーチンの例

PLIDUMP の構文とオプションを次に示します。

▶▶—PLIDUMP—(*character-string-expression 1*,*character-string-expression 2*)——▶▶

character-string-expression 1

次のうちの 1 つ以上のもので構成されるダンプ・オプション文字ストリングです。

- A** マルチタスキング・プログラムのすべてのタスクに関連する要求情報。
- B** BLOCKS (PL/I 16 進ダンプ)。
- C** 続行する。ルーチンはダンプの完了後、続行されます。

- E** マルチタスキング・プログラムの現行タスクからの出口。プログラムは、要求されたダンプの完了後、実行を継続します。
- F** FILES。
- H** STORAGE。
- これには、すべての言語環境プログラム・ストレージが含まれます。したがって、ALLOCATE ステートメントを介して獲得されるすべてのALLOCATE ストレージおよび CONTROLLED ストレージが含まれます。
- 注: CEESNAP の DD 名は、PL/I ルーチンのスナップ・ダンプを作成できるよう、H オプションを使用して指定してください。ただし、これが省略されても、言語環境プログラム はメッセージを出しますが、極めて有用な情報を含むダンプを生成します。
- K** BLOCKS (CICS 下での実行時)。トランザクション作業域が組み込まれます。
- NB** NOBLOCKS。
- NF** NOFILES。
- NH** NOSTORAGE。
- NK** NOBLOCKS (CICS 下での実行時)。
- NT** NOTRACEBACK。
- O** マルチタスキング・プログラムの現行タスクに関連する唯一の情報。
- S** 停止する。エンクレーブはダンプで終了します。
- T** TRACEBACK。

T、F、および C がデフォルト・オプションです。

character-string-expression 2

ダンプ・ヘッダーとして印刷される最長 80 文字のユーザー識別文字ストリング。

PLIDUMP の使用上の注意

PLIDUMP を使用する場合は、次の考慮事項が適用されます。

- ルーチンが PLIDUMP を何回か呼び出すような場合には、PLIDUMP 呼び出しごとに固有のユーザー ID を使ってください。そうすると、各ダンプの開始を簡単に識別できます。
- DD 名が PLIDUMP、PL1DUMP、または CEEDUMP である DD ステートメントは、ダンプ用のデータ・セットの定義に使うことができます。
- PLIDUMP、PL1DUMP、または CEEDUMP DD ステートメントで定義されたデータ・セットでは、ダンプ・レコードの折り返しを防ぐために 133 以上の論理レコード長 (LRECL) を指定する必要があります。これらのいずれかの DD 内でターゲットとして SYSOUT を使用する場合は、MSGFILE(SYSOUT,FBA,133,0) または MSGFILE(SYSOUT,VBA,137,0) を指定して、行が折り返されないようにする必要があります。

- PLIDUMP への呼び出し内に H オプションを指定するときには、PL/I ライブラリーは、仮想記憶域のダンプを入手するため、OS SNAP マクロを出します。PLIDUMP の最初の起動で SNAP ID は 0 になります。それ以降の各呼び出しでは、ID が 1 ずつ、最大 256 まで増え、それから 0 にリセットされます。
- PLIDUMP を使ったスナップ・ダンプは、z/OS 下でのみサポートされます。スナップ・ダンプは、CICS 環境では作成されません。
 - SNAP が正常に完了しない場合は、CEE3DMP DUMP ファイルに次のメッセージが表示されます。
Snap was unsuccessful
 - SNAP が正常に完了した場合は、CEE3DMP に次のメッセージが表示されます。
Snap was successful; snap ID = nnn

この nnn は、上記の SNAP ID に対応します。SNAP が失敗した場合は、ID はインクリメントされません。
- プログラム単位名、プログラム単位アドレス、およびプログラム単位オフセットをダンプ・トレースバック・テーブルに正しくリストするには、TEST(NONE,NOSYM) 以外のコンパイル時オプションを指定して PL/I プログラム単位をコンパイルしておく必要があります。例えば、TEST(NOSYM,NOHOOK,BLOCK) をオプションとして指定することができます。

各システム・プラットフォーム間での移植性を確保するには、PLIDUMP を使って PL/I ルーチンのダンプを生成します。

PLIDUMP 出力内の変数の検出

PLIDUMP 出力内で変数を検出するには、MAP オプションを指定してプログラムをコンパイルする必要があります。MAP オプションを指定すると、AUTOMATIC または STATIC であるすべてのレベル 1 変数の AUTOMATIC および STATIC ストレージ内でのオフセットを示すテーブルが、コンパイラーによってリストに追加されます。

構造体内の 1 つの要素である変数を検出することは、AGGREGATE オプションを指定してプログラムをコンパイルする場合にも便利です。このオプションを指定すると、プログラム内のすべての構造体のすべての要素のオフセットを示すテーブルが、コンパイラーによってリストに追加されます。

AUTOMATIC 変数の検出

ダンプ内で AUTOMATIC 変数を検出するには、MAP オプション (および、必要なら AGGREGATE オプション) を指定した出力を使用して、AUTOMATIC 内でのオフセットを検出する必要があります。PLIDUMP が B オプションを指定して起動された場合、そのダンプ出力には、それぞれのブロックごとの「動的保存域」(つまり、DSA) の 16 進ダンプが含まれます。これが、そのブロックの自動ストレージになります。

例えば、以下の簡単なプログラムがあるとします。

Compiler Source

```
Line.File
2.0      test: proc options(main);
3.0
4.0      dcl a fixed bin(31);
5.0      dcl b fixed bin(31);
6.0
7.0      on error
8.0          begin;
9.0              call plidump('TFBC');
10.0         end;
11.0
12.0      a = 0;
13.0      b = 29;
14.0      b = 17 / a;
```

このプログラムに対するコンパイラの MAP オプションの結果は、実際には右側にもう 1 つ列があること、列の間のスペースはもっと離れていることを除き、以下のようになります。

```
* * * * * S T O R A G E   O F F S E T   L I S T I N G   * * * * *
IDENTIFIER DEFINITION ATTRIBUTES
```

```
A          1-0:4      Class = automatic, Location = 160 : 0xA0(r13),
B          1-0:5      Class = automatic, Location = 164 : 0xA4(r13),
```

したがって、A はレジスター 13 の 16 進 A0 のオフセット位置にあり、B はレジスター 13 の 16 進 A4 のオフセット位置にあることになります (ここで、レジスター 13 は DSA を指しています)。

このプログラムでは PLIDUMP が B オプションを指定して呼び出されているため、現行の呼び出しチェーン内のそれぞれのブロックごとに、自動ストレージの 16 進ダンプが含まれています。これは、以下のようになります(これも、右側の列が省略されています)。

```
Dynamic save area (TEST): 0AD963C8
+000000 0AD963C8 10000000 0AD96188 00000000 00000000
+000020 0AD963E8 00000000 00000000 00000000 00000000
+000040 0AD96408 00000000 00000000 00000000 0AD96518
+000060 0AD96428 00000000 00000000 00000000 00000000
+000080 0AD96448 - +00009F 0AD96467 same as above
+0000A0 0AD96468 00000000 0000001D 00100000 00000000
+0000C0 0AD96488 0B300000 0A700930 0AD963C8 00000000
+0000E0 0AD964A8 00000000 00000000 00000000 00000000
+000100 0AD964C8 0AA47810 0A70E6D0 0AD96540 0AD960F0
+000120 0AD964E8 00000001 0A70F4F8 0AD96318 00000000
+000140 0AD96508 00000000 00000000 00000000 00000000
```

AUTOMATIC 内で、A は 16 進オフセット A0 のところにあり、B は 16 進オフセット A4 のところにあるので、ダンプでは、A と B がそれぞれ (予想通りの) 16 進値である 00000000 と 0000001D になっていることを示しています。

コンパイラ・オプション OPT(2) および OPT(3) を指定すると、一部の変数 (特に FIXED BIN および POINTER スカラー変数) がストレージに割り振られないことがあり、そのためダンプ出力に表示されないことに注意してください。

STATIC 変数の検出

RENT オプションを指定してコードをコンパイルした場合、静的変数は、現行のロード・モジュールの WSA (書き込み可能静的区域) に配置されます。WSA 内での

変数のオフセットは、MAP オプションを指定した出力から検出することができ、WSA は CAA という言語環境プログラム制御ブロックの中に保持されます。WSA の値は、言語環境プログラム・ダンプにもリストされます。

ただし、NORENT オプションを指定してコードをコンパイルした場合、EXTERNAL STATIC は通常どおり (リンカー・リストおよびコンパイラの MAP オプションの出力を使用して) 検出されます。INTERNAL STATIC は、言語環境プログラム・ダンプの一部としてダンプされます (PLIDUMP が B オプションを指定して呼び出された場合)。

旧 PL/I コンパイラと異なり、STATIC のアドレスは、いずれか 1 つのレジスタを占有することはありません。

例えば、上記のプログラムで、変数を STATIC に変更したとします。

Compiler Source

```
Line.File
2.0      test: proc options(main);
3.0
4.0      dcl a fixed bin(31) static;
5.0      dcl b fixed bin(31) static;
6.0
7.0      on error
8.0          begin;
9.0          call plidump('TFBC');
10.0         end;
11.0
12.0      a = 0;
13.0      b = 29;
14.0      b = 17 / a;
```

NORENT オプションを指定してコンパイルすると、このプログラムに対するコンパイラ MAP オプションの結果は、実際には右側にもう 1 つ列があること、列の間のスペースはもっと離れていることを除き、以下のようになります。

```
***** STORAGE OFFSET LISTING *****
IDENTIFIER DEFINITION ATTRIBUTES

A          1-0:4      Class = static, Location = 0 : 0x0 + CSECT ***TEST2
B          1-0:5      Class = static, Location = 4 : 0x4 + CSECT ***TEST2
```

したがって、A はコンパイル単位 TEST の静的 CSECT の中の 16 進オフセット 00 の位置にあり、B は 16 進オフセット 04 の位置にあることになります。

このプログラムでは PLIDUMP が B オプションを指定して呼び出されているため、現行の呼び出しチェーン内のそれぞれのコンパイルごとに、静的ストレージの 16 進ダンプが含まれています。これは、以下のようになります(これも、右側の列が省略されています)。

```
Static for procedure TEST      Timestamp: 2004.08.12
+000000 0FC00AA0 00000000 0000001D 0FC00DC8 0FC00AC0
+000020 0FC00AC0 0FC00AA8 00444042 00A3AE01 0FC009C8
+000040 0FC00AE0 6E3BFFE0 00000000 00000000 00000000
+000060 0FC00B00 00000000 00000000 00000000 00000000
+000080 0AD963C8 10000000 0AD96188 00000000 00000000
```

したがって、16 進オフセット 00 にある A は (予想通りの) 16 進値 00000000 を持ち、16 進オフセット 04 にある B は (これも予想通りの) 16 進値 0000001D、つまり 10 進値 29 を持つことになります。

CONTROLLED 変数の検出

CONTROLLED 変数は本質的に LIFO スタックであり、それぞれの CONTROLLED 変数は、そのスタックの先頭を指す「アンカー」を持っています。CONTROLLED 変数を検出する秘訣はこのアンカーの位置を検出することであり、下記のように、その位置はコンパイラー・オプションによって異なります。

CONTROLLED 変数に関するこの説明の残りの部分では、プログラム・ソースは上記と同じプログラムですが、ストレージ・クラスが CONTROLLED に変更されています。

Compiler Source

```
Line.File
  2.0      test: proc options(main);
  3.0
  4.0      dcl a fixed bin(31) controlled;
  5.0      dcl b fixed bin(31) controlled;
  6.0
  7.0      on error
  8.0      begin;
  9.0      call plidump('TFBHC');
10.0      end;
11.0
12.0      allocate a, b;
13.0      a = 0;
14.0      b = 29;
15.0      b = 17 / a;
```

NORENT WRITABLE を指定した場合

コンパイラー MAP オプションの結果は、この場合も、実際には右側にもう 1 つ列があること、列の間のスペースはもっと離れていることを除き、以下のようになります。

```
* * * * * S T O R A G E   O F F S E T   L I S T I N G   * * * * *
IDENTIFIER DEFINITION ATTRIBUTES

A          1-0:4      Class = static, Location = 8 : 0x8 + CSECT ***TEST2
B          1-0:5      Class = static, Location = 12 : 0xC + CSECT ***TEST2
```

注: これらの行は、(A および B それ自体の場所ではなく) A および B のアンカーの場所を記述しています。したがって、A のアンカーはコンパイル単位 TEST の静的 CSECT 中の 16 進オフセット 08 の位置にあり、B のアンカーは 16 進オフセット 0C の位置にあることになります。

PLIDUMP が B オプションを指定して呼び出されている場合、現行の呼び出しチェーン内のそれぞれのコンパイルごとに、静的ストレージの 16 進ダンプが含まれています。これは、以下のようになります (これも、右側の列が省略されています)。

Static for procedure TEST Timestamp: . . .

```
+000000 0FC00A88 0FC00DB0 0FC00AA8 102B8A30 102B8A50
+000020 0FC00AA8 0FC00A88 00444042 00A3AE01 0FC009B0
+000040 0FC00AC8 6E3BFFE0 00000000 00000000 00000000
```

したがって、A のアンカーは 102B8A30 にあり、B のアンカーは 102B8A50 にあることになります。しかし、CONTROLLED 変数があるため、これらのストレージは ALLOCATE ステートメントを使用して取得されており、そのため、これらのアドレスはヒープ・ストレージ内を指しています。しかし、PLIDUMP が H オプシ

ンを指定して呼び出されている場合は、ヒープ・ストレージの 16 進ダンプが含まれることになります。これは、以下ようになります (これも、右側の列が省略されています)。

```
Enclave Storage:
Initial (User) Heap
+000000 102B7018 C8C1D5C3 0FC0F990 0FC0F990 00000000
. . .
+001A00 102B8A18 102B7018 00000020 0FC00A90 00000014
00000000 00000000 00000000 00000000
+001A20 102B8A38 102B7018 00000020 0FC00A94 00000014
00000000 00000000 0000001D 00000000
```

A のアンカーが 102B8A30 にあったので、A は 16 進 00000000 を持ち、B のアンカーが 102B8A50 にあったので、B は (予想通りの) 16 進 0000001D を持つことになります。

NORENT NOWRITABLE(FWS) を指定した場合

これらのオプションを指定したコンパイラー MAP オプションの結果は、この場合も、実際には右側にもう 1 つ列があること、列の間のスペースはもっと離れていることを除き、以下ようになります。

```
***** STORAGE OFFSET LISTING *****
IDENTIFIER DEFINITION ATTRIBUTES

A          1-0:4      Class = automatic, Location = 236 : 0xEC(r13)
B          1-0:5      Class = automatic, Location = 240 : 0xF0(r13)
```

注: これらのオプションを指定した場合、CONTROLLED 変数の位置を指定するために、追加の間接指示のレベルがあり、そのため、上記の行は A および B のアンカーのアドレスの位置を記述しています。したがって、A のアンカーのアドレスはブロック TEST の AUTOMATIC 内の 16 進 EC の位置にあり、一方 B のアンカーは 16 進 F0 にあることになります。

PLIDUMP が B オプションを指定して呼び出されているため、現行の呼び出しチェーン内のそれぞれのブロックごとに、自動ストレージの 16 進ダンプが含まれています。これは、以下ようになります (これも、右側の列が省略されています)。

```
Dynamic save area (TEST): 102973C8
+000000 102973C8 10000000 10297188 00000000 8FC007DA
. . .
+0000E0 102974A8 0FC00998 00000000 00000000 102B8A40
102B8A28 10297030 102977D0 8FDF3D7E
```

したがって、A のアンカーのアドレスは 102B8A40 になり、B のアンカーのアドレスは 102B8A28 になります。

PLIDUMP が H オプションも指定して呼び出されているので、ヒープ・ストレージの 16 進ダンプが含まれることになります。これは、以下ようになります (これも、右側の列が省略されています)。

```
Enclave Storage:
Initial (User) Heap
+000000 102B7018 C8C1D5C3 0FC0F990 0FC0F990 00000000
. . .
+001A00 102B8A18 102B7018 00000018 00000000 0FC00A78
102B8A80 00000000 102B7018 00000018
+001A20 102B8A38 102B8A20 0FC00A74 102B8A60 00000000
102B7018 00000020 102B8A40 00000014
```

```

+001A40 102B8A58 00000000 00000000 00000000 00000000
               102B7018 00000020 102B8A28 00000014
+001A60 102B8A78 00000000 00000000 0000001D 00000000
               00000000 00000000 00000000 00000000

```

B のアンカーのアドレスが 102B8A28 にあり、B のアンカーが 102B8A80 にある
ので、B は、予想通り 16 進 0000001D、つまり 10 進 29 になります。

NORENT NOWRITABLE(PRV) を指定した場合

これらのオプションを指定してコンパイルした場合の MAP リストは、以下のよう
になります。

```

* * * * * S T O R A G E   O F F S E T   L I S T I N G   * * * * *
IDENTIFIER DEFINITION ATTRIBUTES

***TEST3      1-0:4  Class = ext def,  Location = CSECT ***TEST3
***TEST4      1-0:5  Class = ext def,  Location = CSECT ***TEST4
_PRV_OFFSETS  1-0:1  Class = static,   Location = 8 : 0x8 + CSECT ***TEST2

```

ここでのキーは、この出力の最後の行です。 PRV_OFFSETS は、それぞれの
CONTROLLED 変数についての PRV テーブル内でのオフセットを保持する静的テ
ーブルです。この静的テーブルは、MAP オプションが指定された場合にのみ生成さ
れます。

このテーブルを解釈するために、コンパイラーは、ブロック名テーブルのすぐ後
に、通常は小さな別のリストも生成します。このリストは、このサンプルでは、以
下のようになります。

PRV Offsets

Number	Offset	Name
1	8	A
1	C	B

このテーブルには、名前が示された CONTROLLED 変数ごとに、ランタイム
_PRV_OFFSETS テーブルでの 16 進オフセットがリストされます。ブロック番号
(最初の列) を使用して、同じ名前であるが、異なるブロックに宣言されている変数
を区別することができます。

_PRV_OFFSETS テーブルは静的ストレージ内 (16 進オフセット 8) にあり、
PLIDUMP が B オプションを指定して呼び出されたため、以下に示すようなダンプ
出力が表示されます。

```

Static for procedure TEST      Timestamp: . . .
+000000 10908EC8 02020240 00000005 6DD7D9E5 6DD6C6C6
               00000000 00000004 D00000A0 00100000
+000020 10908EE8 6E3BFFE0 00000000 00000000 00000000
               00000000 90010000 00000000 00000000

```

したがって、PRV テーブル内の A のオフセットは 0 になり、PRV テーブル内の
B のオフセットは 4 になります。_PRV_OFFSETS テーブルの最初の 8 バイトを占
めている目印「_PRV_OFF」にも注目してください。

PRV テーブルは、常に CAA 内のオフセット 4 の位置にあり、PLIDUMP が H オ
プションを指定して呼び出されたため、ダンプ出力に表示され、以下のようにな
ります。

```
Control Blocks Associated with the Thread:
CAA: 0A7107D0
+000000 0A7107D0 00000800 0ADB7DE0 0AD97018 0ADB7018
00000000 00000000 00000000 00000000
```

したがって、PRV テーブルのアドレスは 0ADB7DE0 になり、ヒープ・ストレージ内のダンプ出力にも表示されます。

```
Enclave Storage:
Initial (User) Heap
+000000 102B7018 C8C1D5C3 0FC0F990 0FC0F990 00000000
. . .
+000DC0 0ADB7DD8 00000000 00000000 0ADB8A38 0ADB8A58
0ADB7018 00000488 00000000 00000000
```

したがって、PRV テーブルには 0ADB8A38 0ADB8A58 などが入っており、_PRV_OFFSETS テーブルから得られたとおり、PRV テーブル内での A のオフセットは 0 になり、B のオフセットは 4 になります。これらは、それぞれ A と B のアドレスでもあります。

これらのアドレスは、ダンプ内のヒープ・ストレージにも表示されます。

```
Enclave Storage:
Initial (User) Heap
+000000 102B7018 C8C1D5C3 0FC0F990 0FC0F990 00000000
. . .
+001A00 0ADB8A18 00000000 00000000 0ADB7018 00000020
00000000 00000014 0A7107D4 00000000
+001A20 0ADB8A38 00000000 00000000 0ADB7018 00000020
00000004 00000014 0A7107D4 00000000
+001A40 0ADB8A58 0000001D 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

したがって、A のアドレスは 0ADB8A38 で、その 16 進は予想通りの 00000000 になり、B のアドレスが 0ADB8A58 なので、その 16 進値も予想通りの 0000001D になります。

保存されたコンパイル・データ

コンパイルの間、コンパイラーはロード・モジュールにコンパイルに関するさまざまな情報を保存します。この情報は、デバッグや将来のマイグレーション作業のときに大きな助けになることがあります。この章では、この保存された情報について述べます。

タイム・スタンプ

コンパイラーは各ロード・モジュールに「タイム・スタンプ」を保存します。これは、"YYYYMMDDHHMISSVNRNML" の形式の 20 バイト文字ストリングで、コンパイルの日時、およびこのストリングを作成したコンパイラーのバージョンを記録します。ストリングの要素の意味は次のとおりです。

YYYY

コンパイルされた年

MM

コンパイルされた月

DD	コンパイルされた日
HH	コンパイルされた時間
MI	コンパイルされた分
SS	コンパイルされた秒
VN	コンパイラーのバージョン番号
RN	コンパイラーのリリース番号
ML	コンパイラーの保守レベル

タイム・スタンプは、PPA2 で見付けることができます。PPA2 の中のオフセット 12 は 4 バイト整数で、PPA2 のアドレスからタイム・スタンプにオフセット (負の数のことがあります) を与えます。

次に、PPA2 は、PPA1 で見付けることができます。PPA1 の中のオフセット 4 は、4 バイト整数で、PPA1 に対応するエントリー・ポイント・アドレスから PPA2 にオフセット (負の数のことがあります) を与えます。

次に、PPA1 は、ブロックのエントリー・ポイント・アドレスで見付けることができます。エントリー・ポイント・アドレスからのオフセット 12 は、4 バイト整数で、そのエントリー・ポイント・アドレスから PPA1 にオフセット (負の数のことがあります) を与えます。

保存されたオプション・ストリング

コンパイラーも、ロード・モジュールに 32 バイトの「ストリング」を保管し、ロード・モジュールのビルドに使用されるコンパイラー・オプションを記録します。

保存されたオプション・ストリングの PL/I 宣言は、449 ページの図 101 にあります。構造体の中のほとんどのフィールドにおいて、そのフィールドの意味は名前によって明白ですが、いくつかのフィールドについてはいくらかの説明が必要です。

- `sos_words` は、構造体のバイト数を 4 で割った数値を保持します。
- `sos_version` は、この構造体のバージョン番号です。コンパイラーのバージョン番号ではありません。
- 構造体のサイズ、およびバージョン番号によってどのフィールドが設定されているか。

```

dcl
1 sos based
,2 sos_words          fixed bin(8) unsigned
,2 sos_version        fixed bin(8) unsigned
,2 sos_arch           fixed bin(8) unsigned
,2 sos_tune           fixed bin(8) unsigned
,2 sos_currency       char(1)
,2 sos_optlevel       bit(4) /* set with version >= 2 */
,2 sos_scheduler      bit(1) /* set with version >= 5 */
,2 sos_nowritable_prv  bit(1) /* set with version >= 4 */
,2 sos_noblockedio    bit(1) /* set with version >= 3 */
,2 sos_optimize       bit(1)
,2 sos_window         fixed bin(15)
,2 sos_codepage       fixed bin(31)
,2 sos_limits_intname  fixed bin(8) unsigned
,2 sos_limits_extname  fixed bin(8) unsigned
,2 sos_limits_fixbinp1 fixed bin(8) unsigned
,2 sos_limits_fixbinp2 fixed bin(8) unsigned
,2 sos_limits_fixdec1  fixed bin(8) unsigned
,2 sos_limits_fixdec2 /* set with version >= 4 */
                     fixed bin(8) unsigned

```

図 101. 保存されたオプション・ストリングの宣言 (1/3)

```

,2 sos_flags1
,3 sos_check_stg      bit(1)
,3 sos_compact        bit(1)
,3 sos_csect          bit(1)
,3 sos_dbcs            bit(1)
,3 sos_display_wto     bit(1)
,3 sos_extrn_full      bit(1)
,3 sos_graphic         bit(1)
,3 sos_check_conform   bit(1) /* set with version >= 6 */
,2 sos_flags2
,3 sos_interrupt       bit(1)
,3 sos_reduce          bit(1)
,3 sos_norent          bit(1)
,3 sos_respect_date    bit(1)
,3 sos_rules_ans       bit(1)
,3 sos_stdsys          bit(1)
,3 sos_nowritable      bit(1)
,3 sos_wchar_big       bit(1)
,2 sos_flags3
,3 sos_cmpat           bit(4)
,3 sos_system          bit(4)
,2 sos_flags4
,3 sos_dllinit         bit(1)
,3 sos_xinfo_def       bit(1)
,3 sos_xinfo_xml       bit(1)
,3 sos_static_full     bit(1)
,3 sos_backreg_5       bit(1)
,3 sos_noresexp        bit(1) /* set with version >= 2 */
,3 sos_bifprec         bit(2) /* set with version >= 2 */
                          /* 01 => bifprec(15) */
                          /* 10 => bifprec(31) */
,2 sos_test
,3 sos_test_hooks      bit(4)
,3 sos_test_sym        bit(1)
,3 sos_test_nohook     bit(1) /* set with version >= 5 */
,3 sos_test_separate   bit(1) /* set with version >= 7 */
,3 sos_Static_Length   bit(1) /* set with version >= 2 */
,2 sos_float
,3 sos_afp             bit(1)
,3 sos_dft_nobinlarg   bit(1) /* set with version >= 7 */
,3 sos_dec_forcedsign   bit(1) /* set with version >= 6 */
,3 sos_dec_nofoflonasgn bit(1) /* set with version >= 6 */
,3 sos_prectype        bit(2) /* set with version >= 5 */
,3 sos_floatinmath     bit(2) /* set with version >= 2 */
,2 sos_usage
,3 sos_ans_round       bit(1)
,3 sos_ans_unspec      bit(1)
,3 sos_common          bit(1) /* set with version >= 6 */
,3 sos_initauto        bit(1) /* set with version >= 5 */
,3 sos_initbased       bit(1) /* set with version >= 5 */
,3 sos_initctl         bit(1) /* set with version >= 5 */
,3 sos_initstatic      bit(1) /* set with version >= 5 */
,3 sos_stringofg_is_c  bit(1) /* set with version >= 5 */

```

図 101. 保存されたオプション・ストリングの宣言 (2/3)

```

,2 sos_default
,3 sos_ans          bit(1)
,3 sos_asgn         bit(1)
,3 sos_byaddr       bit(1)
,3 sos_conn         bit(1)
,3 sos_descriptor   bit(1)
,3 sos_ebcdic       bit(1)
,3 sos_nonnative    bit(1)
,3 sos_nonnativeaddr bit(1)
,3 sos_inline       bit(1)
,3 sos_reorder      bit(1)
,3 sos_evendec      bit(1)
,3 sos_null370      bit(1)
,3 sos_recursive    bit(1)
,3 sos_descctr      bit(1)
,3 sos_ret_byaddr   bit(1)
,3 sos_initfill     bit(1)
,3 sos_initfill_char char(1)
,3 sos_short_ieee   bit(1)
,3 sos_dummy_unal   bit(1)
,3 sos_retcode      bit(1)
,3 sos_unaligned    bit(1)
,3 sos_ordinal_max  bit(1)
,3 sos_overlap      bit(1)
,3 sos_hex          bit(1)
,3 sos_e_hex        bit(1)
,3 sos_linkage      fixed bin(8) unsigned
,2 sos_prefix
,3 sos_size         bit(1)
,3 sos_stringrange  bit(1)
,3 sos_stringsize   bit(1)
,3 sos_subrg        bit(1)
,3 sos_fofl         bit(1)
,3 sos_ofl          bit(1)
,3 sos_invalidop    bit(1)
,3 sos_ufl          bit(1)
,3 sos_zdiv         bit(1)
,3 sos_conv         bit(1)
,3 *               bit(1)
,3 sos_dfp          bit(1) /* set with version >= 9 */
,3 sos_nosepname    bit(1) /* set with version >= 8 */
,3 sos_csectcut     bit(3) /* set with version >= 5 */
,2 sos_extension01
,3 sos_hgpr         bit(1) /* set with version >= 10 */
,3 sos_hgpr_preserve bit(1) /* set with version >= 10 */
,3 sos_goff         bit(1) /* set with version >= 10 */
,3 sos_dec_foflonmult bit(1) /* set with version >= 10 */
,3 sos_usage_hex_cstg bit(1) /* set with version >= 10 */
,3 sos_usage_substr_loose /* set with version >= 10 */
                        bit(1)
,3 *               bit(10)
,3 sos_cuname_offset fixed bin(16) unsigned

```

図 101. 保存されたオプション・ストリングの宣言 (3/3)

sos_cmpat フィールドにおいて考えられる値が、これらの宣言によって与えられます。

```

dcl sos_cmpat_le      bit(4) value('0000'b);
dcl sos_cmpat_v1     bit(4) value('0001'b);
dcl sos_cmpat_v2     bit(4) value('0010'b);
dcl sos_cmpat_v3     bit(4) value('0011'b);

```

sos_system フィールドにおいて考えられる値が、これらの宣言によって与えられます。

```

dc1 sos_system_mvs          bit(4) value('0001'b);
dc1 sos_system_tso          bit(4) value('0010'b);
dc1 sos_system_cics          bit(4) value('0011'b);
dc1 sos_system_ims           bit(4) value('0100'b);
dc1 sos_system_os            bit(4) value('0101'b);

```

sos_test_hooks フィールドにおいて考えられる値が、これらの宣言によって与えられます。

```

dc1 sos_test_hooks_none     bit(4) value('0000'b);
dc1 sos_test_hooks_block    bit(4) value('0001'b);
dc1 sos_test_hooks_stmt     bit(4) value('0011'b);
dc1 sos_test_hooks_path     bit(4) value('0101'b);
dc1 sos_test_hooks_all      bit(4) value('0111'b);

```

sos_linkage フィールドにおいて考えられる値が、これらの宣言によって与えられます。

```

dc1 sos_linkage_optlink     fixed bin(8) unsigned value(1);
dc1 sos_linkage_system      fixed bin(8) unsigned value(2);

```

sos_bifprec フィールドにおいて考えられる値が、これらの宣言によって与えられます。

```

dc1 sos_bifprec_15          bit(2) value('01'b);
dc1 sos_bifprec_31          bit(2) value('10'b);

```

sos_floatinmath フィールドにおいて考えられる値が、これらの宣言によって与えられます。

```

dc1 sos_floatinmath_asis    bit(2) value('00'b);
dc1 sos_floatinmath_long    bit(2) value('10'b);
dc1 sos_floatinmath_extnnd  bit(2) value('11'b);

```

保存されたオプション・ストリングは、次の 2 つのどちらかの方法でタイム・スタンプの後に位置指定されます。

1. サービス・オプションが指定されている場合、サービス・オプションの中に指定されたストリングは、文字がストリングを変更すると即時にタイム・スタンプに続きます。次に、保存されたオプション・ストリングが、2 番目の文字がストリングを変更するとサービス・ストリングに続きます。
2. サービス・オプションが指定されていない場合、保存されたオプション・ストリングは、文字がストリングを変更すると即時にタイム・スタンプに続きます。

保存されたオプション・ストリングを保持する可変ストリングの長さは、保存されたオプション・ストリングそれ自体のサイズより長くなります。

サービス・ストリングの存在 (または不在) は、PPA2 内に、PPA2 の 10 進数のオフセット 20 にフラグ・バイトで示されます。このバイトとボックス 20 の AND 演算結果がゼロでなければ、サービス・ストリングが存在します。

以前にリリースされた PL/I コンパイラーには、保存されたオプション・ストリングを、コンパイラーがロード・モジュールに置かないものがありました。保存されたオプション・ストリングの存在 (または不在) は、PPA2 内に、10 進数のオフセット 20 にフラグ・バイトで示されます。このバイトとボックス 02 の AND 演算結果がゼロでなければ、保存されたオプション・ストリングが存在します。

第 18 章 割り込みとアテンションの処理

PL/I プログラムにアテンション割り込みを認識させるには、次の 2 つの操作が可能でなければなりません。

- 割り込みを作成できなければなりません。それを行う方法は、使用する端末とオペレーティング・システムによって異なります。
- ユーザーのプログラムで、割り込みに対応する準備が整っていないければなりません。ATTENTION 条件が生じたときにユーザーのプログラムが制御を得るようにするために、ユーザーのプログラム内に ON ATTENTION ステートメントを作成しておくことができます。

注: プログラムに、呼び出したい ATTENTION ON ユニットがある場合は、次のオプションのどちらかを用いてプログラムをコンパイルしなければなりません。

- INTERRUPT オプション (TSO でのみサポートされる)
- NOTEST または TEST(NONE,NOSYM) 以外の TEST オプション

この方法でコンパイルすると、PLIXOPT で INTERRUPT(OFF) を明示的に指定していない限り、INTERRUPT(ON) が有効になります。

割り込みを作成する手順は、使用しているオペレーティング・システムと端末に関する IBM マニュアルに記載されています。

割り込み (オペレーティング・システムがユーザーの要求を認識すること) と、ATTENTION 条件の発生とは異なります。

割り込み とは、オペレーティング・システムが実行中のプログラムに通知するユーザーの要求のことです。INTERRUPT コンパイル時オプションが指定されて PL/I プログラムがコンパイルされた場合は、プログラム内の個別の位置に内部割り込みスイッチをテストする命令が組み込まれます。この内部割り込みスイッチは、ロード・モジュール内の任意のプログラムが INTERRUPT コンパイル時オプションを指定してコンパイルされた場合に設定することができます。

内部スイッチは、割り込み要求が出されたことをオペレーティング・システムが認識した時に設定されます。特殊テスト命令 (ポーリング) の実行によって、ATTENTION 条件が発生します。ポーリングが起きる前にデバッグ・ツール・フック (つまり CALL PLITEST) が検出されると、ATTENTION 条件の処理が始まる前にデバッグ・ツールに制御が与えられるようにすることができます。

ポーリングにより、ATTENTION 条件は、PL/I ステートメントの内部ではなく、ステートメントとステートメントの間で発生します。

454 ページの図 102 に、骨組みプログラム、ATTENTION ON ユニット、およびポーリング命令が生成されるいくつかの状況を示してあります。プログラム内では、次の位置でポーリングが発生します。

- LABEL1
- DO の各反復

- ELSE PUT SKIP ... ステートメント
- ブロックの END ステートメント

```
%PROCESS INTERRUPT;
.
.
.
ON ATTENTION
BEGIN;
  DCL X FIXED BINARY(15);
  PUT SKIP LIST ('Enter 1 to terminate, 0 to continue. ');
  GET SKIP LIST (X);
  IF X = 1 THEN
    STOP;
  ELSE
    PUT SKIP LIST ('Attention was ignored');
END;
.
.
.
LABEL1:
  IF EMPNO ...
  .
  .
DO I = 1 TO 10;
  .
  .
  .
END;
.
.
.
```

図 102. ATTENTION ON ユニットの使用

ATTENTION ON ユニットの使用

ATTENTION ON ユニット内の処理を使って、プログラム内の潜在的なエンドレス・ループを終了させることができます。

ATTENTION ON ユニットに制御が与えられるのは、割り込みが発生したことをポーリング命令が認識したときです。通常、ON ユニットからの戻りは、ポーリング・コードに続くステートメントに対して行われます。

デバッグ・ツールとの対話

プログラム内で TEST(ALL) または TEST(ERROR) ランタイム・オプションが有効であるときに割り込みが生じると、次にフックが検出されたときに、デバッグ・ツールが制御を受け取ることになります。これは、割り込みが発生したことをプログラムのポーリング・コードが認識する前である可能性があります。

あとから ATTENTION 条件が発生すると、デバッグ・ツールは条件の処理のために再び制御を受け取ります。

第 19 章 チェックポイント/再始動機能の使用

この章では、バッチ環境で長時間稼働するプログラムの実行途中でチェックポイントをとるために役立つ手段である PL/I チェックポイント/再始動機能について説明します。

プログラム内の指定されたポイントで、プログラムの現状についての情報が、データ・セットにレコードとして書き込まれます。システム障害が原因でプログラムが終了したときに、この情報を使えば、プログラムを全部実行し直す必要がなく、障害の発生地点の近くからプログラムを再始動することができます。

この再始動は、自動再始動または据え置き再始動のどちらもあり得ます。自動再始動は、即時に実行される再始動です (ただし、システム・メッセージで要求されたときに、オペレーターがそれを許可することを前提とします)。据え置き再始動は、新しいジョブとして、あとから実行される再始動です。

システム障害が起きていなくても、プログラム内から自動再始動を要求することができます。

PL/I チェックポイント/再始動は、オペレーティング・システムの拡張チェックポイント/再始動機能を使用します。この機能については、525 ページの『参考文献』に掲載したマニュアルを参照してください。

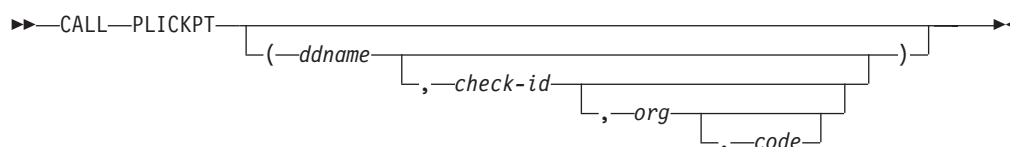
チェックポイント/再始動を使うには、次の操作を行う必要があります。

- ユーザーのプログラム内の適切なポイントで、チェックポイント・レコードを書き込むように要求します。これは、組み込みサブルーチン `PLICKPT` を使って行います。
- チェックポイント・レコードを書き込めるデータ・セットを提供します。
- また、必ず期待どおりに再始動アクティビティーが行われるように、`EXEC` ステートメントまたは `JOB` ステートメント内で `RD` パラメーターを指定しなければならないこともあります (「*z/OS JCL Reference*」を参照してください)。

注: プログラムで使用するデータ・セットに関連した制約事項に注意する必要があります。この制約事項の詳細については、525 ページの『参考文献』を参照してください。

チェックポイント・レコードの要求

チェックポイント・レコードを書かせたいときには、そのたびに、 PL/I プログラムから組み込みサブルーチン `PLICKPT` を呼び出す必要があります。



4 つの引数はすべてオプションです。引数を使わない場合は、所定の順序でその引数に続く別の引数を指定するとき以外は、引数を指定する必要はありません。引数を指定する場合は、未使用引数をヌル・ストリング (") として指定します。次に、引数について説明します。

ddname

文字ストリングの定数または変数で、チェックポイント・レコードに使用するデータ・セットを定義する DD ステートメントの名前を指定します。この引数を省略すると、システムはデフォルトの DD 名 SYSCHK を使用します。

check-id

あとからチェックポイント・レコードを識別できるようにするためにチェックポイント・レコードに割り当てる名前を指定する文字ストリングの定数または変数です。この引数を省略すると、システムは固有の ID を提供し、それをオペレーターのコンソールに印刷します。

org

値がオペレーティング・システム用語でチェックポイント・データ・セットの編成を示す属性 CHARACTER(2) を持つ文字ストリング定数または変数です。PS は順次 (すなわち、CONSECUTIVE) 編成を示し、PO は区分編成を表します。この引数を省略すると、PS がとられます。

code

PLICKPT から戻りコードを受け取ることができる、属性 FIXED BINARY (31) を持つ変数です。戻りコードには、次の値があります。

- 0 チェックポイントは正常にとられた。
- 4 再始動は正常に行われた。
- 8 チェックポイントはとられていない。 PLICKPT ステートメントを調べる必要があります。
- 12 チェックポイントはとられていない。欠落した DD ステートメント、ハードウェア・エラーがないか、またはデータ・セットのスペースが不十分でなかったかを調べてください。 REPLY オプションを指定した DISPLAY ステートメントが完了する前にチェックポイントをとろうとしても、失敗します。
- 16 チェックポイントはとられたが、ENQ マクロ呼び出しが未解決のまま、再始動時に復元されない。このような事態は、通常、PL/I プログラムの場合は生じません。

チェックポイント・データ・セットの定義

チェックポイント・レコードを入れる先のデータ・セットを定義するには、ジョブ制御プロシージャ内に DD ステートメントを入れなければなりません。このデータ・セットは、CONSECUTIVE 編成と区分編成のいずれでもかまいません。任意の有効 DD 名を使用することができます。DD 名 SYSCHK を使う場合は、PLICKPT を呼び出すときに DD 名を指定する必要はありません。

データ・セット名を指定しなければならないのは、据え置き再始動用にデータ・セットをとっておきたい場合だけです。入出力装置は、直接アクセス装置であれば使用できます。

最後のチェックポイント・レコードだけを取得するには、状況を NEW (または、データ・セットが既に存在する場合は OLD) と指定します。これにより、各チェックポイント・レコードが直前のレコードを上書きします。

複数のチェックポイント・レコードを保存するには、状況を MOD と指定します。これで、各チェックポイント・レコードが直前のレコードのあとに続けて追加されます。

チェックポイント・データ・セットがライブラリーであれば、メンバー名として『check-id』が使用されます。したがって、チェックポイントは、それ以前にとられた同一名を持つすべてのチェックポイントを削除します。

直接アクセス・ストレージの場合、保持しようとする最大数のチェックポイント・レコードを保管するのに十分な 1 次スペースを割り振らなければなりません。増分スペース割り振りを指定することはできますが、ここでは使用されません。チェックポイント・レコードはこのステップで割り振られる主記憶域の大きさより約 5000 バイト長くなっています。

DCB 情報は必要ありませんが、適切であれば次のどれでも組み込むことができます。

OPTCD=W, OPTCD=C, RECFM=UT

これらのサブパラメーターについては、「*z/OS JCL User's Guide*」に説明があります。

再始動の要求

再始動には、自動再始動または据え置き再始動があります。自動再始動にできるのは、システム障害のあとか、またはプログラム内から行われる場合です。システム・オペレーターは、システムから要求があれば、すべての自動再始動を許可しなければなりません。

システム障害後の自動再始動

チェックポイントを取り終わってからシステム障害が生じた場合、EXEC ステートメントまたは JOB ステートメント内で RD=R を指定していた (または RD パラメーターを省略していた) ならば、最後にとられたチェックポイントで自動再始動が行われます。

チェックポイントをまだとっていないときにシステム障害が生じた場合に、EXEC ステートメントまたは JOB ステートメント内で RD=R を指定していたのであれば、その場合にも自動再始動が、ジョブ・ステップの初めから行われます。

システム障害が起きてからでも、EXEC ステートメントまたは JOB ステートメント内で RD=RNC を指定すれば、ジョブ・ステップの初めから自動再始動を強制実行させることもできます。別のシステム障害が生じて、RD=RNC を指定すれば、チェックポイントを処理しないで自動ステップ再始動を要求することになります。

プログラム内の自動再始動

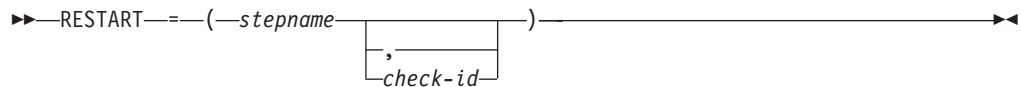
再始動は、プログラム内の任意の点で要求できます。この再始動に関する規則は、システム障害後の再始動の場合と同じです。再始動を要求するには、次のステートメントを実行する必要があります。

```
CALL PLIREST;
```

再始動を行うために、コンパイラーは、システム完了コード 4092 でプログラムを異常終了させます。したがって、この機能を使用するには、システム生成時に適格コードのテーブルからシステム完了コード 4092 を削除してはなりません。

据え置き再始動

自動再始動活動を取り消して、しかもそのチェックポイントを据え置き再始動に使えるように確保しておくには、プログラムの最初の実行時に、EXEC ステートメントまたは JOB ステートメント内で RD=NR を指定します。



あとから据え置き再始動が必要になった場合は、JOB ステートメントに RESTART パラメーターを指定して、そのプログラムを新しいジョブとして実行依頼する必要があります。RESTART パラメーターを使って、再始動を開始するジョブ・ステップを指定します。また、チェックポイントから再始動を行うにはチェックポイント・レコード名を指定します。

チェックポイントからの再始動の場合、チェックポイント・レコードが入ったデータ・セットを定義する DD ステートメントも提供する必要があります。この DD ステートメントには SYSCHK という名前であればなりません。DD ステートメントは、ジョブ・ステップの EXEC ステートメントの直前に挿入しなければなりません。

チェックポイント/再始動活動の変更

次のステートメントを実行すれば、プログラム内でとられている任意のチェックポイントからの自動再始動アクティビティーを取り消すことができます。

```
CALL PLICANC;
```

ただし、JOB ステートメントや EXEC ステートメント内で RD=R または RD=RNC を指定していたのであれば、やはり自動再始動がジョブ・ステップの先頭から行われます。

また、以前にとられているチェックポイントがあれば、それをそのまま据え置き再始動に使用することができます。

JOB ステートメントまたは EXEC ステートメント内で RD=NC を指定すれば、プログラム内で要求したものであっても、任意の自動再始動およびチェックポイントを取る予定を取り消すことができます。

第 20 章 ユーザー出口の用法

PL/I は、ユーザーの必要に合わせて PL/I 製品をカスタマイズできるようにするいくつかのユーザー出口を提供しています。PL/I 製品は、デフォルト出口と、関連するソース・ファイルを提供します。

デフォルト出口によって提供された関数とは異なる機能を出口に実行させたい場合には、提供されたソース・ファイルを適切に変更することをお勧めします。

場合によっては、ユーザーの組織の要件を満たすようにコンパイラーを調整できることが役立つこともあります。例えば、特定のメッセージを抑止したり、ほかのメッセージの重大度を変更したりする必要が生じることがあります。コンパイルについての統計情報のログをファイルに書き込むなど、各コンパイルごとに特定の機能を実行するように指定する必要が生じることがあります。コンパイラー・ユーザー出口は、この種の機能を処理します。

PL/I を使うと、独自のユーザー出口を作成することもできるし、必要に応じて、「そのまま使う」か、変更することもできる製品提供の出口を使うこともできます。製品提供のユーザー出口ソース・コードを、176 ページの図 16 に示しています。

この章の目的は、次の内容を説明することです。

- コンパイラー・ユーザー出口がサポートするプロシージャー
- コンパイラー・ユーザー出口を活動化する方法
- IBMUEXIT、IBM 提供のコンパイラー・ユーザー出口
- 独自のコンパイラー・ユーザー出口作成の要件

コンパイラー・ユーザー出口によって実行されるプロシージャー

コンパイラー・ユーザー出口は、次の 3 つの特定のプロシージャーを実行します。

- 初期化
- コンパイラー・メッセージのインターセプトとフィルター操作
- 終了

図 103 に示してあるように、コンパイラーは、初期化プロシージャー、メッセージ・フィルター・プロシージャー、および終了プロシージャーへ制御を渡します。これらの 3 つのプロシージャーは、それぞれ、要求されたプロシージャーが完了したならば、制御をコンパイラーへ戻します。

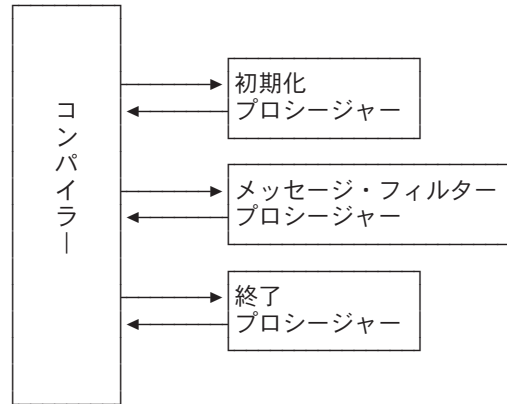


図 103. PL/I コンパイラ・ユーザー出口のプロシージャ

これらの各プロシージャには、次の 2 つの制御ブロックが渡されます。

- コンパイルについての情報が含まれているグローバル制御ブロック。これは、最初のパラメーターとして渡されます。グローバル制御ブロックの特定の情報については、462 ページの『グローバル制御ブロックの構造』を参照してください。
- 2 番目のパラメーターとして渡される機能専用の制御ブロック。この制御ブロックの内容は、どのプロシージャが呼び出されているかによって異なります。詳細については、463 ページの『初期化プロシージャの作成』、463 ページの『メッセージ・フィルター操作プロシージャの作成』、および 465 ページの『終了プロシージャの作成』を参照してください。

コンパイラ・ユーザー出口の活動化

コンパイラ・ユーザー出口を活動化するには、EXIT コンパイル時オプションを指定しなければなりません。EXIT オプションの詳細については、32 ページの『EXIT』を参照してください。

EXIT コンパイル時オプションを使うと、ユーザー出口入力ファイルの DD 名を指定するユーザー・オプション・STRING を指定できます。STRING を指定しない場合は、SYSUEXIT がユーザー出口入力ファイルの DD 名として使用されます。

ユーザー・オプション・STRING は、462 ページの『グローバル制御ブロックの構造』で説明するグローバル制御ブロック内のユーザー出口機能に渡されます。追加情報については、462 ページの『グローバル制御ブロックの構造』の「Uex_UIB_User_char_str」フィールドの説明を参照してください。

IBM 提供のコンパイラ出口、IBMUEXIT

IBM は、ユーザーに代わってメッセージのフィルター操作を行う、サンプルのコンパイラ・ユーザー出口 IBMUEXIT を提供しています。このユーザー出口は、メッセージをモニターし、指定されたメッセージ番号に基づいて、メッセージを抑止したり、メッセージの重大度を変更したりします。

コンパイラー・ユーザー出口のカスタマイズ

前述したように、独自のコンパイラー・ユーザー出口を作成することも、単にコンパイラーに付属の出口を使用することもできます。いずれの場合も、コンパイラー・ユーザー出口用のフェッチ可能ファイルの名前は **IBMUEXIT** でなければなりません。

このセクションでは、次の方法について説明します。

- カスタマイズされたメッセージ・フィルター操作用にユーザー出口入力ファイルを変更する
- 独自のコンパイラー・ユーザー出口を作成する

SYSUEXIT の変更

まったく新しいコンパイラー・ユーザー出口を作成するのではなく、簡単にユーザー出口入力ファイルを変更できます。

ファイルを編集して、抑止するメッセージ番号と、変更するメッセージ番号の重大度レベルを指示します。図 104 に、サンプル・ファイルを示します。

Fac Id	Msg No	Severity	Suppress	Comment
'IBM'	1042	-1	1	String spans multiple lines
'IBM'	1044	-1	1	FIXED BIN 7 mapped to 1 byte
'IBM'	1047	8	0	Order inhibits optimization
'IBM'	1052	-1	1	Nodescriptor with * extent arg
'IBM'	1059	0	0	Select without OTHERWISE
'IBM'	1169	0	1	Precision of result determined

図 104. ユーザー出口入力ファイルの例

最初の 2 行はヘッダー行で、**IBMUEXIT** では無視されます。残りの行には、可変数のブランクで区切られた入力データが含まれています。

ファイルの各列は、コンパイラー・ユーザー出口に次のように関係しています。

- 出口を適用するコンパイラー・メッセージすべてに対して、最初の列に単一引用符で囲んだ **'IBM'** を指定する必要があります。SQL プリプロセッサの SQL 側から出されるメッセージの場合は、SQL メッセージ接頭語 **'SQL'** (やはり単一引用符で囲む) を指定する必要があります。
- 2 番目の列は、4 桁のメッセージ番号です。
- 3 番目の列は、新しいメッセージ重大度です。重大度 **-1** は、重大度がデフォルト値のままにすることを表します。
- 4 番目の列は、メッセージを抑止するかどうかを示します。「1」はメッセージが抑止されることを示し、「0」はメッセージが出力されることを示します。
- 最後の列はコメント欄で、通知の目的の列であり、**IBMUEXIT** では無視されません。

独自のコンパイラー出口の作成

独自のユーザー出口を作成するには、モデルとして **IBMUEXIT** を使用できます (図 16 にあるソースを参照)。出口を作成するときには、初期化、メッセージのフィルター操作、および終了をそれぞれ必ずカバーしてください。

そのセクションでも述べているように、コンパイラー・ユーザー出口は RENT オプションを指定してコンパイルし、DLL としてリンクする必要があります。

グローバル制御ブロックの構造

グローバル制御ブロックは、3 つのユーザー出口プロシージャ（初期化、フィルター操作、および終了）が呼び出されるたびに、それぞれに渡されます。次のコードと説明は、グローバル制御ブロック内の各フィールドの内容を示すものです。

```
Dcl
1 Uex_UIB          native based( null() ),
2 Uex_UIB_Length   fixed bin(31),

2 Uex_UIB_Exit_token    pointer,          /* for user exit's use */

2 Uex_UIB_User_char_str pointer,          /* to exit option str */
2 Uex_UIB_User_char_len fixed bin(31),

2 Uex_UIB_Filename_str  pointer,          /* to source filename */
2 Uex_UIB_Filename_len  fixed bin(31),

2 Uex_UIB_return_code fixed bin(31),      /* set by exit procs */
2 Uex_UIB_reason_code fixed bin(31),      /* set by exit procs */

2 Uex_UIB_Exit_Routs,                    /* exit entries set at
                                         initialization */

3 ( Uex_UIB_Termination,
    Uex_UIB_Message_Filter,              /* call for each msg */
    *, *, *, * )
    limited entry (
        *,                                /* to Uex_UIB */
        *,                                /* to a request area */
    );
```

データ入力フィールド

- **Uex_UIB_Length:** バイト単位の制御ブロックの長さ。値は storage (Uex_UIB) です。
- **Uex_UIB_Exit_token:** ユーザー出口プロシージャによって使われる。例えば、初期化では、メッセージ・フィルター・プロシージャと終了プロシージャの両方によって使われるデータ構造に設定できます。
- **Uex_UIB_User_char_str:** オプショナルの文字ストリングを指す（ただし、指定した場合のみ）。例えば、pli filename (EXIT ('string'))...fn では、31 文字までの文字ストリングにすることができます。
- **Uex_UIB_char_len:** User_char_str が指すストリングの長さ。この値はコンパイラーによって設定されます。
- **Uex_UIB_Filename_str:** コンパイルするソース・ファイルの名前で、ファイル名のほかにドライブとサブディレクトリーも含まれます。この値はコンパイラーによって設定されます。
- **Uex_UIB_Filename_len:** Filename_str が指すソース・ファイルの名前の長さ。この値はコンパイラーによって設定されます。
- **Uex_UIB_return_code:** ユーザー出口プロシージャからの戻りコード。この値はユーザーによって設定されます。
- **Uex_UIB_reason_code:** プロシージャ理由コード。この値はユーザーによって設定されます。

- **Uex_UIB_Exit_Routs:** 初期化プロシージャが設定する出口項目。
- **Uex_UIB_Termination:** 終了時にコンパイラーが呼び出す項目。この値はユーザーによって設定されます。
- **Uex_UIB_Message_Filter:** メッセージを生成する必要があるときにコンパイラーが呼び出す項目。この値はユーザーによって設定されます。

初期化プロシージャの作成

初期化プロシージャは、出口が必要とする初期化、例えば、ファイルのオープンやストレージの割り振りなどを実行する必要があります。初期化プロシージャ特有の制御ブロックは、次のようにコード化されます。

```
Dcl 1 Uex_ISA native based( null() ),
    2 Uex_ISA_Length_fixed bin(31); /* storage(Uex_ISA) * /
```

初期化プロシージャのグローバル制御ブロックの構文については、462 ページの『グローバル制御ブロックの構造』に説明があります。

初期化プロシージャの完了時には、戻りコード/理由コードを次のように設定する必要があります。

0/0

コンパイルを続行する

4/n

将来の利用のために予約済み

8/n

将来の利用のために予約済み

12/n

将来の利用のために予約済み

16/n

コンパイルを打ち切る

メッセージ・フィルター操作プロシージャの作成

メッセージ・フィルター操作プロシージャを使うと、メッセージを抑止したり、メッセージの重大度を変更したりすることができます。どのメッセージの重大度も高くすることはできますが、低くすることができるのは、**ERROR** (重大度コード 8) メッセージまたは、**WARNING** (重大度コード 4) メッセージだけです。

プロシージャ特有の制御ブロックには、メッセージについての情報が含まれています。これは、特定のメッセージの取り扱い方法を示す情報をコンパイラーに渡すために使われます。

プロシージャ特有のメッセージ・フィルター制御ブロックの例を次に示します。

```
Dcl 1 Uex_MFX native based( null() ),
    2 Uex_MFX_Length fixed bin(31),

    2 Uex_MFX_Facility_Id char(3), /* of component writing
                                     message */
    2 * char(1),
    2 Uex_MFX_Message_no fixed bin(31),
    2 Uex_MFX_Severity fixed bin(15),
    2 Uex_MFX_New_Severity fixed bin(15), /* set by exit proc */
```

```

2 Uex_MFX_Inserts          fixed bin(15),
2 Uex_MFX_Inserts_Data( 6 refer(Uex_MFX_Inserts) ),
3 Uex_MFX_Ins_Type         fixed bin(7),
3 Uex_MFX_Ins_Type_Data union unaligned,
4 *                         char(8),
4 Uex_MFX_Ins_Bin8         fixed bin(63),
4 Uex_MFX_Ins_Bin         fixed bin(31),
4 Uex_MFX_Ins_Str,
5 Uex_MFX_Ins_Str_Len      fixed bin(15),
5 Uex_MFX_Ins_Str_Addr     pointer,
4 Uex_MFX_Ins_Series,
5 Uex_MFX_Ins_Series_Sep   char(1),
5 Uex_MFX_Ins_Series_Addr  pointer;

```

データ入力フィールド

- **Uex_MFX_Length:** バイト単位の制御ブロックの長さ。値は storage (Uex_MFX) です。
- **Uex_MFX_Facility_Id:** 機能の ID。コンパイラーの場合、ID は IBM です。SQL プリプロセッサの SQL 側の場合、ID は SQL です。この値はコンパイラーによって設定されます。
- **Uex_MFX_Message_no:** コンパイラーが生成する予定のメッセージ番号。この値はコンパイラーによって設定されます。
- **Uex_MFX_Severity:** メッセージの重大度レベルで、長さは 1 から 15 文字まで。この値はコンパイラーによって設定されます。
- **Uex_MFX_New_Severity:** メッセージの新しい重大度レベルで、長さは 1 から 15 文字まで。この値はユーザーによって設定されます。
- **Uex_MFX_Inserts:** メッセージでの挿入の数。範囲はゼロから 6 までです。この値はコンパイラーによって設定されます。
- **Uex_MFX_Inserts_Data:** 各挿入が記述されたフィールド。これらの値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Type:** 挿入のタイプ。可能な挿入タイプは次のとおりです。
 - **Uex_Ins_Type_Xb31:** 整数タイプに使用され、値 1 を持ちます。
 - **Uex_Ins_Type_Char:** 整数タイプに使用され、値 2 を持ちます。
 - **Uex_Ins_Type_Series:** 整数タイプに使用され、値 3 を持ちます。
 - **Uex_Ins_Type_Xb63:** 整数タイプに使用され、値 4 を持ちます。
 この値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Bin:** 整数タイプの挿入の場合の整数値。この値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Str_Len:** 文字タイプの挿入の場合の長さ (バイト単位)。この値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Str_Addr:** 文字タイプの挿入の場合の文字ストリングのアドレス。この値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Series_Sep:** シリーズ・タイプの挿入の場合の各エレメント間に挿入される文字。通常は、ブランク、ピリオド、またはコンマです。この値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Series_Addr:** シリーズ・タイプの挿入の場合の、さまざまな文字ストリングのシリーズのアドレス。このアドレスは FIXED BIN(31) フィールド

を指します。このフィールドには、連結するストリングの数と、その後にそれらのストリングの各アドレスが保持されます。この値はコンパイラーによって設定されます。

メッセージ・フィルター操作プロシージャーの完了時に、戻りコード/理由コードを次のいずれかに設定してください。

- 0/0**
コンパイルを続行し、メッセージを出力する
- 0/1**
コンパイルを続行し、メッセージを出力しない
- 4/n**
将来の利用のために予約済み
- 8/n**
将来の利用のために予約済み
- 16/n**
コンパイルを打ち切る

終了プロシージャーの作成

ファイルのクローズのような、必要なクリーンアップを実行するには、終了プロシージャーを使います。また、エラー・メッセージ・フィルター・プロシージャーと初期化プロシージャーの実行時に収集される情報に基づいて、最終統計レポートを作成することもできます。

終了プロシージャー特有の制御ブロックは、次のようにコーディングされます。

```
Dcl 1 Uex_ISA native based,  
    2 Uex_ISA_Length_fixed bin(31); /* storage(Uex_ISA)      */
```

終了プロシージャーのグローバル制御ブロック構文については、462 ページの『グローバル制御ブロックの構造』に説明があります。終了プロシージャーの完了時に、戻りコード/理由コードを次のいずれかに設定してください。

- 0/0**
コンパイルを続行する
- 4/n**
将来の利用のために予約済み
- 8/n**
将来の利用のために予約済み
- 12/n**
将来の利用のために予約済み
- 16/n**
コンパイルを打ち切る

第 21 章 PL/I 記述子

この章では、実行時の PL/I ルーチン間での PL/I パラメーターの引き渡しの規則について説明します。記述子以外の言語環境プログラムのランタイム環境に関する考慮事項の詳細については、「z/OS 言語環境プログラム プログラミング・ガイド」を参照してください。このマニュアルには、ランタイム環境の規則と、その規則をサポートするアセンブラー・マクロについての説明があります。

引数の引き渡し

ストリング、配列、または構造体が引数として渡されるときには、呼び出されたルーチンが `OPTIONS(NODESCRIPTOR)` を使って宣言されていない限り、コンパイラーがその引数の記述子を渡します。このような記述子を渡す方法として次の 2 つがあります。

- 記述子リストを使う
- 記述子ロケーターを使う

これらの 2 つの方法について、次のキー機能に注意してください。

- **引数が記述子リストを使って渡されるとき**
 - 渡される引数の数は、いずれかの引数に記述子が必要であるときには、指定された引数の数より 1 大きい数になります。
 - 記述子を使って渡される引数は、値 (BYVALUE) で渡されるポインターとして受け取ることができます。
 - コンパイラーは、`DEFAULT(DESCLIST)` コンパイラー・オプションが有効なときにこのメソッドを使用します。
- **引数が記述子ロケーターによって渡されるとき**
 - 渡される引数の数は常に、指定された引数の数と一致します。
 - 記述子を使って渡される引数は、アドレス (BYADDR) で渡されるポインターとして受け取ることができます。
 - コンパイラーは、`DEFAULT(DESCLOCATOR)` コンパイラー・オプションが有効なときにこのメソッドを使用します。

記述子リストによる引数の引き渡し

引数とその記述子が記述子リストを使って渡される場合は、1 つでも記述子を必要とする引数があれば、1 つの追加の引数が渡されます。この追加の引数は、ポインターのリストを指すポインターです。このリストの入力項目の数は、渡される引数の数と一致します。記述子が必要でない引数の場合は、記述子リストの中で、対応するポインターが `SYSNULL` に設定されます。記述子が必要である引数の場合は、記述子リストの中で、対応するポインターがその引数の記述子のアドレスに設定されます。

例えば、次のようにルーチン `sample` が宣言されたとします。

```
declare sample entry( fixed bin(31), varying char(*) )
                    options( byaddr descriptor );
```

さらに、`sample` が次のようなステートメントで呼び出されたとします。

```
call sample( 1, 'test' );
```

この場合、次の 3 つの引数がルーチンに渡されます。

- 値が 1 の固定 `bin(31)` 一時変数のアドレス
- 値が `test` の可変 `char(4)` 一時変数のアドレス
- 次の項目で構成される記述子リストのアドレス
 - `SYSNULL()`
 - 可変 `char(4)` ストリング用の記述子のアドレス

記述子ロケータによる引数の引き渡し

引数とその記述子が記述子ロケータによって渡される場合は、引数に記述子が必要であるときにはいつでも、その引数用のロケータ/記述子のアドレスが代わりに渡されます。

ストリングを除いて、ロケータ/記述子は、ポインターの対です。最初のポインターはデータのアドレスで、2 番目のポインターは記述子のアドレスです。ストリングの場合、`CMPAT(LE)` では、ロケータ/記述子は同じく、ポインターの対です。ただし、他の `CMPAT` オプションでは、ロケータ/記述子は、ストリングのアドレスおよびストリング記述子自体から成ります。

例えば、次のようにルーチン `sample` が再度宣言されたとします。

```
declare sample entry( fixed bin(31), varying char(*) )
options( byaddr descriptor );
```

さらに、`sample` が次のようなステートメントで呼び出されたとします。

```
call sample( 1, 'test' );
```

この場合、次の 2 つの引数がルーチンに渡されます。

- 値が 1 の固定 `bin(31)` 一時変数のアドレス
- 次の項目で構成されるロケータ/記述子のアドレス
 - 値が `test` の可変 `char(4)` 一時変数のアドレス
 - `CMPAT(LE)` では、可変 `char(4)` ストリングの `CMPAT(LE)` 記述子のアドレス
 - `CMPAT(V*)` では、可変 `char(4)` ストリングの `CMPAT(V*)` 記述子

CMPAT(V*) 記述子

LE 記述子とは異なり、`CMPAT(V*)` 記述子は自己記述型ではありません。ただし、ストリング記述子は、すべての `CMPAT(V*)` オプションで同じであり、LE ストリング記述子と同じコード・ページ・エンコードも共用します。

ストリング記述子

ストリング記述子では、最初の 2 バイトは、ストリングの最大長を指定します。この最大長は常にネイティブ・フォーマットで保持されます。

3 番目のバイトには、(例えば、`VARYING` ストリングのストリング長の保持フォーマットがリトル・エンディアンなのかビッグ・エンディアンなのか、また

WIDECHAR スtringのデータ保持フォーマットがリトル・エンディアンなのかビッグ・エンディアンなのかを示すフラグなど) さまざまなフラグが含まれます。

非可変ビット・StringのString記述子では、4 番目のバイトはビット・オフセットを表します。

CHARACTER StringのString記述子では、4 番目のバイトは、コンパイラー CODEPAGE オプションをエンコードします。

String記述子の宣言は次のとおりです。

```
declare
1 dso_string based( null() ),
2 dso_string_length      fixed bin(15),
2 dso_string_flags,
3 dso_string_is_varying      bit(1),
3 dso_string_is_varyingz     bit(1),
3 dso_string_has_nonnative_len bit(1), /* for varying */
3 dso_string_is_ascii        bit(1), /* for char */
3 dso_string_has_nonnative_data bit(1), /* for wchar */
3 *                          bit(1), /* reserved, '0'b */
3 *                          bit(1), /* reserved, '0'b */
3 *                          bit(1), /* reserved, '0'b */
2 * union,
3 dso_String_Codepage        ordinal ccs_Codepage_Enum,
3 dso_string_bitofs          fixed bin(8) unsigned,
2 dso_string_end            char(0);
```

コード・ページエンコードで指定可能な値は、次のように定義されます。

```
define ordinal
ccs_Codepage_Enum
( ccs_Codepage_01047 value(1)
, ccs_Codepage_01140
, ccs_Codepage_01141
, ccs_Codepage_01142
, ccs_Codepage_01143
, ccs_Codepage_01144
, ccs_Codepage_01145
, ccs_Codepage_01146
, ccs_Codepage_01147
, ccs_Codepage_01148
, ccs_Codepage_01149
, ccs_Codepage_00819
, ccs_Codepage_00813
, ccs_Codepage_00920
, ccs_Codepage_00037
, ccs_Codepage_00273
, ccs_Codepage_00277
, ccs_Codepage_00278
, ccs_Codepage_00280
, ccs_Codepage_00284
, ccs_Codepage_00285
, ccs_Codepage_00297
, ccs_Codepage_00500
, ccs_Codepage_00871
, ccs_Codepage_01026
, ccs_Codepage_01155
) unsigned prec(8);
```

配列記述子

以下の宣言では、配列の上限は 15 として宣言されていますが、実際の上限は常に記述されている配列の次元数に一致しているということを理解する必要があります。

CMPAT(V1) 配列記述子の宣言は次のとおりです。

```
declare
1 dso_v1 based( null() ),
2 dso_v1_rvo      fixed bin(31),    /* relative virtual origin */
2 dso_v1_data(1:15),
3 dso_v1_stride fixed bin(31),      /* multiplier          */
3 dso_v1_hbound fixed bin(15),      /* hbound              */
3 dso_v1_lbound fixed bin(15);      /* lbound              */
```

CMPAT(V2) 配列記述子の宣言は次のとおりです。

```
declare
1 dso_v2 based( null() ),
2 dso_v2_rvo      fixed bin(31),    /* relative virtual origin */
2 dso_v2_data(1:15),
3 dso_v2_stride fixed bin(31),      /* multiplier          */
3 dso_v2_hbound fixed bin(31),      /* hbound              */
3 dso_v2_lbound fixed bin(31);      /* lbound              */
```

CMPAT(V3) 配列記述子の宣言は次のとおりです。

```
declare
1 dso_v3 based( null() ),
2 dso_v3_rvo      fixed bin(63),    /* relative virtual origin */
2 dso_v3_data(1:15),
3 dso_v3_stride fixed bin(63),      /* multiplier          */
3 dso_v3_hbound fixed bin(63),      /* hbound              */
3 dso_v3_lbound fixed bin(63);      /* lbound              */
```

CMPAT(LE) 記述子

すべての LE 記述子は 4 バイト・フィールドで始まります。最初のバイトは、記述子のタイプ (スカラー、配列、構造体、または共用体) を指定します。残りの 3 バイトは、特定の記述子タイプで設定されていない限り、ゼロになります。

記述子ヘッダーの宣言は次のとおりです。

```
declare
1 dsc_Header based( sysnull() ),
2 dsc_Type      fixed bin(8) unsigned,
2 dsc_Datatype   fixed bin(8) unsigned,
2 *             fixed bin(8) unsigned,
2 *             fixed bin(8) unsigned;
```

dsc_Type フィールドに指定できる値は次のとおりです。

```
declare
dsc_Type_Unset      fixed bin(8) value(0),
dsc_Type_Element    fixed bin(8) value(2),
dsc_Type_Array      fixed bin(8) value(3),
dsc_Type_Structure   fixed bin(8) value(4),
dsc_Type_Union       fixed bin(8) value(4);
```

ストリング記述子

ストリング記述子では、ヘッダーの 2 番目のバイトがストリング・タイプ (ビット、文字、またはグラフィックのほかに、非可変、可変、または可変 z) を表します。

非可変ビット・ストリングのストリング記述子では、ヘッダーの 3 番目のバイトがビット・オフセットを表します。

CHARACTER ストリングのストリング記述子では、ヘッダーの 3 番目のバイトは、コンパイラ `CODEPAGE` オプションをエンコードします。

可変ストリングのストリング記述子では、4 番目のバイトに、ストリング長がネイティブ以外のフォーマットで保持されるかどうかを表すビットが含まれます。

文字ストリングのストリング記述子でも、4 番目のバイトに、ストリング・データが EBCDIC であるかどうかを表すビットが含まれます。

ストリング記述子の宣言は次のとおりです。

```
declare
1 dsc_String based( sysnull() ),
2 dsc_String_Header,
3 *          fixed bin(8) unsigned,
3 dsc_String_Type    fixed bin(8) unsigned,
3 * union,
4 dsc_String_Codepage ordinal ccs_Codepage_Enum,
4 dsc_String_BitOfs   fixed bin(8) unsigned,
3 *,
4 dsc_String_Has_Nonnative_Len   bit(1),
4 dsc_String_Is_Ebcdic          bit(1),
4 dsc_String_Has_Nonnative_Data  bit(1),
4 *                             bit(5),
2 dsc_String_Length   fixed bin(31); /* max length of string */
```

dsc_String_Type フィールドに指定できる値は次のとおりです。

```
declare
dsc_String_Type_Unset          fixed bin(8) value(0),
dsc_String_Type_Char_Nonvarying fixed bin(8) value(2),
dsc_String_Type_Char_Varyingz  fixed bin(8) value(3),
dsc_String_Type_Char_Varying2  fixed bin(8) value(4),
dsc_String_Type_Bit_Nonvarying  fixed bin(8) value(6),
dsc_String_Type_Bit_Varying2    fixed bin(8) value(7),
dsc_String_Type_Graphic_Nonvarying fixed bin(8) value(9),
dsc_String_Type_Graphic_Varyingz fixed bin(8) value(10),
dsc_String_Type_Graphic_Varying2 fixed bin(8) value(11),
dsc_String_Type_Widechar_Nonvarying fixed bin(8) value(13),
dsc_String_Type_Widechar_Varyingz  fixed bin(8) value(14),
dsc_String_Type_Widechar_Varying2  fixed bin(8) value(15);
```

配列記述子

配列記述子の宣言は次のとおりです。

```
declare
1 dsc_Array based( sysnull() ),
2 dsc_Array_Header like dsc_Header,
2 dsc_Array_EltLen  fixed bin(31), /* Length of array element */
2 dsc_Array_Rank    fixed bin(31), /* Count of dimensions */
2 dsc_Array_RV0     fixed bin(31), /* Relative virtual origin */
2 dsc_Array_Data( 1: 1 refer(dsc_Array_Rank) ),
```

```
3 dsc_Array_LBound fixed bin(31), /* LBound */
3 dsc_Array_Extent fixed bin(31), /* HBound - LBound + 1 */
3 dsc_Array_Stride fixed bin(31); /* Multiplier */
```

第 6 部 付録

付録. SYSADATA メッセージ情報

XINFO コンパイル時オプションの MSG サブオプションを指定すると、コンパイラーは、以下のものが入った SYSADATA ファイルを生成します。

- カウンター・レコード
- リテラル・レコード
- ファイル・レコード
- メッセージ・レコード

ファイル内のレコードは、必ずしも上記にリストした順序で生成されるとはかぎらず、例えば、リテラル・レコードとファイル・レコードが交互に混ざり合う場合があることに注意してください。SYSADATA ファイルを読み取るコードを作成している場合、以下の例外を除いて、ファイル内のレコードの順序は、あてになりません。

- カウンター・レコードは、ファイル内の最初のレコードになります。
- それぞれのリテラル・レコードは、それが定義するリテラルの参照よりも前にきます。
- それぞれのファイル・レコードは、それが記述するファイルの参照よりも前にきます。

SYSADATA ファイルについて

SYSADATA ファイルは順次バイナリー・ファイルです。z/OS バッチ環境では、コンパイラーは SYSADATA DD ステートメントで指定されたファイルに SYSADATA レコードを書き込みます。このファイルは PDS のメンバーであってはいません。その他のすべてのシステムでは、コンパイラーは拡張子「adt」を持つファイルに書き込みます。

ファイル内のそれぞれのレコードには、ヘッダーが入っています。この 8 バイトのヘッダーには、以下のフィールドがあり、これはそのファイル内のすべてのレコードで同じです。

コンパイラー

データを生成したコンパイラーを表す番号。PL/I の場合、この番号は 40 です。

エディション番号

データを生成したコンパイラーのエディション番号。この製品の場合、この番号は 2 です。

SYSADATA レベル

このファイル・フォーマットが示す SYSADATA のレベルを表す番号。この製品の場合、この番号は 4 です。

ヘッダーには、レコードごとに変わる、以下のフィールドもあります。

- レコード・タイプ

- レコードが次のレコードに継続するかどうか

可能なレコード・タイプは、図 105 に示すような、序数値としてエンコードされています。

```

Define ordinal xin_Rect
(Xin_Rect_Msg      value(50), /* Message record      */
Xin_Rect_Fil       value(57), /* File record        */
Xin_Rect_Sum       value(61), /* Summary record     */
Xin_Rect_Src       value(63), /* Source record      */
Xin_Rect_Tok       value(64), /* Token record       */
Xin_Rect_Sym       value(66), /* Symbol record      */
Xin_Rect_Lit       value(67), /* Literal record     */
Xin_Rect_Syn       value(69), /* Syntax record      */
Xin_Rect_Ord_Type  value(80), /* ordinal type record */
Xin_Rect_Ord_Elem  value(81), /* ordinal element record */
Xin_Rect_Ctr       value(82) ) /* counter record     */
prec(15);

```

図 105. 序数値としてエンコードされたレコード・タイプ

レコードのヘッダー部分の宣言は、図 106 に示されています。

```

Dcl
1 Xin_Hdr Based( null() ), /* Header portion */
/* */
2 Xin_Hdr_Prod /* Language code */
fixed bin(8) unsigned, /* */
/* */
2 Xin_Hdr_Rect /* Record type */
unal ordinal xin_Rect, /* */
/* */
2 Xin_Hdr_Level /* SYSADATA level */
fixed bin(8) unsigned, /* */
/* */
2 * union, /* */
3 xin_Hdr_Flags bit(8), /* flags */
3 *, /* */
4 * bit(6), /* Reserved */
4 Xin_Hdr_Little_Endian /* ints are little endian */
bit(1), /* */
4 Xin_Hdr_Cont bit(1), /* Record continued in next rec */
/* */
2 Xin_Hdr_Edition /* compiler "edition" */
fixed bin(8) unsigned, /* */
/* */
2 Xin_Hdr_Fill bit(32), /* reserved */
/* */
2 Xin_Hdr_Data_Len /* length of data part */
fixed bin(16) unsigned, /* */
/* */
2 Xin_Hdr_End char(0); /* */

```

図 106. レコードのヘッダー部分の宣言

サマリー・レコード

タイプが `Xin_Rect_Sum` のレコード (サマリー・レコード) は、ファイルの最初のレコードになります。サマリー・レコードの宣言を、図 107 に示します。

```
Dcl
1 Xin_Sum      Based( null() ), /* summary record          */
/*                                                    */
2 Xin_Sum_Hdr  /* standard header          */
    like Xin_Hdr, /*                                                    */
/*                                                    */
2 Xin_Sum_Max_Severity /* max severity from compiler */
    fixed bin(32) unsigned, /*                                                    */
/*                                                    */
2 Xin_Sum_Left_Margin /* left margin                */
    fixed bin(16) unsigned, /*                                                    */
/*                                                    */
2 Xin_Sum_Right_Margin /* right margin                */
    fixed bin(16) unsigned, /*                                                    */
/*                                                    */
2 xin_Sum_Rsrvd(15) /* reserved                    */
    fixed bin(32) unsigned; /*                                                    */
```

図 107. サマリー・レコードの宣言

カウンター・レコード

各カウンター・レコードは、後続のレコード・タイプについて、ファイルに入っているそのタイプのレコードの数と、それらのレコードが占有するバイト数を指定します。

```
Dcl
1 xin_Ctr      Based( null() ), /* counter/size record          */
/*                                                    */
2 xin_Ctr_Hdr  /* standard header          */
    like xin_Hdr, /*                                                    */
/*                                                    */
2 xin_Ctr_Rect /* record type              */
    unal ordinal xin_Rect, /*                                                    */
/*                                                    */
2 *            /*                            */
    fixed bin(16) unsigned, /*                                                    */
/*                                                    */
2 xin_Ctr_Count /* count of that record type */
    fixed bin(31) unsigned, /*                                                    */
/*                                                    */
2 xin_Ctr_Size  /* size used                 */
    fixed bin(31) unsigned; /*                                                    */
```

図 108. カウンター・レコードの宣言

リテラル・レコード

それぞれのリテラル・レコードは、リテラル索引と呼ばれる 1 つの番号を割り当て、この特定のレコードに指定された文字を参照するために、後のレコードによって使用されるようにします。

```
Dcl
  1 xin_Lit      Based( null() ), /* literal record          */
                                /*                               */
    2 xin_Lit_Hdr /* standard header          */
      like xin_Hdr, /*                               */
                                /*                               */
    2 xin_Lit_Inx /* adata index for literal */
      fixed bin(31) unsigned, /*                               */
                                /*                               */
    2 xin_Lit_Len /* length of literal      */
      fixed bin(31) unsigned, /*                               */
                                /*                               */
    2 xin_Lit_Val  char(2000); /* literal value          */
```

図 109. リテラル・レコードの宣言

ファイル・レコード

それぞれのファイル・レコードは、ファイル索引と呼ばれる 1 つの番号を割り当て、このレコードに記述されたファイルを参照するために、後のレコードによって使用されるようにします。記述されたファイルは、PL/I 1 次ソース・ファイルか INCLUDE されたファイルの場合があります。それぞれのファイル・レコードは、そのファイルの完全修飾名に対するリテラル索引を指定します。

INCLUDE されたファイルの場合、それぞれのファイル・レコードには、INCLUDE 要求からのファイル索引とソース行番号も入ります。(1 次ソース・ファイルの場合、これらのフィールドはゼロになります。)

```
Dcl
  1 xin_Fil      Based( null() ), /* file record          */
                                /*                               */
    2 xin_Fil_Hdr /* standard header          */
      like xin_Hdr, /*                               */
                                /*                               */
    2 xin_Fil_File_Id /* file id from whence it */
      fixed bin(31) unsigned, /* was INCLUDED          */
                                /*                               */
    2 xin_Fil_Line_No /* line no within that file */
      fixed bin(31) unsigned, /*                               */
                                /*                               */
    2 xin_Fil_Id /* id assigned to this file */
      fixed bin(31) unsigned, /*                               */
                                /*                               */
    2 xin_Fil_Name /* literal index of the     */
      fixed bin(31) unsigned; /* fully qualified file name */
```

図 110. ファイル・レコードの宣言

メッセージ・レコード

それぞれのメッセージ・レコードは、コンパイル中に出されたメッセージを記述します。抑止されたメッセージに対しては、メッセージ・レコードは生成されません。

それぞれのメッセージ・レコードには、以下のものが入っています。

- そのメッセージの起因となったファイルおよび行のファイル索引およびソース行番号。メッセージがすべてコンパイルに関係するものである場合、このフィールドはゼロになります。
- そのメッセージに関連する ID (例えば、IBM1502) および重大度。
- そのメッセージのテキスト。

メッセージ・レコードの宣言は、以下のとおりです。

```
Dcl
  1 xin_Msg      Based( null() ), /* message record          */
                                /*                               */
    2 xin_Msg_Hdr /* standard header          */
        like xin_Hdr, /*                               */
                                /*                               */
    2 xin_Msg_File_Id /* file id              */
        fixed bin(31) unsigned, /*                               */
                                /*                               */
    2 xin_Msg_Line_No /* line no within file  */
        fixed bin(31) unsigned, /*                               */
                                /*                               */
    2 xin_Msg_Id      /* identifier (i.e. IBM1502) */
                                char(16), /*                               */
                                /*                               */
    2 xin_Msg_Severity /* severity (0, 4, 8, 12 or 16) */
        fixed bin(15) signed, /*                               */
                                /*                               */
    2 xin_Msg_Length  /* length of message      */
        fixed bin(16) unsigned, /*                               */
                                /*                               */
    2 Xin_Msg_Text     /* actual message         */
        char( 100 refer(xin_Msg_Length) );
```

図 111. メッセージ・レコードの宣言

SYSADATA シンボル情報について

XINFO コンパイル時オプションの SYM サブオプションを指定すると、コンパイラーは、MSG サブオプションの場合に生成されるレコードに加えて、以下のものが入った SYSADATA ファイルを生成します。

- 序数タイプ・レコード
- 序数エレメント・レコード
- シンボル・レコード

シンボル・レコードは、組み込み関数、総称変数、または非定数エクステンントを持つ変数の場合には生成されません。

序数タイプ・レコード

それぞれの序数タイプ・レコードは、序数タイプ索引と呼ばれる 1 つの番号を割り当て、このレコードに記述された序数タイプを参照するために、後のレコードによって使用されるようにします。タイプの名前は、リテラル索引によって示されます。それぞれの序数タイプ・レコードには、その序数タイプが宣言されたファイルおよび行のファイル索引およびソース行番号が入っています。

それぞれの序数タイプ・レコードには、以下のものが入っています。

- そのタイプによって定義された値の数のカウント
- そのタイプに関連する精度
- 符号付きであるか符号なしであるかを示すビット

```
declare                                     /* */
1 xin_Ord_Type   based( null() ),          /* */
/* */
2 xin_Ord_Type_Hdr /* standard header      /* */
   like xin_Hdr, /* */
/* */
2 xin_Ord_Type_File_Id /* file id          /* */
   fixed bin(31) unsigned, /* */
/* */
2 xin_Ord_Type_Line_No /* line no within file /* */
   fixed bin(31) unsigned, /* */
/* */
2 xin_Ord_Type_Id /* identifying number    /* */
   fixed bin(31), /* */
/* */
2 xin_Ord_Type_Count /* count of elements /* */
   fixed bin(31), /* */
/* */
2 xin_Ord_Type_Prec /* precision for ordinal /* */
   fixed bin(08) unsigned, /* */
/* */
2 *, /* */
3 xin_Ordinal_Type_Signed /* signed attribute applies /* */
   bit(1), /* */
3 xin_Ordinal_Type_Unsigned /* unsigned attribute applies /* */
   bit(1), /* */
3 * /* unused /* */
   bit(6), /* */
/* */
2 * /* unused /* */
   char(2), /* */
/* */
2 xin_Ord_Type_Name /* type name          /* */
   fixed bin(31); /* */
```

図 112. 序数タイプ・レコードの宣言

序数エレメント・レコード

それぞれの序数タイプ・レコードの直後には、その序数によって指定される値を記述する、一連のレコードが (序数タイプ・カウントで指定された数だけ) 続きます。

それぞれの序数エレメント・レコードは、序数エレメント索引と呼ばれる 1 つの番号を割り当て、このレコードに記述された序数エレメントを参照するために、後のレコードによって使用されるようにします。エレメントの名前は、リテラル索引によって示されます。それぞれの序数エレメント・レコードには、その序数エレメントが宣言されたファイルおよび行のファイル索引およびソース行番号が入っています。

さらに、各序数エレメント・レコードには、以下のものが入っています。

- そのエレメントが属している序数タイプの序数タイプ索引
- そのエレメントの値

```
declare                                     /*
1 xin_Ord_Elem   based( null() ),          /*
                                     /*
2 xin_Ord_Elem_Hdr      /* standard header  /*
    like xin_Hdr,        /*
                                     /*
2 xin_Ord_Elem_File_Id  /* file id          /*
    fixed bin(31) unsigned, /*
                                     /*
2 xin_Ord_Elem_Line_No  /* line no within file /*
    fixed bin(31) unsigned, /*
                                     /*
2 xin_Ord_Elem_Id       /* identifying number /*
    fixed bin(31),       /*
                                     /*
2 xin_Ord_Elem_Type_Id  /* id of ordinal type /*
    fixed bin(31),       /*
                                     /*
2 xin_Ord_Elem_Value    /* ordinal value    /*
    fixed bin(31),       /*
                                     /*
2 xin_Ord_Elem_Name     /* ordinal name     /*
    fixed bin(31);       /*
```

図 113. 序数エレメント・レコードの宣言

シンボル・レコード

それぞれのシンボル・レコードは、シンボル索引と呼ばれる 1 つの番号を割り当て、このレコードに記述されたシンボル (例えば、ユーザー変数または定数の名前) を参照するために、後のレコードによって使用されるようにします。ID の名前は、リテラル索引によって示されます。それぞれのシンボル・レコードには、そのシンボルが宣言されたファイルおよび行のファイル索引およびソース行番号が入っています。

ID が構造体または共用体の一部である場合、シンボル・レコードには、以下のそれぞれに対するシンボル索引が入ります。

- 最初の兄弟 (存在する場合)
- 親 (存在する場合)
- 最初の子 (存在する場合)

以下の構造体を見てください。

```

dcl
  1 a
    , 3 b      fixed bin
    , 3 c      fixed bin
    , 3 d
      , 5 e    fixed bin
      , 5 f    fixed bin
;

```

上記の構造体のエレメントに割り当てられたシンボル索引は、以下のようになります。

symbol	index	sibling	parent	child
a	1	0	0	2
b	2	3	1	0
c	3	4	1	0
d	4	0	1	5
e	5	6	4	0
f	6	0	4	0

図 114. 構造体のエレメントに割り当てられたシンボル索引

それぞれのシンボル・レコードには、さまざまな属性がこの変数に適用されるかどうかを示す一連の bit(1) フィールドも含まれています。

それぞれのシンボル・レコードには、以下のエレメントも入っています。

ユーザー指定の構造体レベル

これは、ID に対するユーザー指定の構造体レベルです。上記の構造体のエレメント c の場合、値は 3 です。非構造体メンバーの場合、値は 1 に設定されます。

論理構造体レベル

ID の論理構造体レベル。上記の構造体のエレメント c の場合、値は 2 です。非構造体メンバーの場合、値は 1 に設定されます。

次元

継承次元をカウントせずに、変数に宣言された次元の数。

すべての継承次元を含む、変数の次元の数。

オフセット

最外部の親構造体の中でのオフセット。

基本サイズ

基本サイズは、変数がビット位置合わせしている場合はビット単位、それ以外の場合はバイト単位です。いずれの場合も、これは、どの次元の因数でもありません。

サイズ

その次元の因数となるバイト単位でのサイズ。

位置合わせ

以下によって、示されます。

- 0 (ビット位置合わせ)
- 7 (バイト位置合わせ)
- 15 (ハーフワード位置合わせ)
- 31 (フルワード位置合わせ)
- 63 (4 倍長ワード位置合わせ)

レコード内の共用体は、変数のストレージ・クラスに従属する情報を記述するための専用のものです。

静的変数

変数が、別個の外部名を持つ外部 (`dcl x ext('y')`) として宣言された場合、その名前のリテラル索引が指定されます。

基底付き変数

変数が、配列のエレメントではない別のマップ変数を基にするものとして宣言された場合、その変数のシンボル索引が指定されます。

定義済み変数

変数が、配列のエレメントではない別のマップされた変数で定義されたものとして宣言された場合、その変数のシンボル索引がここに指定されます。これは、その位置属性が定数である場合にも指定されます。

変数のデータ・タイプは、484 ページの図 115 に示された序数によって指定されます。

```

define
ordinal
xin_Data_Kind
(
xin_Data_Kind_Unset
,xin_Data_Kind_Character
,xin_Data_Kind_Bit
,xin_Data_Kind_Graphic
,xin_Data_Kind_Fixed
,xin_Data_Kind_Float
,xin_Data_Kind_Picture
,xin_Data_Kind_Pointer
,xin_Data_Kind_Offset
,xin_Data_Kind_Entry
,xin_Data_Kind_File
,xin_Data_Kind_Label
,xin_Data_Kind_Format
,xin_Data_Kind_Area
,xin_Data_Kind_Task
,xin_Data_Kind_Event
,xin_Data_Kind_Condition
,xin_Data_Kind_Structure
,xin_Data_Kind_Union
,xin_Data_Kind_Descriptor
,xin_Data_Kind_Ordinal
,xin_Data_Kind_Handle
,xin_Data_Kind_Type
) prec(8) unsigned;

```

図 115. 変数のデータ・タイプ

レコード内の共用体は、変数のデータ・タイプに従属する情報を記述するための専用のものです。この情報のほとんどは、おそらく以下の場合を除き、説明の必要がありません (例えば、算術型の精度など)。

ピクチャー変数

ピクチャー指定のリテラル索引が指定されます。

入り口変数

変数が戻り属性を持っている場合、戻り記述のシンボル索引が指定されます。

序数変数

序数タイプ索引が指定されます。

タイプ付き変数およびハンドル

基礎となるタイプのシンボル索引が指定されます。

ストリングおよび区域変数

戻り記述のシンボル索引に加えて、エクステントのタイプと値が指定されます。エクステントのタイプは、以下の値によってエンコードされます。

```

declare
(
xin_Extent_Constant    value(01)
,xin_Extent_Star       value(02)
,xin_Extent_Nonconstant value(04)
,xin_Extent_Refer      value(08)
,xin_Extent_In_Error   value(16)
)
fixed bin;

```

エレメントが何らかの次元を持つ場合、その下部と上部の境界のタイプと値が、レコードの最後に指定されます。これらのフィールドは、エレメントが次元を持たな

い場合には存在しません。 図 116 に、シンボル・レコードの宣言を示します。

```

declare                                     /* */
1 xin_Sym      based( null() ),           /* */
                                     /* */
2 Xin_Sym_Hdr   /* standard header        /* */
   like Xin_Hdr, /* */
                                     /* */
2 Xin_Sym_File_Id /* file id              /* */
   fixed bin(32) unsigned, /* */
                                     /* */
2 Xin_Sym_Line_No /* line no within file  /* */
   fixed bin(32) unsigned, /* */
                                     /* */
2 xin_Id         /* identifying number    /* */
   fixed bin(31), /* */
                                     /* */
2 xin_Sibling    /* xin_id of next sibling /* */
   fixed bin(31), /* */
                                     /* */
2 xin_Parent     /* xin_id of parent      /* */
   fixed bin(31), /* */
                                     /* */
2 xin_Child      /* xin_id of first child /* */
   fixed bin(31), /* */
                                     /* */
2 xin_Blz_Id     /* blk_id of owning block /* */
   fixed bin(31), /* */
                                     /* */
2 xin_Sym_Tok    /* token id of declaring token /* */
   fixed bin(31), /* */
                                     /* */
2 xin_Logical_Level /* logical level in structure /* */
   unsigned fixed bin(08), /* */
                                     /* */
2 xin_Physical_Level /* given level in structure /* */
   unsigned fixed bin(08), /* */
                                     /* */
2 xin_Total_Dims  /* Total number of dims /* */
   unsigned fixed bin(08), /* */
                                     /* */
2 xin_Own_Dims    /* count of self-made dims /* */
   unsigned fixed bin(08), /* */
                                     /* */

```

図 116. シンボル・レコードの宣言 (1/7)

```

2 xin_Attr_Flags union,      /* */
                               /* */
3 *          bit(64), /* */
                               /* */
3 *,          /* */
                               /* */
4 xin_Attr_Automatic      /* */
                               /* */
4 xin_Attr_Based          bit(1), /* */
                               /* */
4 xin_Attr_Controlled     bit(1), /* */
                               /* */
4 xin_Attr_Defined        bit(1), /* */
                               /* */
4 xin_Attr_Parameter      bit(1), /* */
                               /* */
4 xin_Attr_Position       bit(1), /* */
                               /* */
4 xin_Attr_Reserved       bit(1), /* */
                               /* */
4 xin_Attr_Static         bit(1), /* */
                               /* */
4 xin_Attr_Condition      bit(1), /* */
                               /* */
4 xin_Attr_Constant       bit(1), /* */
                               /* */
4 xin_Attr_Variable       bit(1), /* */
                               /* */
4 xin_Attr_Internal       bit(1), /* */
                               /* */
4 xin_Attr_External       bit(1), /* */
                               /* */
4 xin_Attr_Abnormal       bit(1), /* */
                               /* */
4 xin_Attr_Normal         bit(1), /* */
                               /* */
4 xin_Attr_Assignable     bit(1), /* */
                               /* */
4 xin_Attr_Nonassignable  bit(1), /* */
                               /* */
4 xin_Attr_Aligned        bit(1), /* */
                               /* */
4 xin_Attr_Unaligned      bit(1), /* */
                               /* */
4 xin_Attr_Descriptor     bit(1), /* */
                               /* */
4 xin_Attr_Value          bit(1), /* */
                               /* */
4 xin_Attr_Byvalue        bit(1), /* */
                               /* */
4 xin_Attr_Byaddr         bit(1), /* */
                               /* */
4 xin_Attr_Connected      bit(1), /* */
                               /* */
4 xin_Attr_Nonconnected   bit(1), /* */
                               /* */
4 xin_Attr_Optional       bit(1), /* */
                               /* */

```

図 116. シンボル・レコードの宣言 (2/7)

```

4 xin_Attr_Native      /* */
                        bit(1), /* */
4 xin_Attr_Nonnative   /* */
                        bit(1), /* */
4 xin_Attr_Initial     /* */
                        bit(1), /* */
4 xin_Attr_Typedef     /* */
                        bit(1), /* */
4 xin_Attr_Builtin     /* */
                        bit(1), /* */
4 xin_Attr_Generic     /* */
                        bit(1), /* */
4 xin_Attr_Date        /* */
                        bit(1), /* */
4 xin_Attr_Noinit     /* */
                        bit(1), /* */
2 xin_Data_Is          /* */
    ordinal xin_Data_Kind, /* */
2 xin_Misc_Flags union, /* */
3 *                    bit(8), /* */
3 *,                  /* */
4 xin_Implicit_Dcl     /* dcl is implicit */
                        bit(1), /* */
4 xin_Contextual_Dcl   /* dcl is contextual */
                        bit(1), /* */
4 xin_Has_Been_Mapped  /* aggregate has been mapped */
                        bit(1), /* */
2 xin_Align            /* alignment */
    unsigned fixed bin(08), /* */
2 xin_Begin_Offset     /* bitlocation(sym) */
    unsigned fixed bin(08), /* */
2 xin_Offset           /* location(sym) */
    fixed bin(31), /* */
2 xin_Size             /* length in bytes, with all */
    fixed bin(31), /* children and array */
                        /* elements factored in */
2 xin_Base_Size        /* element length - in bytes */
    fixed bin(31), /* unless bit aligned */
2 xin_Name             /* name - id of lit record */
    fixed bin(31), /* */
2 * union,            /* */
3 xin_Static_Data,    /* */
4 xin_Static_Ext      /* id of literal specifying its */
    fixed bin(31), /* external name */

```

図 116. シンボル・レコードの宣言 (3/7)

```

3 xin_Based_Data,          /* */
                          /* */
4 xin_Based_On_Id          /* xin_Id of basing reference */
    fixed bin(31),        /* 0 if not simple */
                          /* */
3 xin_Defined_Data,        /* */
                          /* */
4 xin_Defined_On_Id        /* xin_Id of basing reference */
    fixed bin(31),        /* 0 if not simple */
                          /* */
4 xin_Defined_Pos          /* -1 if not constant */
    fixed bin(31),        /* */
                          /* */
3 xin_Parm_Data,           /* */
                          /* */
4 xin_Parm_Index           /* index of parm */
    fixed bin(31),        /* 1 for first, etc */
                          /* */
2 * union,                 /* */
                          /* */
3 xin_Str_Data,            /* used for char, bit, graphic */
                          /* and area, but not used for */
                          /* picture character or numeric */
                          /* */
4 xin_Str_Len_Node         /* length as parse tree */
    fixed bin(31),        /* */
                          /* */
4 xin_Str_Len              /* length: if type is constant */
    fixed bin(31),        /* */
                          /* */
4 xin_Str_Len_Type         /* type */
    unsigned fixed bin(08), /* */
                          /* */
4 *,                       /* */
5 xin_Str_Varying          /* */
    bit(1),               /* */
5 xin_Str_Nonvarying       /* */
    bit(1),               /* */
5 xin_Str_Varyingz         /* */
    bit(1),               /* */
                          /* */
4 xin_Str_Date             /* index of date literal */
    fixed bin(31),        /* */
                          /* */
3 xin_Arith_Data,          /* used for fixed and float */
                          /* */
4 xin_Arith_Precision       /* precision */
    unsigned fixed bin(08), /* */
                          /* */
4 xin_Arith_Scale_Factor   /* scale factor */
    signed fixed bin(07),  /* */
                          /* */

```

図 116. シンボル・レコードの宣言 (4/7)

```

4 *,                               /* */
5 xin_Arith_Binary                 /* */
    bit(1),                        /* */
5 xin_Arith_Decimal                /* */
    bit(1),                        /* */
5 xin_Arith_Fixed                  /* */
    bit(1),                        /* */
5 xin_Arith_Float                  /* */
    bit(1),                        /* */
5 xin_Arith_Real                   /* */
    bit(1),                        /* */
5 xin_Arith_Complex                /* */
    bit(1),                        /* */
5 xin_Arith_Signed                 /* */
    bit(1),                        /* */
5 xin_Arith_Unsigned               /* */
    bit(1),                        /* */
5 xin_Arith_Ieee                   /* */
    bit(1),                        /* */
5 xin_Arith_Hexadec                /* */
    bit(1),                        /* */
    /* */
4 *                                /* unused */
    fixed bin(31),                 /* */
    /* */
4 *                                /* unused */
    fixed bin(31),                 /* */
    /* */
4 xin_Arith_Date                   /* index of date literal */
    fixed bin(31),                 /* */
    /* */
3 xin_Ordinal_Data,                /* used for ordinal */
    /* */
4 xin_Ordinal_Type_Id              /* type id */
    fixed bin(31),                 /* */
    /* */
3 xin_Type_Data,                   /* used for typed */
    /* */
4 xin_Type_Is                      /* type id */
    fixed bin(31),                 /* */
    /* */
3 xin_Pic_Data,                    /* used for all pictures */
    /* */
4 xin_Pic_Ext                      /* external specification */
    fixed bin(31),                 /* */
    /* */
4 *,                               /* */
5 xin_Pic_Fixed                    /* */
    bit(1),                        /* */
5 xin_Pic_Float                    /* */
    bit(1),                        /* */
5 xin_Pic_Character                /* */
    bit(1),                        /* */
5 xin_Pic_Real                     /* */
    bit(1),                        /* */
5 xin_Pic_Complex                  /* */
    bit(1),                        /* */

```

図 116. シンボル・レコードの宣言 (5/7)

```

5 *                               /* */
        bit(3), /* */
5 *                               /* */
        bit(8), /* */
5 xin_Pic_Prec /* */
        unsigned /* */
        fixed bin(08), /* */
5 xin_Pic_Scale /* */
        signed /* */
        fixed bin(07), /* */
        /* */
4 *                               /* unused */
        fixed bin(31), /* */
        /* */
4 xin_Pic_Date /* index of date literal */
        fixed bin(31), /* */
        /* */
3 xin_Entry_Data, /* */
        /* */
4 xin_Entry_Min /* min number of args */
        fixed bin(15), /* allowed when invoked */
        /* */
4 xin_Entry_Max /* max number of args */
        fixed bin(15), /* allowed when invoked */
        /* */
4 xin_Entry_Returns_Id /* xin_Id of returns descriptor */
        fixed bin(31), /* */
        /* */
4 xin_Entry_Parms_Id /* xin_Id of first parms */
        fixed bin(31), /* */
        /* */
4 *, /* */
5 xin_Entry_Returns /* */
        bit(1), /* */
5 xin_Entry_Limited /* */
        bit(1), /* */
5 xin_Entry_Fetchable /* */
        bit(1), /* */
5 xin_Entry_Is_Proc /* */
        bit(1), /* */
5 xin_Entry_Is_Secondary /* */
        bit(1), /* */
        /* */
3 xin_Ptr_Data, /* */
        /* */
4 *, /* */
5 xin_Ptr_Segmented /* */
        bit(1), /* */
        /* */
3 xin_Offset_Data, /* */
        /* */
4 xin_Offset_Area /* */
        fixed bin(31), /* */
        /* */
3 xin_Sym_Bif_Id /* */
        ordinal xin_Bif_Kind, /* */
        /* */
        /* */

```

図 116. シンボル・レコードの宣言 (6/7)

```

3 xin_File_Data,          /* */
                          /* */
4 *,                      /* */
  5 xin_File_Buffered     /* */
    bit(1),               /* */
  5 xin_File_Direct       /* */
    bit(1),               /* */
  5 xin_File_Exclusive    /* */
    bit(1),               /* */
  5 xin_File_Input        /* */
    bit(1),               /* */
  5 xin_File_Keyed        /* */
    bit(1),               /* */
  5 xin_File_Output       /* */
    bit(1),               /* */
  5 xin_File_Print        /* */
    bit(1),               /* */
  5 xin_File_Record       /* */
    bit(1),               /* */
  5 xin_File_Stream       /* */
    bit(1),               /* */
  5 xin_File_Transient    /* */
    bit(1),               /* */
  5 xin_File_Unbuffered   /* */
    bit(1),               /* */
  5 xin_File_Update       /* */
    bit(1),               /* */
                          /* */
2 * union,                /* */
                          /* */
  3 xin_Value_Id           /* id of value lit - if the */
    fixed bin(31),         /* xin_Attr_Value flag is set */
                          /* */
  3 xin_First Stmt_Id      /* id of first stmt record - */
    fixed bin(31),         /* if xin_Attr_Entry and */
                          /* xin_Entry_Is_Proc flags */
                          /* are both set */
                          /* */
2 xin_Bounds dim(15),     /* */
                          /* */
  3 xin_Lbound_Type        /* lbound type */
    unsigned fixed bin(08), /* */
                          /* */
  3 xin_Hbound_Type        /* hbound type */
    unsigned fixed bin(08), /* */
                          /* */
3 *                        /* */
    char(2),               /* */
                          /* */
  3 xin_Lbound_Node        /* expression parse tree */
    fixed bin(31),         /* */
                          /* */
  3 xin_Hbound_Node        /* expression parse tree */
    fixed bin(31),         /* */
                          /* */
  3 xin_Lbound             /* value: if type is constant */
    fixed bin(31),         /* xin_Id: if type is refer */
                          /* */
  3 xin_Hbound             /* value: if type is constant */
    fixed bin(31),         /* xin_Id: if type is refer */
                          /* */
2 *                        /* */
    char(0);

```

図 116. シンボル・レコードの宣言 (7/7)

序数 `xin_Bif_Kind` の定義を、図 117 に示します。

```
define
ordinal
xin_Bif_Kind
(
xin_Bif_Unknown
,xin_bif_abs
,xin_bif_acos
,xin_bif_add
,xin_bif_addr
,xin_bif_all
,xin_bif_allocation
,xin_bif_allocn
,xin_bif_any
,xin_bif_asin
,xin_bif_atan
,xin_bif_atand
,xin_bif_atanh
,xin_bif_bin
,xin_bif_binvalue
,xin_bif_binary
,xin_bif_binaryvalue
,xin_bif_bit
,xin_bif_bool
,xin_bif_ceil
,xin_bif_char
,xin_bif_completion
,xin_bif_complex
,xin_bif_conjg
,xin_bif_copy
,xin_bif_cos
,xin_bif_cosd
,xin_bif_cosh
,xin_bif_count
,xin_bif_cpln
,xin_bif_cplx
,xin_bif_cstg
,xin_bif_currentstorage
,xin_bif_datafield
,xin_bif_date
,xin_bif_datetime
,xin_bif_dec
,xin_bif_decimal
,xin_bif_dim
,xin_bif_divide
,xin_bif_empty
,xin_bif_entryaddr
,xin_bif_erf
,xin_bif_erfc
,xin_bif_exp
,xin_bif_fixed
,xin_bif_float
,xin_bif_floor
,xin_bif_graphic
,xin_bif_hbound
```

図 117. `xin_Bif_Kind` の宣言 (1/5)

```
,xin_bif_high
,xin_bif_imag
,xin_bif_index
,xin_bif_lbound
,xin_bif_length
,xin_bif_lineno
,xin_bif_log
,xin_bif_log10
,xin_bif_log2
,xin_bif_low
,xin_bif_max
,xin_bif_min
,xin_bif_mod
,xin_bif_mpstr
,xin_bif_multiply
,xin_bif_null
,xin_bif_offset
,xin_bif_onchar
,xin_bif_oncode
,xin_bif_oncount
,xin_bif_onfile
,xin_bif_onkey
,xin_bif_onloc
,xin_bif_onsource
,xin_bif_pageno
,xin_bif_plicanc
,xin_bif_plickpt
,xin_bif_plidump
,xin_bif_plirest
,xin_bif_pliretc
,xin_bif_pliretv
,xin_bif_plisrta
,xin_bif_plisrtb
,xin_bif_plisrtc
,xin_bif_plisrtd
,xin_bif_plitest
,xin_bif_pointer
,xin_bif_pointeradd
,xin_bif_pointervalue
,xin_bif_poly
,xin_bif_prec
,xin_bif_precision
,xin_bif_priority
,xin_bif_prod
,xin_bif_ptr
,xin_bif_ptradd
,xin_bif_ptrvalue
,xin_bif_real
,xin_bif_repeat
,xin_bif_round
,xin_bif_samekey
,xin_bif_sign
,xin_bif_sin
,xin_bif_sind
,xin_bif_sinh
,xin_bif_sqrt
,xin_bif_status
,xin_bif_stg
```

図 117. *xin_Bif_Kind* の宣言 (2/5)

```
,xin_bif_storage
,xin_bif_string
,xin_bif_substr
,xin_bif_sum
,xin_bif_sysnull
,xin_bif_tan
,xin_bif_tand
,xin_bif_tanh
,xin_bif_time
,xin_bif_translate
,xin_bif_trunc
,xin_bif_unspec
,xin_bif_verify
,xin_bif_days
,xin_bif_daystodate

,xin_bif_acosf
,xin_bif_addrdata
,xin_bif_alloc
,xin_bif_allocate
,xin_bif_alloctype
,xin_bif_asinf
,xin_bif_atanf
,xin_bif_auto
,xin_bif_automatic
,xin_bif_availablearea
,xin_bif_bitloc
,xin_bif_bitlocation
,xin_bif_byte
,xin_bif_cds
,xin_bif_center
,xin_bif_centerleft
,xin_bif_centerright
,xin_bif_centre
,xin_bif_centreleft
,xin_bif_centreright
,xin_bif_character
,xin_bif_charg
,xin_bif_chargraphic
,xin_bif_charval
,xin_bif_checkstg
,xin_bif_collate
,xin_bif_compare
,xin_bif_cosf
,xin_bif_cs
,xin_bif_currentsize
,xin_bif_daystosecs
,xin_bif_dimension
,xin_bif_edit
,xin_bif_endfile
,xin_bif_epsilon
,xin_bif_expf
,xin_bif_exponent
,xin_bif_fileddint
,xin_bif_fileddtest
,xin_bif_fileddword
,xin_bif_fileid
,xin_bif_fileread
,xin_bif_fileseek
,xin_bif_filetell
,xin_bif_filewrite
,xin_bif_gamma
,xin_bif_getenv
```

図 117. *xin_Bif_Kind* の宣言 (3/5)

```
,xin_bif_handle
,xin_bif_hex
,xin_bif_heximage
,xin_bif_huge
,xin_bif_iand
,xin_bif_ior
,xin_bif_inot
,xin_bif_ior
,xin_bif_isinged
,xin_bif_isll
,xin_bif_ismain
,xin_bif_isrl
,xin_bif_iunsigned
,xin_bif_left
,xin_bif_loc
,xin_bif_location
,xin_bif_log10f
,xin_bif_logf
,xin_bif_loggamma
,xin_bif_lower2
,xin_bif_lowercase
,xin_bif_maxexp
,xin_bif_maxlength
,xin_bif_memindex
,xin_bif_memsearch
,xin_bif_memsearchr
,xin_bif_memverify
,xin_bif_memverifyr
,xin_bif_minexp
,xin_bif_offsetadd
,xin_bif_offsetdiff
,xin_bif_offsetsubtract
,xin_bif_offsetvalue
,xin_bif_omitted
,xin_bif_oncondcond
,xin_bif_oncondid
,xin_bif_ongsource
,xin_bif_onsubcode
,xin_bif_onwchar
,xin_bif_onsource
,xin_bif_ordinalname
,xin_bif_ordinalpred
,xin_bif_ordinalsucc
,xin_bif_packagename
,xin_bif_picspec
,xin_bif_places
,xin_bif_pliascii
,xin_bif_pliebcdic
,xin_bif_plifill
,xin_bif_plifree
,xin_bif_plimove
,xin_bif_pliover
,xin_bif_plisaxa
,xin_bif_plisaxb
```

図 117. *xin_Bif_Kind* の宣言 (4/5)

```
,xin_bif_pointerdiff
,xin_bif_pointersubtract
,xin_bif_pred
,xin_bif_present
,xin_bif_procedurename
,xin_bif_procname
,xin_bif_ptrdiff
,xin_bif_ptrsubtract
,xin_bif_putenv
,xin_bif_radix
,xin_bif_raise2
,xin_bif_random
,xin_bif_rank
,xin_bif_rem
,xin_bif_repattern
,xin_bif_replaceby2
,xin_bif_reverse
,xin_bif_right
,xin_bif_scale
,xin_bif_search
,xin_bif_searchr
,xin_bif_secs
,xin_bif_secstodate
,xin_bif_secstodays
,xin_bif_signed
,xin_bif_sinf
,xin_bif_size
,xin_bif_sourcefile
,xin_bif_sourceline
,xin_bif_sqrtf
,xin_bif_subtract
,xin_bif_succ
,xin_bif_system
,xin_bif_tally
,xin_bif_tanf
,xin_bif_threadid
,xin_bif_tiny
,xin_bif_trim
,xin_bif_type
,xin_bif_unallocated
,xin_bif_unsigned
,xin_bif_uppercase
,xin_bif_valid
,xin_bif_validddate
,xin_bif_varglist
,xin_bif_vargsize
,xin_bif_verifyr
,xin_bif_wchar
,xin_bif_wcharval
,xin_bif_weekday
,xin_bif_which
,xin_bif_widechar
,xin_bif_wlow
,xin_bif_xmlchar
,xin_bif_y4date
,xin_bif_y4julian
,xin_bif_y4year
) prec(16) unsigned;
```

図 117. *xin_Bif_Kind* の宣言 (5/5)

属性フラグには、コンパイラーがすべてのデフォルトを適用した後の属性が反映されることにも注意してください。したがって、例えば、どの数値変数 (数値 PICTURE 変数を含む) にも、REAL または COMPLEX のどちらかの属性フラグが設定されることになります。

SYSADATA 構文情報について

XINFO コンパイル時オプションの SYN サブオプションを指定すると、コンパイラーは、MSG および SYM サブオプションの場合に生成されるレコードに加えて、以下の情報が入った SYSADATA ファイルを生成します。

- ソース・レコード
- トークン・レコード
- 構文レコード

ソース・レコード

各ソース・レコードでは、ソース ID と呼ばれる 1 つの番号が割り当てられます。後のレコードでこのレコードに記述されたソース行を参照するときは、この番号が使用されます。行は、PL/I 1 次ソース・ファイルまたは INCLUDE されたファイルの行の場合があります。行は、このレコードのソース・ファイル ID と行番号のフィールドで示されます。このレコードの残りの部分は、ソース行の実データを保持します。

```
Dcl
  1 Xin_Src      Based( null() ), /* source record          */
    /*                                     */
    2 Xin_Src_Hdr /* standard header      */
      like Xin_Hdr, /*                                     */
    /*                                     */
    2 Xin_Src_File_Id /* file id                */
      fixed bin(32) unsigned, /*                                     */
    /*                                     */
    2 Xin_Src_Line_No /* line no within file    */
      fixed bin(32) unsigned, /*                                     */
    /*                                     */
    2 Xin_Src_Id      /* id for this source record */
      fixed bin(32) unsigned, /*                                     */
    /*                                     */
    2 Xin_Src_Length  /* length of text           */
      fixed bin(16) unsigned, /*                                     */
    /*                                     */
    2 Xin_Src_Text    /* actual text              */
      char( 137 refer(xin_Src_Length) );
```

図 118. ソース・レコードの宣言

トークン・レコード

各トークン・レコードでは、トークン索引と呼ばれる 1 つの番号が割り当てられます。後のレコードで PL/I コンパイラーに認識されるトークンを参照するときは、この番号が使用されます。また、このレコードは、トークンのタイプに加えて、ト

クンの開始と終了それぞれの列と行を示します。

```
Dcl
 1 Xin_Tok      Based( null() ), /* token record          */
                                /*                          */
  2 Xin_Tok_Hdr /* standard header      */
    like Xin_Hdr, /*                          */
                                /*                          */
  2 Xin_Tok_Inx /* adata index for token */
    fixed bin(32) unsigned, /*                          */
                                /*                          */
  2 Xin_Tok_Begin_Line /* starting line no within file */
    fixed bin(32) unsigned, /*                          */
                                /*                          */
  2 Xin_Tok_End_Line_Offset /* offset of end line from first */
    fixed bin(16) unsigned, /*                          */
                                /*                          */
  2 Xin_Tok_Kind_Value /* token kind          */
    ordinal xin_Tok_Kind, /*                          */
                                /*                          */
  2 Xin_Tok_Rsrvd /* reserved            */
    fixed bin(8) unsigned, /*                          */
                                /*                          */
  2 Xin_Tok_Begin_Col /* starting column     */
    fixed bin(16) unsigned, /*                          */
                                /*                          */
  2 Xin_Tok_End_Col /* ending column       */
    fixed bin(16) unsigned; /*
```

図 119. トークン・レコードの宣言

序数 `xin_Tok_Kind` は、トークン・レコードのタイプを示します。

```
Define
ordinal
xin_Tok_Kind
(
  xin_Tok_Kind_Unset
  ,xin_Tok_Kind_Lexeme
  ,xin_Tok_Kind_Comment
  ,xin_Tok_Kind_Literal
  ,xin_Tok_Kind_Identifier
  ,xin_Tok_Kind_Keyword
) prec(8) unsigned;
```

図 120. トークン・レコードの種類の宣言

構文レコード

各構文レコードでは、ノード ID と呼ばれる 1 つの番号が割り当てられます。後のレコードで他の構文レコードを参照するときは、この番号が使用されます。

最初の構文レコードの種類は `xin_Syn_Kind_Package` です。コンパイル単位の中にプロシージャールがある場合、このレコードの子ノードが、プロシージャールのうちの最

初のものを指します。次に親ノード、兄弟ノード、および子ノードに基づいて、コンパイル単位に含まれるすべてのプロシージャと開始ブロックの特有の関係が 1 つのマッピングに設定されます。

次の単純なプログラムで考えてみます。

```
a: proc;  
  call b;  
  call c;  
b: proc;  
end b;  
c: proc;  
  call d;  
  d: proc;  
  end d;  
end c;  
end a;
```

このプログラムのブロックに割り当てられるノード索引は、次のようになります。

symbol	index	sibling	parent	child
----	-----	-----	-----	-----
-	1	0	0	2
a	2	0	1	3
b	3	4	2	0
c	4	0	2	5
d	5	0	4	0

図 121. プログラムのブロックに割り当てられるノード索引

```

Dcl
1 Xin_Syn      Based( null() ), /* syntax record          */
/* standard header          */
2 Xin_Syn_Hdr  like Xin_Hdr, /* node id              */
/* node id              */
2 Xin_Syn_Node_Id fixed bin(32) unsigned, /* node type          */
/* node type          */
2 Xin_Syn_Node_Kind ordinal xin_syn_kind, /* node sub type      */
/* node sub type      */
2 Xin_Syn_Node_Exp_Kind ordinal xin_exp_kind, /* reserved          */
/* reserved          */
2 *           fixed bin(16) unsigned, /* node id of parent  */
/* node id of parent  */
2 Xin_Syn_Parent_Node_Id fixed bin(32) unsigned, /* node id of sibling  */
/* node id of sibling  */
2 Xin_Syn_Sibling_Node_Id fixed bin(32) unsigned, /* node id of child   */
/* node id of child   */
2 Xin_Syn_Child_Node_Id fixed bin(32) unsigned, /* id of first spanned token */
/* id of first spanned token */
2 xin_Syn_First_Tok fixed bin(32) unsigned, /* id of last spanned token */
/* id of last spanned token */
2 xin_Syn_Last_Tok fixed bin(32) unsigned,

```

図 122. 構文レコードの宣言 (1/3)

```

2 * union,                                /* qualifier for node */
3 Xin_Syn_Int_Value                       /* used if int */
    fixed bin(31),                        /*
3 Xin_Syn_Literal_Id                     /* used if name, number, picture */
    fixed bin(31),                        /*
3 Xin_Syn_Node_Lex                       /* used if lexeme, assignment, */
    ordinal xin_Lex_kind,                /* infix_op, prefix_op */
3 Xin_Syn_Node_Voc                       /* used if keyword, end_for_do */
    ordinal xin_Voc_kind,                /*
3 Xin_Syn_Block_Node                     /* used if call_begin */
    fixed bin(31),                       /* to hold node of begin block */
3 Xin_Syn_Bif_Id                         /* used if bif_rfrnc */
    fixed bin(32) unsigned,              /*
3 Xin_Syn_Sym_Id                         /* used if label, unsub_rfrnc, */
    fixed bin(32) unsigned,              /* subscripted_rfrnc */
3 Xin_Syn_Proc_Data,                     /* used if package, proc or begin */
4 Xin_Syn_First_Sym                     /* id of first contained sym */
    fixed bin(32) unsigned,              /*
4 Xin_Syn_Block_Sym                     /* id of sym for this block */
    fixed bin(32) unsigned,              /*

```

図 122. 構文レコードの宣言 (2/3)

```

3 Xin_Syn_Number_Data,      /* used if number          */
                             /* id of literal           */
4 Xin_Syn_Number_Id         /* type                     */
    fixed bin(32) unsigned, /* ordinal xin_Number_Kind, */
                             /* precision                */
4 Xin_Syn_Number_Prec       /* scale factor             */
    fixed bin(8) unsigned,  /* bytes it would occupy    */
                             /* in its internal form     */
4 Xin_Syn_Number_Scale     /* used if char_string,    */
    fixed bin(7) signed,    /* bit_string, graphic_string */
                             /* id of literal            */
4 Xin_Syn_String_Id        /* string length in its units */
    fixed bin(32) unsigned, /* used if stmt             */
                             /* file id                   */
3 Xin_Syn_String_Data,     /* line no within file      */
                             /* char(0);                  */
4 Xin_Syn_File_Id          /*                          */
    fixed bin(32) unsigned, /*                          */
                             /*                          */
4 Xin_Syn_Line_No          /*                          */
    fixed bin(32) unsigned, /*                          */
                             /*                          */
2 *                          /*                          */
    char(0);               /*                          */

```

図 122. 構文レコードの宣言 (3/3)

序数 `xin_Syn_Kind` は、構文レコードのタイプを示します。

```

Define
ordinal
xin_Syn_Kind
(
xin_Syn_Kind_Unset
,xin_Syn_Kind_Lexeme
,xin_Syn_Kind_Asterisk
,xin_Syn_Kind_Int
,xin_Syn_Kind_Name
,xin_Syn_Kind_Expression
,xin_Syn_Kind_Parenthesized_Expr
,xin_Syn_Kind_Argument_List
,xin_Syn_Kind_Keyword
,xin_Syn_Kind_Proc_Stmt
,xin_Syn_Kind_Begin_Stmt
,xin_Syn_Kind_Stmt
,xin_Syn_Kind_Substmt
,xin_Syn_Kind_Label
,xin_Syn_Kind_Invoke_Begin
,xin_Syn_Kind_Assignment
,xin_Syn_Kind_Assignment_Byname
,xin_Syn_Kind_Do_Fragment
,xin_Syn_Kind_Keyed_List
,xin_Syn_Kind_Iteration_Factor
,xin_Syn_Kind_If_Clause
,xin_Syn_Kind_Else_Clause
,xin_Syn_Kind_Do_Stmt
,xin_Syn_Kind_Select_Stmt
,xin_Syn_Kind_When_Stmt
,xin_Syn_Kind_Otherwise_Stmt
,xin_Syn_Kind_Procedure
,xin_Syn_Kind_Package
,xin_Syn_Kind_Begin_Block
,xin_Syn_Kind_Picture
,xin_Syn_Kind_Raw_Rfrnc
,xin_Syn_Kind_Generic_Desc
) prec(8) unsigned;

```

図 123. 構文レコードの種類の宣言

次の単純なプログラムで考えてみます。

```

a: proc(x);
  dcl x char(8);
  x = substr(datetime(),1,8);
end;

```

このプログラムのブロックに割り当てられるノード索引は、次のようになります。

node_kind	index	sibling	parent	child
package	1	0	0	2
procedure	2	0	1	0
expression	3	0	0	0
stmt	4	5	2	6
stmt	5	10	2	11
label	6	7	4	0
keyword	7	8	4	0
expression	8	9	4	0
lexeme	9	0	4	0
stmt	10	0	2	18
assignment	11	12	5	13
lexeme	12	0	5	0
expression	13	14	11	0
expression	14	0	11	15
expression	15	16	14	0
expression	16	17	14	0
expression	17	0	14	0
keyword	18	19	10	0
lexeme	19	0	10	0

図 124. プログラムの構文レコードに割り当てられるノード索引

プロシーチャー・レコードには、ENTRY A のシンボル・レコードの ID が含まれます (block_sym フィールド内)。このシンボル・レコードには、そのプロシーチャーの最初のステートメントのノード ID が含まれます (first_stmt_id フィールド内)。

ステートメント・レコードでは、次の点に注意してください。

- 兄弟ノード ID は、次のステートメント・レコードを指します (ある場合)。
- 子ノード ID は、そのステートメント・レコードの最初のエレメントを指します。

PROCEDURE ステートメントのレコードは、次の 4 つのレコードから成ります。

- ラベル・レコード。
- キーワード・レコード (PROCEDURE キーワード用)。
- 式レコード (パラメーター X 用)。これには、式の種類 unsub_rfrnc とシンボル X の sym_id が含まれます。
- 字句レコード (セミコロン用)。

割り当てステートメントのレコードは、次の 2 つのレコードから成ります。

- 次の 2 つの子を持つ割り当てレコード。
 - 式レコード (ターゲット X 用)。これには、式の種類 unsub_rfrnc とシンボル X の sym_id が含まれます。
 - 式レコード (ソース用)。これには、式の種類 builtin_rfrnc とシンボル SUBSTR の sym_id が含まれます。このレコード自体が、次の 3 つの子を持ちます。
 - 式レコード (最初の引数用)。これには、式の種類 builtin_rfrnc とシンボル DATETIME の sym_id が含まれます。

- 式レコード (2 番目の引数用)。これには、式の種類 `number` と値 1 の `literal_id` が含まれます。
- 式レコード (3 番目の引数用)。これには、式の種類 `number` と値 8 の `literal_id` が含まれます。
- 字句レコード (セミコロン用)。

END ステートメントのレコードは、次の 2 つのレコードから成ります。

- キーワード・レコード (END キーワード用)。
- 字句レコード (セミコロン用)。

序数 `xin_Exp_Kind` は、式が記述された構文レコードの式のタイプを示します。このレコードの中には、子ノードの数がゼロでないものもあります。例えば、次のような場合です。

- 挿入演算子 (減算を表すマイナスなど) は、その左オペランドが記述された子ノードを持ちます (このオペランドの兄弟ノードに、右演算子が記述されます)。
- 接頭演算子 (否定を表すマイナスなど) は、そのオペランドが記述された子ノードを持ちます。

```

Define
ordinal
xin_Exp_Kind
(
xin_Exp_Kind_Unset
,xin_Exp_Kind_Bit_String
,xin_Exp_Kind_Char_String
,xin_Exp_Kind_Graphic_String
,xin_Exp_Kind_Number
,xin_Exp_Kind_Infix_Op
,xin_Exp_Kind_Prefix_Op
,xin_Exp_Kind_Builtin_Rfrnc
,xin_Exp_Kind_Entry_Rfrnc
,xin_Exp_Kind_Qualified_Rfrnc
,xin_Exp_Kind_Unsub_Rfrnc
,xin_Exp_Kind_Subscripted_Rfrnc
,xin_Exp_Kind_Type_Func
,xin_Exp_Kind_Widechar_String
) prec(8) unsigned;

```

図 125. 式の種類の宣言

序数 `xin_Number_Kind` は、数値が記述された構文レコードの数値のタイプを示します。

```
Define
ordinal
xin_Number_Kind
(  xin_Number_Kind_Unset
  ,xin_Number_Kind_Real_Fixed_Bin
  ,xin_Number_Kind_Real_Fixed_Dec
  ,xin_Number_Kind_Real_Float_Bin
  ,xin_Number_Kind_Real_Float_Dec
  ,xin_Number_Kind_Cplx_Fixed_Bin
  ,xin_Number_Kind_Cplx_Fixed_Dec
  ,xin_Number_Kind_Cplx_Float_Bin
  ,xin_Number_Kind_Cplx_Float_Dec
) prec(8) unsigned;
```

図 126. 数値の種類の宣言

序数 `xin_Lex_Kind` は、字句単位が記述された構文レコードの字句のタイプを示します。この序数名のうち、

- 「vrule」は、例えば「or」記号として使用される「垂直罫線」を意味します。
- 「dbl」は「二重」を意味します。したがって `dbl_Vrule` は、例えば「連結」記号として使用される二重垂直罫線です。

```
Define
ordinal
xin_Lex_Kind
(  xin_Lex_Undefined
, xin_Lex_Period
, xin_Lex_Colon
, xin_Lex_Semicolon
, xin_Lex_Lparen
, xin_Lex_Rparen
, xin_Lex_Comma
, xin_Lex_Equals
, xin_Lex_Gt
, xin_Lex_Ge
, xin_Lex_Lt
, xin_Lex_Le
, xin_Lex_Ne
, xin_Lex_Lctr
, xin_Lex_Star
, xin_Lex_Dbl_Colon
, xin_Lex_Not
, xin_Lex_Vrule
, xin_Lex_Dbl_Vrule
, xin_Lex_And
, xin_Lex_Dbl_Star
, xin_Lex_Plus
, xin_Lex_Minus
, xin_Lex_Slash
, xin_Lex_Equals_Gt
, xin_Lex_Lparen_Colon
, xin_Lex_Colon_Rparen
, xin_Lex_Plus_Equals
, xin_Lex_Minus_Equals
, xin_Lex_Star_Equals
, xin_Lex_Slash_Equals
, xin_Lex_Vrule_Equals
, xin_Lex_And_Equals
, xin_Lex_Dbl_Star_Equals
, xin_Lex_Dbl_Vrule_Equals
, xin_Lex_Dbl_Slash
) unsigned prec(16);
```

図 127. 字句の種類宣言

序数 `xin_Voc_Kind` は、コンパイラの「語彙」から項目が記述された構文レコードのキーワードを示します。

```
Define
ordinal
xin_Voc_Kind
(  xin_Voc_Undefined
, xin_Voc_a
, xin_Voc_abnormal
, xin_Voc_act
, xin_Voc_activate
, xin_Voc_adata
, xin_Voc_addbuff
, xin_Voc_aggregate
, xin_Voc_aix
, xin_Voc_alias
, xin_Voc_alien
, xin_Voc_aligned
, xin_Voc_all
, xin_Voc_alloc
, xin_Voc_allocate
, xin_Voc_anno
, xin_Voc_ans
, xin_Voc_any
, xin_Voc_anycond
, xin_Voc_anycondition
, xin_Voc_area
, xin_Voc_as
, xin_Voc_ascii
, xin_Voc_asgn
, xin_Voc_asm
, xin_Voc_asmtcli
, xin_Voc_assembler
, xin_Voc_assignable
, xin_Voc_attach
, xin_Voc_attention
, xin_Voc_attn
, xin_Voc_attribute
, xin_Voc_attributes
, xin_Voc_auto
, xin_Voc_automatic
, xin_Voc_b
, xin_Voc_backwards
, xin_Voc_based
, xin_Voc_begin
, xin_Voc_beta
, xin_Voc_bigendian
, xin_Voc_bin
, xin_Voc_binary
, xin_Voc_bind
, xin_Voc_bit
, xin_Voc_bkwd
, xin_Voc_blksize
, xin_Voc_block
, xin_Voc_buf
```

図 128. 語彙の種類宣言 (1/15)

```
,xin_Voc_buffered
,xin_Voc_buffers
,xin_Voc_bufnd
,xin_Voc_bufni
,xin_Voc_bufoff
,xin_Voc_bufsp
,xin_Voc_build
,xin_Voc_builtin
,xin_Voc_by
,xin_Voc_byaddr
,xin_Voc_byname
,xin_Voc_byvalue
,xin_Voc_c
,xin_Voc_call
,xin_Voc_cdecl
,xin_Voc_cdecl16
,xin_Voc_cee
,xin_Voc_ceetdli
,xin_Voc_cell
,xin_Voc_char
,xin_Voc_character
,xin_Voc_charg
,xin_Voc_chargraphic
,xin_Voc_charset
,xin_Voc_check
,xin_Voc_cics
,xin_Voc_class
,xin_Voc_close
,xin_Voc_cmp
,xin_Voc_cmpat
,xin_Voc_cms
,xin_Voc_cmstp1
,xin_Voc_cobol
,xin_Voc_col
,xin_Voc_column
,xin_Voc_compile
,xin_Voc_complex
,xin_Voc_cond
,xin_Voc_condition
,xin_Voc_conn
,xin_Voc_connected
,xin_Voc_consecutive
,xin_Voc_constant
,xin_Voc_control
,xin_Voc_controlled
,xin_Voc_conv
,xin_Voc_conversion
,xin_Voc_copy
,xin_Voc_count
,xin_Voc_cplx
```

図 128. 語彙の種類の宣言 (2/15)

```
,xin_Voc_create
,xin_Voc_cs
,xin_Voc_ct
,xin_Voc_ctl
,xin_Voc_ctl360
,xin_Voc_ctlasa
,xin_Voc_currency
,xin_Voc_d
,xin_Voc_data
,xin_Voc_dataonly
,xin_Voc_db
,xin_Voc_dcl
,xin_Voc_deact
,xin_Voc_deactivate
,xin_Voc_debug
,xin_Voc_dec
,xin_Voc_decimal
,xin_Voc_deck
,xin_Voc_declare
,xin_Voc_def
,xin_Voc_default
,xin_Voc_define
,xin_Voc_defined
,xin_Voc_defines
,xin_Voc_delay
,xin_Voc_delete
,xin_Voc_desclist
,xin_Voc_desclocator
,xin_Voc_descriptor
,xin_Voc_descriptors
,xin_Voc_detach
,xin_Voc_dft
,xin_Voc_dim
,xin_Voc_dimension
,xin_Voc_direct
,xin_Voc_directed
,xin_Voc_display
,xin_Voc_dli
,xin_Voc_dllinit
,xin_Voc_do
,xin_Voc_downthru
,xin_Voc_dummydesc
,xin_Voc_duplicate
,xin_Voc_e
,xin_Voc_ebcdic
,xin_Voc_edit
,xin_Voc_alpha
,xin_Voc_else
,xin_Voc_emulate
,xin_Voc_enclave
```

図 128. 語彙の種類の宣言 (3/15)

```
,xin_Voc_end
,xin_Voc_endf
,xin_Voc_endfile
,xin_Voc_endif
,xin_Voc_endp
,xin_Voc_endpage
,xin_Voc_entry
,xin_Voc_enu
,xin_Voc_env
,xin_Voc_environment
,xin_Voc_error
,xin_Voc_esd
,xin_Voc_evendec
,xin_Voc_event
,xin_Voc_exclusive
,xin_Voc_exec
,xin_Voc_execops
,xin_Voc_execute
,xin_Voc_exit
,xin_Voc_exports
,xin_Voc_ext
,xin_Voc_extchk
,xin_Voc_external
,xin_Voc_externalonly
,xin_Voc_extname
,xin_Voc_extonly
,xin_Voc_f
,xin_Voc_fastcall
,xin_Voc_fastcall16
,xin_Voc_fb
,xin_Voc_fbs
,xin_Voc_fetch
,xin_Voc_fetchable
,xin_Voc_file
,xin_Voc_finish
,xin_Voc_first
,xin_Voc_fixed
,xin_Voc_fixeddec
,xin_Voc_fixedoverflow
,xin_Voc_flag
,xin_Voc_float
,xin_Voc_flow
,xin_Voc_flush
,xin_Voc_fofl
,xin_Voc_forever
,xin_Voc_format
,xin_Voc_fortran
```

図 128. 語彙の種類の宣言 (4/15)

```
,xin_Voc_free
,xin_Voc_from
,xin_Voc_fromalien
,xin_Voc_fs
,xin_Voc_full
,xin_Voc_g
,xin_Voc_generic
,xin_Voc_genkey
,xin_Voc_get
,xin_Voc_gn
,xin_Voc_go
,xin_Voc_gonumber
,xin_Voc_gostmt
,xin_Voc_goto
,xin_Voc_gr
,xin_Voc_graphic
,xin_Voc_gs
,xin_Voc_halt
,xin_Voc_handle
,xin_Voc_hexadec
,xin_Voc_hexadecimal
,xin_Voc_i
,xin_Voc_ibm
,xin_Voc_ieee
,xin_Voc_if
,xin_Voc_ign
,xin_Voc_ignore
,xin_Voc_imp
,xin_Voc_impl
,xin_Voc_implicit
,xin_Voc_imported
,xin_Voc_imprecise
,xin_Voc_ims
,xin_Voc_in
,xin_Voc_inc
,xin_Voc_incafter
,xin_Voc_incdir
,xin_Voc_include
,xin_Voc_incpath
,xin_Voc_indexarea
,xin_Voc_indexed
,xin_Voc_inherits
,xin_Voc_init
,xin_Voc_initfill
,xin_Voc_initial
,xin_Voc_inline
,xin_Voc_inout
```

図 128. 語彙の種類宣言 (5/15)

```
,xin_Voc_input
,xin_Voc_insource
,xin_Voc_instance
,xin_Voc_int
,xin_Voc_inter
,xin_Voc_internal
,xin_Voc_interrupt
,xin_Voc_into
,xin_Voc_invalidop
,xin_Voc_ipa
,xin_Voc_irred
,xin_Voc_irreducible
,xin_Voc_is
,xin_Voc_iterate
,xin_Voc_itrace
,xin_Voc_jpn
,xin_Voc_k
,xin_Voc_key
,xin_Voc_keyed
,xin_Voc_keyfrom
,xin_Voc_keylength
,xin_Voc_keyloc
,xin_Voc_keyto
,xin_Voc_l
,xin_Voc_label
,xin_Voc_langlvl
,xin_Voc_last
,xin_Voc_laxconv
,xin_Voc_laxdcl
,xin_Voc_laxif
,xin_Voc_laxint
,xin_Voc_laxqual
,xin_Voc_lc
,xin_Voc_leave
,xin_Voc_library
,xin_Voc_libs
,xin_Voc_like
,xin_Voc_limited
,xin_Voc_limits
,xin_Voc_line
,xin_Voc_linecount
,xin_Voc_lineno
,xin_Voc_linesize
,xin_Voc_linkage
,xin_Voc_list
,xin_Voc_littleendian
,xin_Voc_lmessage
,xin_Voc_lmsg
,xin_Voc_local
,xin_Voc_localonly
```

図 128. 語彙の種類の宣言 (6/15)

```
,xin_Voc_locate
,xin_Voc_log
,xin_Voc_loop
,xin_Voc_lowerinc
,xin_Voc_lsfirst
,xin_Voc_m
,xin_Voc_macro
,xin_Voc_main
,xin_Voc_map
,xin_Voc_mar
,xin_Voc_margini
,xin_Voc_margins
,xin_Voc_mask
,xin_Voc_max
,xin_Voc_maxgen
,xin_Voc_maxmem
,xin_Voc_md
,xin_Voc_mdeck
,xin_Voc_member
,xin_Voc_metaclass
,xin_Voc_method
,xin_Voc_methods
,xin_Voc_mi
,xin_Voc_min
,xin_Voc_msfirst
,xin_Voc_msg
,xin_Voc_multi
,xin_Voc_mvs
,xin_Voc_n
,xin_Voc_na
,xin_Voc_nag
,xin_Voc_name
,xin_Voc_names
,xin_Voc_nan
,xin_Voc_native
,xin_Voc_nativeaddr
,xin_Voc_natlang
,xin_Voc_nc
,xin_Voc_ncp
,xin_Voc_nct
,xin_Voc_nd
,xin_Voc_nest
,xin_Voc_new
,xin_Voc_ngn
,xin_Voc_ngr
,xin_Voc_ngs
,xin_Voc_nign
,xin_Voc_nimp
,xin_Voc_nimpl
,xin_Voc_ninc
```

図 128. 語彙の種類 of 宣言 (7/15)

```
,xin_Voc_nint
,xin_Voc_nis
,xin_Voc_nm
,xin_Voc_nmd
,xin_Voc_nmi
,xin_Voc_nnum
,xin_Voc_noadata
,xin_Voc_noaggregate
,xin_Voc_noanno
,xin_Voc_noattributes
,xin_Voc_noauto
,xin_Voc_noautomatic
,xin_Voc_nobj
,xin_Voc_nobuild
,xin_Voc_nocee
,xin_Voc_nocharg
,xin_Voc_nochargraphic
,xin_Voc_nocheck
,xin_Voc_nocompile
,xin_Voc_noconv
,xin_Voc_noconversion
,xin_Voc_nocount
,xin_Voc_nodebug
,xin_Voc_nodeck
,xin_Voc_nodef
,xin_Voc_nodescriptor
,xin_Voc_nodescriptors
,xin_Voc_nodirected
,xin_Voc_nodli
,xin_Voc_nodllinit
,xin_Voc_nodummydesc
,xin_Voc_noduplicate
,xin_Voc_noemulate
,xin_Voc_noesd
,xin_Voc_noevendec
,xin_Voc_noexecops
,xin_Voc_noexit
,xin_Voc_noext
,xin_Voc_noextchk
,xin_Voc_nof
,xin_Voc_nofetchable
,xin_Voc_nofixedoverflow
,xin_Voc_noflow
,xin_Voc_nofofl
,xin_Voc_nofromalien
,xin_Voc_nogonumber
,xin_Voc_nogostmt
,xin_Voc_nographic
,xin_Voc_noignore
,xin_Voc_noimplicit
```

図 128. 語彙の種類 of 宣言 (8/15)

```
,xin_Voc_noimprecise
,xin_Voc_noinclude
,xin_Voc_noinitfill
,xin_Voc_noinline
,xin_Voc_noinsource
,xin_Voc_nointerrupt
,xin_Voc_noinvalidop
,xin_Voc_noipa
,xin_Voc_nolaxasgn
,xin_Voc_nolaxconv
,xin_Voc_nolaxdcl
,xin_Voc_nolaxif
,xin_Voc_nolaxint
,xin_Voc_nolaxqual
,xin_Voc_nolib
,xin_Voc_nolist
,xin_Voc_nolock
,xin_Voc_nolog
,xin_Voc_nomacro
,xin_Voc_nomap
,xin_Voc_nomapin
,xin_Voc_nomapout
,xin_Voc_nomargini
,xin_Voc_nomdeck
,xin_Voc_nomsg
,xin_Voc_nonasgn
,xin_Voc_nonassignable
,xin_Voc_nonconn
,xin_Voc_nonconnected
,xin_Voc_none
,xin_Voc_nonest
,xin_Voc_nonlocal
,xin_Voc_nonnative
,xin_Voc_nonnativeaddr
,xin_Voc_nonrecursive
,xin_Voc_nonumber
,xin_Voc_nonvar
,xin_Voc_nonvarying
,xin_Voc_noobject
,xin_Voc_nooffset
,xin_Voc_noofl
,xin_Voc_nooptimize
,xin_Voc_nooptions
,xin_Voc_nooverflow
,xin_Voc_nop
,xin_Voc_nopp
,xin_Voc_nopptrace
,xin_Voc_noprobe
,xin_Voc_noproceed
,xin_Voc_noprofile
```

図 128. 語彙の種類 of 宣言 (9/15)

```
,xin_Voc_nopt
,xin_Voc_norb
,xin_Voc_noreserve
,xin_Voc_noretcode
,xin_Voc_normal
,xin_Voc_norunops
,xin_Voc_noscheduler
,xin_Voc_nosemantic
,xin_Voc_nosequence
,xin_Voc_noshort
,xin_Voc_nosize
,xin_Voc_nosnap
,xin_Voc_nosource
,xin_Voc_nosprog
,xin_Voc_nostmt
,xin_Voc_nostorage
,xin_Voc_nostrg
,xin_Voc_nostringrange
,xin_Voc_nostringsize
,xin_Voc_nostrz
,xin_Voc_nosubrg
,xin_Voc_nosubscriptrang
,xin_Voc_nosym
,xin_Voc_nosyntax
,xin_Voc_not
,xin_Voc_noterminal
,xin_Voc_notest
,xin_Voc_notiled
,xin_Voc_notrace
,xin_Voc_noufl
,xin_Voc_nounderflow
,xin_Voc_nowcode
,xin_Voc_nowrite
,xin_Voc_noxref
,xin_Voc_nozdiv
,xin_Voc_nozerodivide
,xin_Voc_npro
,xin_Voc_ns
,xin_Voc_nsem
,xin_Voc_nseq
,xin_Voc_nstg
,xin_Voc_nsyn
,xin_Voc_nterm
,xin_Voc_null370
,xin_Voc_nullsys
,xin_Voc_num
,xin_Voc_number
,xin_Voc_nx
,xin_Voc_obj
,xin_Voc_object
,xin_Voc_of
,xin_Voc_offset
,xin_Voc_ofl
,xin_Voc_on
,xin_Voc_onproc
,xin_Voc_op
```

図 128. 語彙の種類宣言 (10/15)

```
,xin_Voc_open
,xin_Voc_opt
,xin_Voc_optimize
,xin_Voc_optional
,xin_Voc_options
,xin_Voc_optlink
,xin_Voc_or
,xin_Voc_order
,xin_Voc_ordinal
,xin_Voc_organization
,xin_Voc_os
,xin_Voc_os2
,xin_Voc_other
,xin_Voc_otherwise
,xin_Voc_out
,xin_Voc_output
,xin_Voc_overflow
,xin_Voc_overrides
,xin_Voc_owns
,xin_Voc_p
,xin_Voc_package
,xin_Voc_page
,xin_Voc_pagesize
,xin_Voc_parameter
,xin_Voc_parents
,xin_Voc_parm
,xin_Voc_pascal
,xin_Voc_pascal16
,xin_Voc_password
,xin_Voc_path
,xin_Voc_pending
,xin_Voc_pentium
,xin_Voc_pic
,xin_Voc_picture
,xin_Voc_plitdli
,xin_Voc_plitest
,xin_Voc_pointer
,xin_Voc_pos
,xin_Voc_position
,xin_Voc_pp
,xin_Voc_pptrace
,xin_Voc_prec
,xin_Voc_precision
,xin_Voc_prefix
,xin_Voc_preproc
,xin_Voc_preview
,xin_Voc_print
,xin_Voc_priority
,xin_Voc_private
,xin_Voc_pro
```

図 128. 語彙の種類 of 宣言 (11/15)

```
,xin_Voc_probe
,xin_Voc_proc
,xin_Voc_procedure
,xin_Voc_proceed
,xin_Voc_process
,xin_Voc_profile
,xin_Voc_protected
,xin_Voc_ptr
,xin_Voc_public
,xin_Voc_put
,xin_Voc_r
,xin_Voc_range
,xin_Voc_read
,xin_Voc_real
,xin_Voc_record
,xin_Voc_recsz
,xin_Voc_recursive
,xin_Voc_red
,xin_Voc_reducible
,xin_Voc_reentrant
,xin_Voc_refer
,xin_Voc_refine
,xin_Voc_regional
,xin_Voc_relative
,xin_Voc_release
,xin_Voc_renames
,xin_Voc_reorder
,xin_Voc_repeat
,xin_Voc_reply
,xin_Voc_reread
,xin_Voc_reserve
,xin_Voc_reserved
,xin_Voc_reserves
,xin_Voc_resignal
,xin_Voc_retcode
,xin_Voc_return
,xin_Voc_returns
,xin_Voc_reuse
,xin_Voc_revert
,xin_Voc_rewrite
,xin_Voc_rules
,xin_Voc_runops
,xin_Voc_s
,xin_Voc_s386
,xin_Voc_s486
,xin_Voc_saa
,xin_Voc_saa2
,xin_Voc_saa3
,xin_Voc_scalarvarying
,xin_Voc_scheduler
```

図 128. 語彙の種類宣言 (12/15)

```
,xin_Voc_segmented
,xin_Voc_select
,xin_Voc_sem
,xin_Voc_semantic
,xin_Voc_seq
,xin_Voc_seq1
,xin_Voc_sequence
,xin_Voc_sequential
,xin_Voc_set
,xin_Voc_short
,xin_Voc_signal
,xin_Voc_signed
,xin_Voc_single
,xin_Voc_sis
,xin_Voc_size
,xin_Voc_sizefrom
,xin_Voc_sizeto
,xin_Voc_skip
,xin_Voc_smessage
,xin_Voc_smsg
,xin_Voc_snap
,xin_Voc_source
,xin_Voc_spill
,xin_Voc_sprog
,xin_Voc_sql
,xin_Voc_static
,xin_Voc_stdcall
,xin_Voc_stg
,xin_Voc_stmt
,xin_Voc_stop
,xin_Voc_storage
,xin_Voc_stream
,xin_Voc_strg
,xin_Voc_string
,xin_Voc_stringrange
,xin_Voc_stringsize
,xin_Voc_struct
,xin_Voc_structure
,xin_Voc_strz
,xin_Voc_subrg
,xin_Voc_subscriptrange
,xin_Voc_suspend
,xin_Voc_sym
,xin_Voc_syn
,xin_Voc_syntax
,xin_Voc_sysin
,xin_Voc_sysparm
,xin_Voc_sysprint
,xin_Voc_system
,xin_Voc_sz
```

図 128. 語彙の種類 of 宣言 (13/15)

```
,xin_Voc_task
,xin_Voc_term
,xin_Voc_terminal
,xin_Voc_test
,xin_Voc_then
,xin_Voc_thread
,xin_Voc_tiled
,xin_Voc_time
,xin_Voc_title
,xin_Voc_to
,xin_Voc_total
,xin_Voc_tp
,xin_Voc_trace
,xin_Voc_transient
,xin_Voc_transmit
,xin_Voc_trkofl
,xin_Voc_tso
,xin_Voc_tstack
,xin_Voc_type
,xin_Voc_u
,xin_Voc_uen
,xin_Voc_ufl
,xin_Voc_unal
,xin_Voc_unaligned
,xin_Voc_unbuf
,xin_Voc_unbuffered
,xin_Voc_undefinedfile
,xin_Voc_underflow
,xin_Voc_undf
,xin_Voc_union
,xin_Voc_unlimited
,xin_Voc_unlock
,xin_Voc_unroll
,xin_Voc_unsigned
,xin_Voc_until
,xin_Voc_update
,xin_Voc_upperinc
,xin_Voc_upthru
,xin_Voc_v
,xin_Voc_v1
,xin_Voc_v2
,xin_Voc_value
,xin_Voc_valuelist
,xin_Voc_valuerange
,xin_Voc_var
,xin_Voc_variable
,xin_Voc_varying
,xin_Voc_varyingz
,xin_Voc_varz
,xin_Voc_vb
```

図 128. 語彙の種類の宣言 (14/15)

```
,xin_Voc_vbs  
,xin_Voc_virtual  
,xin_Voc_vs  
,xin_Voc_vsam  
,xin_Voc_w  
,xin_Voc_wait  
,xin_Voc_wcode  
,xin_Voc_when  
,xin_Voc_while  
,xin_Voc_windows  
,xin_Voc_winproc  
,xin_Voc_wkeep  
,xin_Voc_write  
,xin_Voc_x  
,xin_Voc_xchar  
,xin_Voc_xinfo  
,xin_Voc_xoptions  
,xin_Voc_xref  
,xin_Voc_zdiv  
,xin_Voc_zerodivide  
) unsigned prec(16);
```

図 128. 語彙の種類宣言 (15/15)

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711
東京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態を提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。

国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。

本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

商標

以下は、International Business Machines Corporation の米国およびその他の国における商標です。

IBM	IMS
IBM LOGO	IMS/ESA
ibm.com	Language Environment
AIX	MVS
CICS	OS/390
CICS/ESA	RACF
DB2	System/390
DFSMS	VisualAge
DFSORT	z/OS

Intel および Pentium は、Intel Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

Microsoft、Windows、および Windows NT は、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名などはそれぞれ各社の商標または登録商標です。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

参考文献

Enterprise PL/I 資料

- 「プログラミング・ガイド」、SC88-9123
- 「言語解説書」、SC88-9126
- 「メッセージおよびコード」、SC88-9127
- 「コンパイラーおよびランタイム・プログラム 移行ガイド」、GC88-9124

PL/I for MVS & VM

- 「導入およびカスタマイズ (MVS)」、SC88-7221
- 「言語解説書」、SC88-7219
- 「コンパイル時メッセージおよびコード」、SC88-7224
- 「診断の手引き」、SC88-7223
- 「移行の手引き」、SC88-7220
- 「プログラミングの手引き」、SC88-7218
- 「参照要約」、SX88-7011

z/OS 言語環境プログラム

- 「概念」、SA88-8555
- 「デバッグのガイド」、GA88-8548
- 「ランタイム・メッセージ」、SA88-8554
- 「カスタマイズ」、SA88-8552
- 「プログラミング・ガイド」、SA88-8549
- 「プログラミング・リファレンス」、SA88-8550
- 「ランタイム・アプリケーション マイグレーション・ガイド」、GA88-8553
- 「ILC (言語間通信) アプリケーションの作成」、SA88-8551

CICS Transaction Server

- 「アプリケーション・プログラミング・ガイド」、SC88-7689
- 「アプリケーション・プログラミング・リファレンス」、SC88-7690
- 「カスタマイズ・ガイド」、SC88-7686
- 「外部インターフェース・ガイド」、SD88-7026

DB2 UDB (OS/390 版および z/OS 版)

- 「管理ガイド」、SC88-8761
- 「DB2 入門」、SC88-8767
- 「アプリケーション・プログラミングおよび SQL ガイド」、SC88-8763
- 「コマンド解説書」、SC88-8764

「メッセージおよびコード」、GC88-8768

「SQL 解説書」、SC88-8772

DFSORT™

「アプリケーション・プログラミングの手引き」、SC88-7061

「導入およびカスタマイズ」、SC88-7163

IMS/ESA®

「アプリケーション・プログラミング: データベース管理プログラム」、SC88-7552

「Application Programming: Database Manager Summary」、SC26-8037

「アプリケーション・プログラミング: 設計の手引き」、SC88-7542

「アプリケーション・プログラミング: トランザクション管理プログラミング」、SC88-7553

「Application Programming: Transaction Manager Summary」、SC26-8038

「アプリケーション・プログラミング: EXEC DLI コマンド (CICS および IMS™)」、SC88-7554

「Application Programming: EXEC DLI Commands for CICS and IMS Summary」、SC26-8036

z/OS MVS

「JCL 解説書」、SA88-8569

「JCL ユーザーズ・ガイド」、SA88-8570

「システム・コマンド」、SA88-8593

z/OS UNIX システム・サービス

「z/OS UNIX システム・サービス コマンド解説書」、SA88-8641

「z/OS UNIX システム・サービス・プログラミング: アセンブラ呼び出し可能サービス 解説書」、SA88-8642

「z/OS UNIX システム・サービス ユーザーズ・ガイド」、SA88-8640

z/OS TSO/E

「コマンド解説書」、SA88-8628

「ユーザーズ・ガイド」、SA88-8638

z/Architecture

「Principles of Operation」、SA22-7832

Unicode® および文字表現

「z/OS Support for Unicode: Using Conversion Services」、SC33-7050

用語集

この用語集は、PL/I のすべてのプラットフォームとリリースで使用する用語を定義したものです。このマニュアルで使用されていない用語が含まれていることがあります。該当する用語が見つからない場合は、本書の索引を調べるか、「*IBM Dictionary of Computing*」、SC20-1699 を参照してください。

【ア行】

あいまい参照 (ambiguous reference). 参照時点で認識されている名前をただ 1 つだけ識別するためには修飾が不十分な参照。

アクセス (access). データを参照するかまたは取り出すこと。

アクティブ (active). 活動化から終了にいたるまでのブロックの状態。ソース・プログラム・テキスト中の対応する ID をプリプロセッサ変数やプリプロセッサ入り口名の値に置き換えることができるときの、その変数や入り口の状態。イベント変数が非同期操作に結び付けられている間に置かれている状態。タスク変数に関連するタスクが付加されるときにタスク変数が置かれている状態。タスクが終了する前に置かれている状態。

値参照 (value reference). データ項目の値を得るのに使用する参照。

アテンション (attention). タスクに割り込みが生じる原因となるような、タスクにとっては外部の事柄の発生。

暗黙オープン (implicit opening). OPEN ステートメント以外の入カステートメントまたは出カステートメントが原因で、ファイルがオープンされること。

暗黙処置 (implicit action). 使用可能な条件が生じたときに、その条件用に現在確立されている ON ユニットがない場合にとられる処置。ON ステートメント処置 (ON-statement action) と対比。

暗黙宣言 (implicit declaration). DECLARE ステートメント内で明示的に宣言されていないか、または内容に従って宣言されていない名前。

暗黙の (implicit). 明示指定のないまま取られる処置。

位置合わせ (alignment). 機械に依存する特定の境界 (例えば、フルワード境界またはハーフワード境界) に関連付けて、データ項目を保管すること。

イベント (event). 状況および完了を、関連したイベント変数から決定することのできるプログラムの活動。

イベント変数 (event variable). イベントと関連付けることができる EVENT 属性を持つ変数。その値は、処置が完了したかどうか、および完了の状況を示す。

入り口値 (entry value). 入り口定数または入り口変数によって表されるエントリー・ポイント。入り口値には、その入り口定数に関連した活動化環境が含まれる。

入り口参照 (entry reference). 入り口値を返す入り口定数、入り口変数参照、または関数参照。

入り口式 (entry expression). 評価されると入り口名を生じるような式。

入り口データ (entry data). プロシーチャーへのエントリー・ポイントを表すデータ項目。

入り口定数 (entry constant). PROCEDURE ステートメントのラベル接頭部 (入り口名)。ENTRY 属性を指定し、VARIABLE 属性を指定しないで名前を宣言すること。

入り口変数 (entry variable). 入り口値を割り当てる対象となりうる変数。これは、ENTRY 属性と VARIABLE 属性を両方とも持っている必要がある。

入り口名 (entry name). ENTRY 属性を持つものとして明示的または内容に従って宣言された ID (ただし、VARIABLE 属性が与えられていない場合に限る)。または、ENTRY 属性を暗黙指定された入り口変数の値を持った ID。

埋め込み (padding). スtringの長さを必要な長さまで拡張するために、Stringの右側に連結される、1 つ以上の文字、漢字、またはビット。構造体または共用体の中に挿入される、1 つ以上のバイトまたはビット。その構造、または共用体内の後続エレメントが正しい規定境界に位置合わせされるようにするためのもの。

英字 (alphabetic character). A から Z までの任意の英字と、#, \$, @ (これらのグラフィック表記は国によって異なる場合がある) の拡張英字。

英数字 (alphameric character). 英字または数字。

エクステント (extent). 配列の次元の境界、ストリング長、または区域サイズによって示される範囲。この区域がターゲット区域に割り当てられる場合は、ターゲット区域のサイズ。

エピローグ (epilogue). ブロックまたはタスクの終了時に自動的に生じる各種処理。

エレメント (element). 配列などのデータ項目の集まりとは対照的な、単一のデータ項目。スカラー項目。

エレメント名 (elementary name). 「基本エレメント (*base element*)」を参照。

演算子 (operator). 実行する演算を指定する記号。

演算式 (operational expression). 1 つ以上の演算子から成る式。

エントリー・ポイント (entry point). そこでプロシージャを呼び出すことができるプロシージャ内の 1 地点。「1 次エントリー・ポイント (*primary entry point*)」および「2 次エントリー・ポイント (*secondary entry point*)」も参照。

オープン (ファイルの) (opening (of a file)). ファイルをデータ・セットに関連付けること。

オブジェクト (object). 単一名で参照されるデータの集まり。

オフセット変数 (offset variable). OFFSET 属性を持ったロケータ変数のことであり、その値は、ストレージ内のある区域の先頭からの相対位置を識別する。

オペランド (operand). ID、定数、または式。式には演算子が、時には他のオペランドとともに使用される。

オン条件 (ON-condition). PL/I プログラムにおける、プログラム割り込みの原因となりうるオカレンス。予期しないエラーが検出されたり、予期できる出来事ではあるものの、予期しない時にそれが起きたときに発生する。

[力行]

介在添え字 (interleaved subscripts). 添え字付き修飾参照の最下位レベル以外のレベルに存在する添え字。

介在配列 (interleaved array). 非結合ストレージを参照する配列。

開始ブロック (begin-block). BEGIN ステートメントと END ステートメントによって区切られ、名前有効範囲

を形成するステートメントの集まり。開始ブロックの活動化は、条件が生じたために行われる (開始ブロックが ON ユニットへの処置指定である場合) か、または GOTO ステートメントの結果の分岐を含め、通常の制御の流れを介して行われる。

外部シンボル (external symbol). それ自身が定義されている制御セクションを除く制御セクション内で参照できる名前。

外部シンボル辞書 (External Symbol Dictionary (ESD)). オブジェクト・モジュール内で使われるすべての外部シンボルの一覧表。

外部プロシージャ (external procedure). 他のいずれのプロシージャにも組み込まれないプロシージャ。パッケージ内に入っていて同様にエクスポートされるレベル 2 のプロシージャ。

外部名 (external name). 有効範囲が必ずしも 1 つのブロックとその収容ブロックだけに限定されない (EXTERNAL 属性を持つ) 名前。

返される値 (returned value). 関数プロシージャから返される値。

拡張英字 (extended alphabet). A から Z までの大文字、小文字の英字、\$, @、および #、または NAMES コンパイラー・オプションで指定されたもの。

確立された処置 (established action). 条件が生じたときにとられる処置。「暗黙の処置 (*implicit action*)」および「ON ステートメント処置 (*ON-statement action*)」も参照。

下限 (lower bound). 配列次元の下限。

仮想起点 (virtual origin (VO)). すべてゼロの添え字を持った配列のエレメントを保持するための位置。このようなエレメントが配列内になれば、仮想起点は本来それが保持されるべき場所になる。

型変換 (conversion). ある 1 つの表現法から、一組の特定属性に合うよう別の表現法に値を変換すること。例えば、文字ストリングを FIXED BINARY (15,0) などの算術値に変換すること。

活動化 (プリプロセッサ変数またはプリプロセッサ・エントリー・ポイントの) (activate (a preprocessor variable or preprocessor entry point)). マクロ機能 ID を、それに後続するソース・コード内で置換可能にすること。%ACTIVATE ステートメントは、プリプロセッサ変数やプリプロセッサ・エントリー・ポイントを活動化する。

活動化 (ブロックの) (activate (a block)). ブロックの実行を開始すること。プロシーチャー・ブロックは、呼び出されるときに、活動化される。開始ブロックが活動化するの、分岐を含め、通常の制御の流れ内に現れたときである。パッケージを活動化することはできない。

仮引数 (dummy argument). 参照によって渡すことのできない引数の値を保持するため自動的に作成される一時記憶域。

環境 (活動化の) (environment (of an activation)). 収容ブロック内で宣言されたデータに関して、呼び出されたブロックと関連し、そのブロック内で使用される情報。

環境 (ラベル定数の) (environment (of a label constant)). ステートメント・ラベル定数への参照が適用されるブロックの個々の活動化の識別情報。この情報が決定されるのは、ステートメント・ラベル定数が、引数として渡されたり、またはステートメント・ラベル変数に割り当てられ、それが定数と一緒に渡されたり割り当てられたときである。

関数 (プロシーチャー) (function (procedure)). PROCEDURE ステートメント内に RETURNS オプションのあるプロシーチャー。RETURNS 属性を指定して宣言された名前。これは、関数参照内にその入り口名のうちの 1 つがあると呼び出され、スカラー値を参照点に返す。サブルーチン (subroutine) と対比。

関数参照 (function reference). 入り口定数または入り口変数のことで、このどちらも関数を表さなければならないが、その後に空と考えられる引数リストが続く。サブルーチン呼び出し (subroutine call) と対比。

完全修飾名 (fully-qualified name). 名前が参照するメンバーより上の階層順序内のすべての名前と、そのメンバー自身の名前が組み込まれている名前。

キー (key). 直接アクセス・データ・セット内のレコードを識別するデータ。「ソース・キー (source key)」および「記録済みキー (recorded key)」を参照。

キーワード (keyword). PL/I において定義されたコンテキスト内で使用されると特定の意味を持つ ID。

キーワード・ステートメント (keyword statement). ステートメントの機能を示すキーワードで始まる単純ステートメント。

疑似変数 (pseudovariable). ターゲット変数を指定するのに使用できるすべての組み込み関数の名前。これは通常、代入ステートメントの左側にある。

記述子 (descriptor). 区域サイズ、配列境界、またはストリング長などの変数に関する情報を保持する制御ブロック。

基数 (base). 算術値を表現するための数体系。

規定境界 (integral boundary). そこでデータを位置合わせすることができる任意の 8 ビット単位のバイト・マルチアドレス。通常はハーフワード、フルワード、またはダブルワード (2、4、または 8 バイトの長さの整数倍) 境界である。

基底付き参照 (based reference). 基底付きストレージ・クラスを持った参照。

基底付きストレージ割り振り (based storage allocation). 基底付き変数用のストレージの割り振り。

基底付き変数 (based variable). ストレージ・アドレスがロケーターによって与えられる変数。同一変数の複数の世代をアクセスすることができる。これは、ストレージ内の固定位置を識別しない。

起動 (invocation). プロシーチャーの活動化。

起動する (invoke). プロシーチャーを活動化すること。

基本エレメント (base element). それ自身は別の構造体や共用体ではない、構造体や共用体のメンバー。

基本項目 (base item). 定義変数を定義するための、自動、被制御、または静的変数、またはパラメーター。

境界 (bounds). 任意の配列次元の上限と下限。

共用体 (union). 同一のストレージを占有し、相互にオーバーレイしたデータ・エレメントの集まり。メンバーは構造体、共用体、基本変数、または配列のいずれであっても構わない。それらは、同一の属性を持っていないてもかまわない。

切り捨て (truncation). ターゲット変数のストリング長や精度が限度を超えたときに、データ項目の片方の端から 1 つ以上の数字文字、グラフィックス、またはビットを除去すること。

記録済みキー (recorded key). 直接アクセス・データ・セット内でレコードを識別する文字ストリングのことであり、そこでは文字ストリングそのものもデータの一部として記録される。

区切り文字 (break character). 下線記号 (_)。ID を読みやすくするために使用することができる。例えば、変数を OLDINVENTORYTOTAL とする代わりに OLD_INVENTORY_TOTAL と記述できる。

区切り文字 (delimiter). すべてのコメントと、パーセント記号、括弧、コンマ、ピリオド、セミコロン、コロン、割り当て記号、ブランク、ポインター、アスタリスク、および単一引用符。これらは ID、定数、ピクチャー指定、iSUB、およびキーワードの限界を定めるものとなる。

区切る (delimit). 1 つ以上の項目またはステートメントの前後を、文字またはキーワードで囲むこと。

組み込み関数 (built in function). SQRT (平方根) のような、言語が提供する定義済み関数。

組み込み関数参照 (built-in function reference). オプショナルの引数リストを持つ組み込み関数名。

組み込みサブルーチン (built-in subroutine). コンパイル時に定義され、CALL ステートメントによって呼び出される入り口名を持つサブルーチン。

組み込み名 (built-in name). 組み込みサブルーチンの入り口名。

グループ (group). より大きいプログラム単位に入っているステートメントの集まり。グループは、DO グループまたは選択グループのどちらかであるが、ON ユニットとしての場合を除き、単一ステートメントを使用できるところでは常に使用することができる。

クローズ (ファイルの) (closing (of a file)). ファイルをデータ・セットまたは装置と切り離すこと。

継承次元 (inherited dimension). 構造体、共用体、またはエレメントでの、収容構造から派生する次元。名前が配列ではないエレメントであれば、その次元全体が継承次元で構成される。名前が配列であるエレメントであれば、その次元は、継承次元および明示的に宣言された次元で構成される。1 つ以上の継承次元を持つ構造体を、非結合集合と呼ぶ。結合集合 (connected aggregate) と対比。

現行世代 (current generation). 変数名を参照して、現在使用できる自動変数または被制御変数の世代。

コード化算術データ (coded arithmetic data). 数値を表し、基数 (10 進数または 2 進数)、スケール (固定小数点または浮動小数点)、および精度 (個々に持ちうる桁数) を特徴とするデータ項目。このデータは、変換しなくても、算術計算用に受け入れることのできる形式で保管される。

合成演算子 (composite operator). <=、**、および /* などの特殊文字を複数含む演算子。

構造化 (structuring). メンバー数、配置順、属性、および論理レベルによって表現される構造階層。

構造式 (structure expression). 評価されると構造体の値セットを生成する式。

構造体 (structure). 必ずしも同じ属性を持たなくても差し支えないデータ項目の集まり。配列 (array) と対比。

構造体の配列 (array of structures). 次元属性を構造体名に与えて指定される、順番に並べられた同一構造体の集まり。

構造体メンバー (structure member). 「メンバー (member)」を参照。

固定小数点定数 (fixed-point constant). 「算術定数 (arithmetic constant)」を参照。

コメント (comment). 文書化のために使用され、/* および */ で区切られる、ゼロ以上の文字数の文字ストリング。

コンテキスト宣言 (contextual declaration). DECLARE ステートメントで明示的に宣言されていないが、その使用の前後関係から、特定の属性が ID に関連付けられるような ID の存在。

コンパイラー・オプション (compiler options). コンパイルの特定の面を制御するために指定されるキーワード。例えば、生成するオブジェクト・モジュールの特徴や、作成する印刷出力のタイプなどがある。

コンパイル時 (間) (compile time). 一般に、ソース・プログラムがオブジェクト・モジュールに変換されている時間。PL/I では、変更したい場合に、ソース・プログラムを変更して、オブジェクト・プログラムに変換し終わるまでに経過する時間。

[サ行]

再帰的プロシージャ (recursive procedure). そのプロシージャ自身からでも、または別のアクティブ・プロシージャからでも呼び出すことのできるプロシージャ。

再入可能プロシージャ (reentrant procedure). 複数のタスク、スレッド、またはプロセスから同時に活動化でき、しかもこれらのタスク、スレッド、およびプロセス間で相互に干渉が生じないプロシージャ。

サブタスク (subtask). 特定のタスクによって生成されるタスク、または特定のタスクから最後に生成されたタスクへの直接ライン内の任意のタスク。

サブルーチン (subroutine). PROCEDURE ステートメント内に RETURNS オプションのないプロシージャー。関数 (function) と対比。

サブルーチン呼び出し (subroutine call). 後に CALL ステートメント内にあるオプションの引数リストが付く、サブルーチンを表さなければならないエントリー参照。関数参照 (function reference) と対比。

算術演算子 (arithmetic operators). 接頭演算子の + と -、あるいは挿入演算子 + - * / ** のうちのいずれか。

算術データ (arithmetic data). 基数、スケール、モード、および精度の特性を持つデータ。コード化算術データとピクチャー数字データも含まれる。

算術定数 (arithmetic constant). 固定小数点定数または浮動小数点定数。大部分の算術定数には符号を付けることができるが、符号は定数の一部ではない。

算術比較 (arithmetic comparison). 数値の比較。「ビット比較 (bit comparison)」、「文字比較 (character comparison)」も参照。

算術変換 (arithmetic conversion). ある 1 つの算術表現から別の表現に値を変換すること。

参照 (reference). 明示宣言を生じることになる 1 つのコンテキスト内以外の名前の出現。

式 (expression). 値、値の配列、または一連の構造化値セットを表すのに、プログラム内で使われる表記。単独で使用される定数または参照、あるいは、定数または参照あるいはその両方を演算子と組み合わせたもの。

字句単位の (lexically). 単位を左から右への順序に扱うことに関連した用語。

次元属性 (dimension attribute). 配列の次元数を指定し、各次元の境界を示す属性。

自己定義データ (self-defining data). プログラム実行時に決定され、集合のメンバー内に保管される境界、長さ、およびサイズを持つデータ項目を含む集合。

指数文字 (exponent characters). 以下のピクチャー指定文字のこと。

1. K および E. 指数フィールドの先頭を示すため、浮動小数点ピクチャー指定内で使用される文字。
2. F. 10 進小数点をその想定位置から右方向へ (正定数の場合) かまたは左方向へ (負定数の場合) 移動するときに、小数部の桁数を示す整数を使って指定されるスケール因数文字。

実際の起点 (actual origin (AO)). 配列または構造体内の最初の項目の位置。

自動ストレージ割り振り (automatic storage allocation). 自動変数用のストレージ割り振り。

自動変数 (automatic variable). ブロックの起動時に自動的にストレージを割り振られ、そのブロックの終了時に自動的にそれを解除される変数。

シフト (shift). ストレージ内のデータを元の位置の左または右へ変更すること。

シフトアウト (shift-out). 2 バイト・ストリングの先頭でコンパイラーにシグナルを送るために使用される記号。

シフトイン (shift-in). 2 バイト・ストリングの終わりをコンパイラーに知らせるために使用される記号。

集合 (aggregate). 「データ集合」を参照。

集合式 (aggregate expression). 配列式、構造式、または共用体式のこと。

集合タイプ (aggregate type). どのデータ項目の場合も、それが構造物、共用体、または配列のいずれであるかの指定。

修飾名 (qualified name). 構造物メンバーまたは共用体メンバーの階層順序。ピリオドで結合されていて、構造物の中の名前を識別するのに使用される。どの名前にも添え字を付けることができる。

修正 (fix-up). コンパイル済みプログラムを実行可能にするために、コンパイル時にエラーを検出したあとでコンパイラーが実行する解決手段。

収容ブロック (containing block). 該当する宣言、ステートメント、プロシージャー、またはその他のソース・テキストを収容している、パッケージ、プロシージャー、または開始ブロック。

終了 (タスクの) (termination (of a task)). タスクへの制御の流れを停止すること。

終了 (ブロックの) (termination (of a block)). ブロックの実行が終了して、RETURN ステートメントまたは END ステートメントによって、制御がその起動側ブロックに戻るか、または GO TO ステートメントによって起動側ブロックまたは他のアクティブ・ブロックに制御が渡ること。

主プロシージャー (main procedure). OPTIONS (MAIN) 属性を持った PROCEDURE ステートメントの

ある外部プロシージャ。このプロシージャは、プログラム実行の最初のステップで自動的に呼び出される。

使用可能 (enabled). 条件により割り込みが生じて、該当する規定 ON ユニットが呼び出される条件の状態。

条件 (condition). エラー (オーバーフローなど) または予期される状況 (入力ファイルの終わりなど) のいずれかの例外的な状態。条件が発生する (検出される) と、その条件に対する規定のアクションが処理される。「確立された処置 (*established action*)」および「暗黙処置 (*implicit action*)」も参照。

上限 (upper bound). 配列次元の上限。

条件接頭語 (condition prefix). ステートメントの接頭部として付けられる、括弧で囲まれた 1 つ以上の条件名のリスト。条件接頭語は、指定した条件を使用可能にするか使用不能にするかを指定する。

条件名 (condition name). PL/I 定義またはプログラマー定義の条件の名前。

小構造 (minor structure). 別の構造体または共用体の中に組み込まれている構造体。小構造の名前は、1 よりも大きくかつ親構造体または親共用体よりも大きいレベル番号を指定して宣言される。

使用不可の (disabled). 割り込みが発生せず、規定の処置も取られないような事態になった状態。

商用文字 (commercial character).

- CR (貸方) ピクチャー指定文字。
- DB (借方) ピクチャー指定文字。

処置指定 (action specification). ON ステートメント内にある、ON ユニットまたは単一のキーワード SYSTEM。該当する条件が発生すれば、2 つのうちいずれかがとるべき処置を指定する。

数字 (digit). 0 から 9 までの文字の 1 つ。

数字データ (numeric-character data). 「10 進ピクチャー・データ (*decimal picture data*)」を参照。

数値ピクチャー・データ (numeric picture data). 算術値と文字値を持ったピクチャー・データ。このタイプのピクチャー・データは、'A' または 'X.' という文字を含むことはできない。

スカラー変数 (scalar variable). 構造、共用体、配列ではない変数。

スケール (scale). 1 つの数値表記体系であり、その算術値は固定小数点または浮動小数点で表現される。

スケール因数 (scale factor). 固定小数点数内の小数桁数の指定。

スケール因数 (scaling factor). 「スケール因数 (*scale factor*)」を参照。

ステートメント (statement). キーワード、区切り文字、ID、演算子、および定数から構成され、セミコロン (;) で終わる PL/I ステートメント。任意で、条件接頭語リストとラベルのリストを付けることができる。「キーワード・ステートメント (*keyword statement*)」、「代入ステートメント (*assignment statement*)」、および「ヌル・ステートメント (*null statement*)」も参照。

ステートメント本体 (statement body). ステートメント本体は、単純ステートメントまたは複合ステートメントのどちらでもかまわない。

ステートメント・ラベル (statement label). 「ラベル定数 (*label constant*)」を参照。

ストリーム指向データ伝送 (stream-oriented data transmission). 文字形式になった個々のデータ値の連続ストリームであるものとしてデータを扱って、データを伝送すること。レコード単位データ伝送 (*record-oriented data transmission*) と対比。

ストリング. 単一のデータ項目として処理される、連続した文字、グラフィックス、またはビットの列。

ストリング変数 (string variable).

BIT、CHARACTER、または GRAPHIC 属性を指定して宣言される変数。この変数の値は、ビット・ストリング、文字ストリング、または漢字ストリングのいずれでもかまわない。

制御セクション (control sections). オブジェクト・モジュール内のグループ化された機械命令。

制御の流れ (flow of control). 実行の連なり。

制御フォーマット項目 (control format item). ストリーム内または印刷ページの内での、あるデータ項目の位置付けを指定するために、編集指示伝送の中で使用される指定。

制御変数 (control variable). DO ステートメントの反復実行を制御するのに使用する変数。

制御文字 (control character). 特定コンテキスト内に存在することによって制御機能が指定される、文字セット内の文字。1 つの例としてファイルの終わり (EOF) マーカーがある。

制限付き式 (restricted expression). コンパイル時にコンパイラーによって評価されて定数を生じる式。このような式のオペランドは、定数、指定した定数、および制限付きの式になる。

整数 (integer). 符号を付けるか付けないかは任意の、10 進または 2 進小数点のない一連の数字、または一連のビット。通常は、FIXED BINARY (p,0) または FIXED DECIMAL (p,0) と記述される、符号を付けるか付けないかは任意の整数。

静的ストレージ割り振り (static storage allocation). 静的変数用のストレージの割り振り。

静的変数 (static variable). プログラム実行の開始前に割り振られ、その実行の継続時間中はその割り振りの変わらない変数。

精度 (precision). 固定小数点データ項目内にある桁数またはビット数、または、浮動小数点データ項目での最小確保有効数字 (指数は除く) の数。

世代 (変数の) (generation (of a variable)). 静的変数の割り振り、被制御変数または自動変数の特定の割り振り、または基底付き変数の特定のロケーター修飾で、または定義された変数がパラメーターで指示されるストレージ。

接頭演算子 (prefix operator). オペランドの前に置かれ、そのオペランドにだけ適用される演算子。接頭演算子には、プラス (+)、マイナス (-)、および not (¬) がある。

接頭部 (prefix). ステートメントの先頭に付けられるラベル、または 1 つ以上の条件名の括弧で囲まれたリスト。

ゼロ抑止文字 (zero-suppression characters). ピクチャー指定文字の Z と *。これは、対応する桁位置のゼロを抑止し、それぞれをブランクまたはアスタリスクで置き換えるのに使用する。

宣言 (declaration). ID を名前として確立し、その ID 用に一連の属性を (部分的または全体的に) 指定すること。特定名の属性のソース。

先行ゼロ (leading zeroes). 算術値としては意味のないゼロ。ある数値内で最初の非ゼロより左側にあるすべてのゼロ。

選択グループ (select-group). SELECT ステートメントと END ステートメントで区切られたステートメントの連なり。

選択文節 (selection clause). 選択グループの WHEN 文節または OTHERWISE 文節。

ソース (source). 問題データに変換されるデータ項目。

ソース変数 (source variable). 他の演算に使用されるが、その演算で変更されることのない変数。ターゲット変数 (target variable) と対比。

ソース・キー (source key). 直接アクセス・データ・セット内で個々のレコードを識別するため、レコード単位伝送ステートメント内で参照されるキー。

ソース・プログラム (source program). ソース・プログラム・プロセッサ、およびコンパイラーへの入力となるプログラム。

総称キー (generic key). キー・クラスを識別する文字ストリング。そのストリングで始まるキーはすべて、そのクラスのメンバーである。例えば、'ABCD'、'ABCE'、および 'ABDF'、という記録済みキーは、すべて総称キー 'A' および 'AB' で識別されるクラスのメンバーであり、最初の 2 つは、'ABC' というクラスのメンバーでもある。そして、これら 3 つの記録済みキーは、それぞれ 'ABCD'、'ABCE'、'ABDF' というクラスの固有のメンバーであると見なすことができる。

総称記述子 (generic descriptor). GENERIC 属性内で使用する記述子。

総称名 (generic name). 入り口名ファミリーの名前。総称名への参照は、呼び出し点にある引数リスト内の引数の属性に一致するパラメーター記述子を持った入り口名によって置き換えられる。

相対仮想起点 (relative virtual origin (RVO)). 配列の実際の原点から配列の仮想原点を引いたもの。

挿入演算子 (infix operator). 2 つのオペランド間にある演算子。

挿入点文字 (insertion point character). 関連データを文字ストリングへ割り当てるときに、指示位置に挿入されるピクチャー指定文字。入力のときに P フォーマット項目内で使用される挿入文字は、検査の目的で用いられる。

添え字 (subscript). 配列の次元内の位置を指定するための要素式。添え字がアスタリスクであれば、次元のすべてのエレメントを指定する。

添え字リスト (subscript list). 括弧に入れられた、1 つ以上の添え字のリスト。配列の各々の次元に対して 1 つの添え字が対応する。これらによって配列の単一エレメントまたはクロスセクションを一意的に識別する。

属性 (attribute). 表明された特性を記述するのに名前と関連付けた記述特性。式の計算の結果の特性を説明するために用いられる記述特性。

属性分配 (factoring). 1 つ以上の属性を、DECLARE ステートメント内の括弧で囲まれた名前リストに対して適用して、複数の名前に共通する属性を反復する必要をなくすこと。

[タ行]

ターゲット (target). データ項目 (ソース) が変換される属性。

ターゲット参照 (target reference). 受取側変数 (または受取側変数の一部) を指定する参照。

ターゲット変数 (target variable). 値が割り当てられる変数。

大構造 (major structure). レベル番号 1 を指定して宣言された名前を持つ構造。

代替属性 (alternative attribute). 属性グループから選択するファイル記述属性。何も指定しないと、デフォルトがとられる。追加属性 (*additive attribute*) と対比。

タイプ (type). データの世代、値、または項目に対して適用される一連のデータ属性とストレージ属性。

多重宣言 (multiple declaration). 同一ブロックに対して内部であり、別の修飾を持たない同一 ID の複数宣言。同一 ID の複数外部宣言。

タスク (task). 単一の制御の流れによる 1 つ以上のプロシージャーの実行。

タスクの生成 (attachment of a task). 呼び込まれたプロシージャー (およびこれが呼び出すプロシージャー) を、呼び出しプロシージャーの実行と一緒に、非同期で実行するために、プロシージャーを呼び出して別に制御の流れを確立すること。

タスク変数 (task variable). TASK 属性を持ち、その値がタスクの相対優先順位を示す変数。

タスク名 (task name). タスク変数を参照するのに使用される ID。

単純ステートメント (simple statement).

IF、ON、WHEN、および OTHERWISE 以外のステートメント。

単純パラメーター (simple parameter). ストレージ・クラス属性が指定されていないパラメーター。単純パラメ

ーターはどのストレージ・クラスの引数も表すことができるが、被制御引数の現行世代だけを表すことができる。

ダンプ (dump). エラーの原因のトレースなどの、プログラムが使用するストレージの一部または全部、または他のプログラム情報の印刷出力。

調節可能エクステンツ (adjustable extent). 関連変数の世代によって異なることのある境界 (配列の)、長さ (ストリングの)、またはサイズ (区域の)。調節可能エクステンツは、世代ごとに別々に評価される式またはアスタリスク (ただし、基底付き変数の場合は REFER オプション) で指定される。静的変数に使用することはできない。

追加属性 (additive attribute). デフォルトを持たず、必要であれば明示的に述べるか、または、明示的に述べられた別の属性で暗黙指定しなければならないファイル記述属性。代替属性 (*alternative attribute*) と対比。

データ (data). 処理に適合した形式の情報または値の表現。

データ項目 (data item). 単一の名前付きデータ単位。

データ指示伝送 (data-directed transmission). データを伝送するための、ストリーム指向伝送のタイプ。代入ステートメントに似ていて、name = constant の形式をとる。

データ指定 (data specification). 伝送モード (DATA、LIST、または EDIT) を指示し、さらにデータ・リストと、編集指示モードの場合はフォーマット・リストを含む、ストリーム指向伝送ステートメントの一部。

データ集合 (data aggregate). 異なったデータ項目の集まりであるデータ項目。

データ属性 (data attribute). FIXED BINARY などの、データ項目が表すデータのタイプを指定するキーワード。

データ伝送 (data transmission). データ・セットからプログラムへ、およびその逆に、データを転送すること。

データ・ストリーム (data stream). ストリーム指向伝送でデータ・セットから、またはデータ・セットへ、文字形式のデータ・エレメントの連続ストリームとして転送されるデータ。

データ・セット (data set). 単一のファイル名の参照によってアクセスすることができる、プログラムの外部にあるデータの集まり。参照されることが可能な装置。

データ・タイプ (data type). 一連のデータ属性。

データ・リスト (data list). ストリーム指向伝送における、GET および PUT ステートメント内で使用するデータ項目を括弧で囲んだリスト。フォーマット・リスト (*format list*) と対比。

定義された変数 (defined variable). 指定された基底付き変数用の一部または全部のストレージに関連付けられる変数。

定数 (constant). 名前が付いておらず、変更できない値を持った、算術またはストリング・データ項目。VALUE 属性を指定して宣言された ID。FILE 属性または ENTRY 属性を指定し、VARIABLE 属性を指定しないで宣言された ID。

定数参照 (constant reference). 対象として定数を持つ値参照。

デバッグ (debugging). プログラムからバグを除去する処理。

デフォルト (default). 指定がされていないときに、とられる値、属性、またはオプション。

同期 (synchronous). プログラムの順次実行での単一の制御の流れ。

特別言語文字 (extralingual character). 英数字にも特殊文字にも分類されない文字 (\$、@、および # など)。このグループには、NAMES コンパイラー・オプションで指定された文字も含まれる。

[ナ行]

内部プロシージャ (internal procedure). ブロックの中に組み込まれている別のプロシージャ。外部プロシージャ (*external procedure*) と対比。

内部ブロック (internal block). ブロックの中に組み込まれている別のブロック。

内部名 (internal name). 名前が宣言されたブロック内のみで認識されている名前、またそのブロック内に入っているブロックの中でも認識されている可能性もある名前。

入出力 (input/output). 補助メディアと主記憶装置との間でデータを転送すること。

認識された (名前に関する用語) (known (applied to a name)). 宣言された意味で認識されること。名前は、その有効範囲内で認識される。

ヌル・ステートメント (null statement). セミコロン記号 (;) のみの入ったステートメント。これは、何も処置はとられないことを示す。

ヌル・ストリング (null string). 長さゼロの文字ストリング、漢字ストリング、またはビット・ストリング。

ヌル・ロケータ値 (null locator value). 内部記憶域内のどの位置も識別できない特殊ロケータ値。これは、現在ロケータ変数がデータの世代を識別できないことを示すのに役立つ。

ネスト (nesting). 次のものの発生。

- ブロック内にある別のブロック。
- グループ内にある別のグループ。
- THEN 文節または ELSE 文節内の IF ステートメント。
- 関数参照の引数としての関数参照。
- FORMAT ステートメントのフォーマット・リスト内のリモート・フォーマット項目。
- パラメーター記述子リスト内の別のパラメーター記述子リスト。
- 1 つ以上の属性が分配されている括弧で囲まれた名前リスト内の属性の指定。

[ハ行]

配列 (array). 同じ属性を持ち、1 つ以上の次元別にグループ分けされた 1 つ以上のデータ・エレメントに、名前を付けて順番に並べた集合体。

配列式 (array expression). 評価されると値の配列が生成される式。

配列の共用体 (union of arrays). DIMENSION 属性を持った共用体。

配列のクロス・セクション (cross section of an array). 配列の少なくとも 1 つの次元のエクステントで表すことのできるエレメント。配列参照内に添え字の代わりにアスタリスクがあれば、それはその次元のエクステント全体を表す。

配列の構造体 (structure of arrays). 次元属性を持つ構造体。

配列変数 (array variable). 同じ属性を持っていないなければならないデータ項目の集合を表す変数。構造変数 (*structure variable*) と対比。

パック 10 進 (packed decimal). 固定小数点 10 進データ項目の内部表現。

パッケージ定数 (package constant). PACKAGE ステートメントのラベル接頭語。

バッファ (buffer). レコードが入力時に読み込まれ、レコードが出力時に書き出される、入出力操作に使用する中間記憶域。

パラメーター (parameter). PROCEDURE ステートメントの後に続くパラメーター・リスト中の名前。そのプロシーチャーが呼び出されれば、渡される引数を指定する。

パラメーター記述子 (parameter descriptor). ENTRY 属性指定内でパラメーター用に指定される一連の属性。

パラメーター記述子リスト (parameter descriptor list). ENTRY 属性指定内のすべてのパラメーター記述子のリスト。

パラメーター・リスト (parameter list). コンマで区切られ、プロシーチャー・ステートメント内のキーワード PROCEDURE の後に続くか、または ENTRY ステートメント内のキーワード ENTRY の後に続く、括弧で囲まれた 1 つ以上のパラメーターのリスト。このリストは、呼び出し時に渡される引数リストと対応する。

範囲 (デフォルト指定の) (range (of a default specification)). DEFAULT ステートメント内の属性を適用される ID またはパラメーター記述子のどちらか、またはこの両方のセット。

反復 DO グループ (iterative do-group). 制御変数または WHILE や UNTIL オプション、またはこの両方を指定した DO ステートメントを持つ DO グループ。

反復因数 (iteration factor). INITIAL 属性指定において、特定の値を使って初期化されることになっている配列の連続エレメント数を指定するための式。フォーマット・リストにおける、特定のフォーマット項目またはフォーマット項目のリストを連続して使用する回数を指定するための式。

反復因数 (repetition factor). 以下のものを指定する、括弧に入れられた符号なし整数。

1. 後続するストリング定数を繰り返す回数。
2. 後続するピクチャー文字を繰り返す回数。

反復指定 (repetitive specification). 1 つ以上のデータ項目伝送の被制御反復を指定するためのデータ・リストの 1 エレメントであり、通常は配列と同時に使用する。

非アクティブ (deactivated). ある ID の値で、ソース・プログラム・テキスト内のプリプロセッサ ID を置き換えることができない状態。アクティブ (active) と対比。

比較演算子 (comparison operator). 関係内の項目を相互に比較するよう指示するための算術、ストリング・ロケーター、または論理関係で使用される演算子。比較演算子は次のとおり。

- = (に等しい)
- > (より大)
- < (より小)
- >= (より大か等しい)
- <= (より小か等しい)
- ≠ (等しくない)
- ≠> (より大ではない)
- ≠< (より小ではない)

引数 (argument). サブルーチンまたは機能の呼び出しの一部である引数リスト内にある式。

引数リスト (argument list). コンマで区切られ、入り口名定数、入り口名変数、総称名、または組み込み関数名に続く、括弧で囲まれたゼロまたはそれ以上の引数のリスト。そのリストは、エントリー・ポイントのパラメーター・リストである。

ピクチャー指定 (picture specification). PICTURE 属性を指定した宣言内で、または P フォーマット項目内でピクチャー文字を使用して宣言されたデータ項目。

ピクチャー指定文字 (picture specification character). ピクチャー指定で使用するすべての文字。

ピクチャー・データ (picture data). 文字形式で表された、数値データ、文字データ、またはそれらの混合。

非結合ストレージ (nonconnected storage). 非結合データ項目が占有するストレージ。例えば、継承次元を持つ介在配列や構造体は、非結合ストレージ内にある。

被制御ストレージ割り振り (controlled storage allocation). 被制御変数用のストレージの割り振り。

被制御パラメーター (controlled parameter). DECLARE ステートメント内で CONTROLLED 属性を指定されるパラメーター。これは、CONTROLLED 属性を持った引数としか関連付けることはできない。

被制御変数 (controlled variable). 現行世代にだけアクセスすることができ、ALLOCATE と FREE ステートメントによって割り振りと解放が制御される変数。

ビット (bit). 0 または 1。コンピューター・ストレージの最小スペース量。

ビット値 (bit value). ビット・タイプを表す値。

ビット比較 (bit comparison). 2 進数字を、左から右へビットごとに比較すること。「算術比較 (*arithmetic comparison*)」、「文字比較 (*character comparison*)」も参照。

ビット・ストリング (bit string). ゼロ以上のビットで構成されたストリング。

ビット・ストリング演算子 (bit string operators). 論理演算子 NOT と排他 OR (\vee)、AND ($\&$)、および OR (\vee)。

ビット・ストリング定数 (bit string constant). 囲まれていて、接尾部 B が直後に付いた 2 進数字の連なり。文字定数 (*character constant*) と対比。単一引用符に囲まれ、後に接尾部 B4 が付いた 16 進数字の連なり。

非同期操作 (asynchronous operation). ステートメントの実行と、入出力操作が並行して行われること。各種タスクに複数の制御の流れを使った、プロシーチャーの並行実行。

評価 (evaluation). 単一の値、値の配列、または値の構造化値へ式を換算すること。

標準システム処置 (standard system action). 使用可能な条件のための ON ユニットがないときにその条件が発生した場合にとられる、言語で指定された処置。

標準デフォルト (値) (standard default). 属性またはオプションの指定がなく、適用できる DEFAULT ステートメントがない場合の、代替属性または代替オプション。

標準ファイル (standard file). GET ステートメントや PUT ステートメントで FILE オプションまたは STRING オプションがない場合に、PL/I が想定するファイル。SYSIN が標準入力ファイルであり、SYSPRINT が標準出力ファイルである。

ファイル (file). プログラムにおいて、単数または複数のデータ・セットを名前付きで表現したもの。ファイルは、オープンするごとに、単数または複数のデータ・セットに関連付けられる。

ファイル記述属性 (file description attribute). 各ファイル定数の個々の特性を記述したキーワード。「代替属性 (*alternative attribute*)」と「追加属性 (*additive attribute*)」も参照。

ファイル式 (file expression). 評価されるとファイル・タイプを生じる式。

ファイル定数 (file constant). FILE 属性を指定し、VARIABLE 属性を指定しないで宣言された名前。

ファイル変数 (file variable). ファイル定数を割り当てることのできる変数。この場合、ファイルは、FILE 属性と VARIABLE 属性を持っていなければならない、ファイル記述属性を持っていることはできない。

ファイル名 (file name). ファイル用に宣言された名前。

フィールド (データ・ストリーム中の) (field (in the data stream)). 単一データまたはスペーシング・フォーマット項目によって、幅 (文字数) を定義されるデータ・ストリームの部分。

フィールド (ピクチャー指定の) (field (of a picture specification)). 任意の文字ストリング・ピクチャー指定、または固定小数点数を記述した数字ピクチャー指定の部分 (または全部)。

フォーマット (format). ストリーム内のデータ項目の表現法を記述したり (データ・フォーマット項目)、またはストリーム内のデータ項目の個々の位置決めを記述する (制御フォーマット項目) ために編集指示データ伝送内で使用される仕様。

フォーマット定数 (format constant). FORMAT ステートメントでのラベル接頭部。

フォーマット・データ (format data). FORMAT 属性を指定された変数。

フォーマット・ラベル (format label). FORMAT ステートメントでのラベル接頭部。

フォーマット・リスト (format list). ストリーム指向伝送における外部メディアでのデータ項目のフォーマットを指定したリスト。データ・リスト (*data list*) と対比。

複合ステートメント (compound statement). 他のステートメントが含まれているステートメント。PL/I では、IF、ON、OTHERWISE、および WHEN だけが、複合ステートメントである。「ステートメント本体 (*statement body*)」を参照。

複合ネストの深さ (combined nesting depth). プログラム内の PROCEDURE/BEGIN/ON、DO、SELECT、および IF...THEN...ELSE によるネストのレベルをカウントして決定される、最も深いネスト・レベル。

複素数データ (complex data). おおのこの項目が実数部と虚数部で構成された算術データ。

含まれているブロック、宣言、またはソース・テキスト (contained block, declaration, or source text). 開始、プロシーチャー、またはパッケージのブロック内のすべてのブロック、プロシーチャー、ステートメント、宣

言、またはソース・テキスト。パッケージ、プロシージャ、および BEGIN ステートメントとそれに対応する END ステートメント全体は、ブロック内には含まれていない。

符号および通貨記号 (sign and currency symbol characters). ピクチャー指定文字。S、+、-、および \$ (または < と > で囲まれたその他の通貨記号)。

浮動小数点定数 (floating-point constant). 「算術定数 (arithmetic constant)」を参照。

部分修飾名 (partially-qualified name). 不完全な修飾名。これには名前が参照する構造メンバーまたは共用体メンバーより上の階層順序内にある名前うちの全部ではない 1 つ以上の名前、およびそれ自身のメンバー名が含まれる。

プリプロセッサ (preprocessor). コンパイルを実行する前に、ソース・プログラムを調べるためのプログラム。

プリプロセッサ・ステートメント (preprocessor statement). プリプロセッサがとる処置を指定するために、ソース・プログラム内に入れる特殊ステートメント。これは、プリプロセッサによって検出されると実行される。

プログラム (program). 1 つ以上の外部プロシージャまたはパッケージのセット。外部プロシージャのうちの 1 つは、PROCEDURE ステートメント内に OPTIONS(MAIN) 指定を持っていなければならない。

プログラム制御データ (program control data). PL/I プログラムの処理を制御するのに使用するための区域、ロケータ、ラベル、フォーマット、項目、およびファイルのデータ。

プロシージャ (procedure). PROCEDURE ステートメントと END ステートメントで区切られたステートメントの集まり。プロシージャとはプログラムまたはプログラムの一部であり、名前の有効範囲を区切り、そのプロシージャまたは入り口名の 1 つへの参照によって活動化される。「外部プロシージャ (external procedure)」および「内部プロシージャ (internal procedure)」も参照。

プロシージャ参照 (procedure reference). 入り口定数または入り口変数。この後に引数リストを続けることができる。プロシージャ参照は、CALL ステートメントや CALL オプションに入れることも、または関数参照として使用することもできる。

ブロック (block). その中で宣言された名前の有効範囲と、その名前用のストレージ割り振りを指定する、1 つ

の単位として処理される一連のステートメント。ブロックとしては、パッケージ、プロシージャ、または開始ブロックのいずれもありえる。

プロローグ (prologue). ブロックの起動時に自動的に生じる処理。

分離文字 (separator). 「区切り文字 (delimiter)」を参照。

編集指示伝送 (edit-directed transmission). データが連続した文字ストリームとして中にあり、関連データ・リストに対して行いたい編集を指定するにはフォーマット・リストを必要とするようなタイプのストリーム指向伝送。

変数 (variable). データを参照するのに使用され、値を割り当てる対象となりうる名前付きのエンティティ。その属性は一定のままであるが、場合に依じてそれぞれ異なる値を参照することができる。

変数参照 (variable reference). 変数全体またはその一部を指定する参照。

ポインタ (pointer). ストレージ内の位置を識別するための変数のタイプ。

ポインタ値 (pointer value). ポインタ型を識別する値。

ポインタ変数 (pointer variable). ポインタ値が入った POINTER 属性を持ったロケータ変数。

[マ行]

マルチタスキング (multitasking). 複数の PL/I プロシージャをプログラムが同時に実行できるようにする機能。

マルチプログラミング (multiprogramming). 単一の処理装置を使って、複数のプログラムを並行して処理するのに、計算機システムを使用すること。

マルチプロセッシング (multiprocessing). 複数のプログラムを同時に実行するために、複数の処理装置を備えた計算機システムを使用すること。

未定義 (undefined). ユーザーが行ってはならないことを示す。未定義機能が使用されると、PL/I 製品の個々のインプリメンテーションによって違った結果が出る可能性がある。このような場合、アプリケーション・プログラムはエラーとなる。

名 (name). 変数や定数にユーザーが与える ID。コンテキスト中に現れる、キーワードではない ID。場合によっては、ユーザー定義名とも呼ぶ。

明示宣言 (explicit declaration). ラベル接頭部として DECLARE ステートメント内、またはパラメーター・リスト内に ID (名前) を出すこと。暗黙宣言 (*implicit declaration*) と対比。

メンバー (member). 構造体または共用体の中の、構造体、共用体、あるいはエレメントの名前。ライブラリー内のデータ・セット。

モード (算術データの) (mode (of arithmetic data)). 算術データの属性。これは、実数 または複素数 のどちらかである。

文字ストリング定数 (character string constant). 単一引用符で囲まれる一連の文字。例えば、'Shakespeare's Hamlet:' など。

文字ストリング・ピクチャー・データ (character string picture data). 文字値だけを持ったピクチャー・データ。このタイプのピクチャー・データは、少なくとも 1 つの A または X ピクチャー指定文字を持っていなければならない。数値ピクチャー・データ (*numeric picture data*) と対比。

文字セット (character set). あらかじめ決められた文字の集まり。ASCII と EBCDIC を参照。

文字比較 (character comparison). 照合順序に従って、左から右へ文字単位で行われる比較。「算術比較 (*arithmetic comparison*)」、「ビット比較 (*bit comparison*)」も参照。

問題状態プログラム (problem-state program). オペレーティング・システムの問題プログラム状態内で稼働するプログラム。これには、入出力指示やその他の特権命令は入らない。

問題データ (problem data). コード化された算術データ、ビット・データ、文字データ、グラフィック・データ、およびピクチャー・データ。

[ヤ行]

有効範囲 (条件接頭語の) (scope (of a condition prefix)). 全体にわたって特定の条件接頭語が適用される、プログラムの部分。

有効範囲 (宣言の) (scope (of a declaration or name)). 全体にわたって特定名が認識されているプログラムの部分。

優先度 (priority). タスクに関連する値で、他のタスクに対するそのタスクの優先順位 (指名順位) を指定する。

要素式 (element expression). 評価されるとエレメント値を生じる式。

要素変数 (element variable). エレメントを表す変数。スカラー変数。

呼び出されたプロシージャ (invoked procedure). 活動化されているプロシージャ。

呼び出し (call). CALL ステートメントまたは CALL オプションを使用してサブルーチンを呼び出すこと。

呼び出し側ブロック (invoking block). プロシージャを活動化するブロック。

呼び出し点 (point of invocation). 呼び込まれたプロシージャへの参照が現れる呼び込み側ブロック内の地点。

予備ファイル (spill file). 一時作業ファイルとして使われる SYSUT1 という名前のデータ・セット。

[ラ行]

ライブラリー (library). メンバーと呼ばれるその他のデータ・セットを保管するのに使用できる MVS 区分データ・セット、または CMS MACLIB のこと。

ラベル (label). ステートメントの接頭部として付く名前。PROCEDURE ステートメント上の名前を、入り口定数と呼び、FORMAT ステートメント上の名前を、フォーマット定数と呼ぶ。その他の種類のステートメント上の名前を、ラベル定数と呼ぶ。LABEL 属性を持つデータ項目。

ラベル接頭部 (label prefix). ステートメントの接頭部として付けられたラベル。

ラベル定数 (label constant). ステートメント (PROCEDURE、ENTRY、FORMAT、または PACKAGE を除く) のラベル接頭語として書き込まれる名前。実行時に、そのラベル接頭語が参照されれば、そのステートメントにプログラムの制御を渡すことができる。

ラベル変数 (label variable). LABEL 属性を指定して宣言された変数。その値は、プログラム内でのラベル定数である。

ラベル・データ (label data). ラベル定数または、ラベル変数の値。

リスト指示 (list-directed). ストリーム内のデータがブランクやコンマで区切られた定数になり、フォーマット設定が自動的に行われるタイプのストリーム指向伝送。

リモート・フォーマット項目 (remote format item). R という文字の後に FORMAT ステートメントのラベル (括弧に囲まれている) のあるもの。フォーマット指定ステートメントは、転送するデータのフォーマットを制御するために、編集指示データ伝送ステートメントにより使用する。

領域 (area). 基底付き変数を割り振ることのできる、ストレージ中の部分。

ループ (loop). 繰り返し実行される一連の命令。

レコード (record). レコード単位入力または出力の操作における、伝送の論理単位。1 つ以上の関連データ項目の集まり。これらの項目には通常、それぞれ異なったデータ属性があり、また通常は構造体か共用体の宣言で記述される。

レコード単位データ伝送 (record-oriented data transmission). 別々のレコードの形式でデータを伝送すること。ストリーム指向のデータ伝送 (*stream data transmission*) と対比。

レベル 1 変数 (level-one variable). 大構造体または共用体の名前。構造体または共用体の中に含まれていない、添え字なし変数。

レベル番号 (level number). DECLARE ステートメント中の名前の前に付く番号で、構造体名の階層内のその相対位置を指定するもの。

連結 (concatenation). 2 つのストリングを指定順に結合し、元の 2 つのストリングの合計長に等しい長さを持った 1 つのストリングを作成する操作。これは、演算子 `||` で指定する。

連結参照 (connected reference). 連結ストレージに対する参照。プログラムを実行するには、ストレージが連結されていることが明らかでなければならない。

連結集合 (connected aggregate). エレメントが、間にデータ項目の入らない連続したストレージを占有する配列または構造体。非連結集合 (*nonconnected aggregate*) と対比。

連結ストレージ (connected storage). 単一名を使って参照することができる諸項目の非中断かつ線形の連なりの主記憶域。

ロケータ (locator). 変数のアドレスまたはその記述子を保持する制御ブロック。

ロケータ値 (locator value). ストレージ・アドレスを識別する値か、またはストレージ・アドレスを識別するのに使用できる値。

ロケータ修飾 (locator qualification). 基底付き変数への参照において、その参照が参照している基底付き変数の世代を指定するために、基底付き変数の左側に矢印で接続されているロケータ変数または関数参照。これは、暗黙参照であることもある。

ロケータ変数 (locator variable). 変数またはバッファの主記憶域内の位置を識別する値を持った変数。これは、POINTER 属性または OFFSET 属性を持つ。

ロケータ/記述子. その後に記述子の付いたロケータ。ロケータは、記述子のアドレスではなく、変数のアドレスを保持する。

ロック・レコード (locked record). EXCLUSIVE DIRECT UPDATE ファイル内のレコードであって、1 つのタスクにだけしか使用することはできず、そのレコードを使用しているタスクによって解放されるまで、他のタスクからアクセスできないレコード。

論理演算子 (logical operators). ビット・ストリング演算子の NOT や排他 OR (`^`)、AND (`&`)、および OR (`|`)。

論理レベル (構造体または共用体メンバーの) (logical level (of a structure or union member)). 全レベル番号が直接順序になっているとき (あるレベル番号から次のレベル番号までの増分が 1 のとき) に、レベル番号で示される深さ。

[ワ行]

割り当て (assignment). 値を変数に与える処理。

割り込み (interrupt). 条件やアテンションの発生の結果として、プログラムの制御の流れを変更すること。

割り振られた変数 (allocated variable). 主記憶域が関連付けられ解放されない変数。

割り振り (allocation). 変数用の主記憶域の予約。割り振られた変数の世代。PL/I ファイルを、システム・データ・セット、装置、またはファイルに関連付けること。

[数字]

1 次エントリー・ポイント (primary entry point). PROCEDURE ステートメントのラベル・リスト内の任意の名前によって識別されるエントリー・ポイント。

10 進 (decimal). 0 から 9 までの数字を使った数体系。

10 進固定小数点値 (decimal fixed-point value). 小数点の想定位置を持つ 10 進数の連なりで構成される有理数。2 進固定小数点値 (*binary fixed-point value*) と対比。

10 進固定小数点定数 (decimal fixed-point constant). 1 つ以上の 10 進数 (および任意で小数点を付けたもの) から成る定数。

10 進ピクチャー文字 (decimal digit picture character). ピクチャー指定文字 9 のこと。

10 進ピクチャー・データ (decimal picture data). 「数値ピクチャー・データ (*numeric picture data*)」を参照。

10 進浮動小数点値 (decimal floating-point value). 10 進小数部と考えることができる仮数形式の実数、および 10 を底とする整数のべき乗と考えることができる指数の近似値。2 進浮動小数点値 (*binary floating-point value*) と対比。

10 進浮動小数点定数 (decimal floating-point constant). 10 進固定小数点定数から成る仮数と、3 桁を超えないオプションの符号付き整数が後に付いた文字 E から成る指数とで構成される値。

16 進 (hex). 「16 進数字 (*hexadecimal digit*)」を参照。

16 進数 (hexadecimal). 16 の基数を持った数体系。有効数は 0 から 9 の数字と、A は 10 を、F は 15 を表す A から F までの文字。

16 進数字 (hexadecimal digit). 0 から 9 までと A から F までの数字のいずれか。A から F までは、それぞれ 10 進数値の 10 から 15 までを表す。

2 次エントリー・ポイント (secondary entry point). 入り口ステートメントのラベル・リスト内の任意の名前によって識別されるエントリー・ポイント。

2 進固定小数点値 (binary fixed-point value). 2 進数字で構成され、オプションの 2 進小数点とオプションの符号を持った整数。10 進固定小数点値 (*decimal fixed-point value*) と対比。

2 進数 (binary). 0 と 1 が唯一の数表示である数体系。

2 進数字 (binary digit). 「ビット (*bit*)」を参照。

2 進浮動小数点値 (binary floating-point value). 2 進小数部と見なせる仮数と、2 の基数に対する整数指数と

見なせる指数の形式の実数の近似値。10 進浮動小数点値 (*decimal floating-point value*) と対比。

A

ASCII. 情報交換用米国標準コード (American National Standard Code for Information Interchange)。

D

DBCS. 文字セットにおいて、それぞれの文字は 2 つの連続するバイトで表される。

DO グループ (do-group). DO ステートメントで区切れ、それに対応する END ステートメントで終了する、制御目的に使用される一連のステートメント。ブロック (*block*) と対比。

DO ループ (do-loop). 「反復 DO グループ (*iterative do-group*)」を参照。

E

EBCDIC. 拡張 2 進化 10 進コード (Extended Binary-Coded Decimal Interchange Code)。8 ビットのコード化文字からなるコード化文字セット。

end-of-step メッセージ (end-of-step message). ジョブ制御ステートメントとジョブ・スケジューラー・メッセージのリストに続いており、各ステップの成功または失敗を示す戻りコードを含むメッセージ。

I

ID (identifier). コメントや定数内に入ることがなく、前後に区切り文字を伴う文字のストリング。ID の先頭文字は、26 個の英字、または特別言語文字 (ある場合) でなければならない。その他の文字がある場合には、拡張英字、数字、区切り文字を追加して入れることができる。

IEEE. 米国電気電子学会 (Institute of Electrical and Electronics Engineers)。

O

ON ステートメント処置 (ON-statement action). ある条件が生じたときに処置を取れるよう、条件に対して明示的に設定された処置法。プログラムの制御の流れ内で ON ステートメントが見つかったと、とられる処置で、その条件に対する処置が設定される。この処置は、ON ユニットが設定されたままであるか、RESIGNAL ステー

トメントで再設定されて条件が生じたときにとられる。
暗黙の処置 (*implicit action*) と対比。

ON ユニット (ON-unit). 該当する条件が起きたときに、とられるよう指定された処置。

option. ステートメントの実行や解釈に影響を及ぼすのに使われるステートメント中の指定。

P

PL/I プロンプター (PL/I prompter). PL/I コマンドのコマンド・プロセッサ・プログラムで、オペランドを調べ、コンパイラーに必要なデータ・セットを割り振る。

PL/I 文字セット (PL/I character set). PL/I のプログラム・エレメントを表現するために定義されている文字セット。

R

REFER オブジェクト (REFER object). REFER オプション中の変数。メンバーの現行境界、長さ、またはサイズを保持しているか、あるいは保持する予定のもの。REFER オブジェクトは、同一構造体または共用体のメンバーでなければならない。これは、ロケータ修飾したり添え字を付けてはならず、また REFER オプションを持ったメンバーの前になければならない。

REFER 式 (REFER expression). REFER というキーワードの前に付いた式。この式は、REFER オプションを含む基底付き変数が、ALLOCATE ステートメントまたは LOCATE ステートメントのいずれかによって割り振られるときの境界、長さ、またはサイズとして使用される。

RETURNS 記述子 (RETURNS descriptor). RETURNS 属性内と、PROCEDURE および ENTRY ステートメントの RETURNS オプション内で使用する記述子。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

アクセス
 相対レコード・データ・セット 304
 ESDS 285
 REGIONAL(1) データ・セット 266
アクセス方式サービス
 領域データ・セット 267
 REGIONAL(1) データ・セット
 順次アクセス 264
 直接アクセス 264
アセンブラー・ルーチン
 FETCH 182
アテンション処理
 アテンション割り込み、効果 42
 主要な説明 453
 デバッグ・ツール 454
 ATTENTION ON ユニット 454
アプリケーション・パフォーマンスの向上
 309
一時作業ファイル
 SYSUT1 162
インクルード・プリプロセッサ
 構文 106
印刷
 PRINT ファイル
 行長 236
 ストリーム入出力 235
 フォーマット 243
印刷制御文字 48, 235
エラー
 エラー・コンパイルの重大度 17
エラー装置
 リダイレクト 219
エン트리・ポイント
 ソート・プログラム 334
オブジェクト
 モジュール
 作成と保管 54
 レコード・サイズ 162
オプション
 コメントの指定 89
 コンパイル用 95
 コンパイル用に指定する 163
 ストリングの指定 89

オプション (続き)
 領域データ・セットの作成用の 258
 PLIDUMP 内の保存されたオプション・ストリング 448
オプション、z/OS UNIX での使用
 DD 情報 192
 TITLE 192
DD_DDNAME 環境変数
 APPEND 194
 DELAY 196
 DELIMIT 196
 LRECL 196
 LRMSKIP 196
 PROMPT 197
 PUTPAGE 197
 RECCOUNT 197
 RECSIZE 197
 SAMELINE 198
 SKIPO 198
 TYPE 199
PL/I ENVIRONMENT 属性
 BUFSIZE 195
オフセット
 タブ・カウン트의 238
 テーブルの 97
オペレーティング・システム
 z/OS UNIX でのデータ定義 (DD) 情報 191

[カ行]

改行 (LF)
 定義 199
外部参照
 連結名 188
カウンター・レコード、SYSADATA 477
拡張 2 進化 10 進コード (EBCDIC) 201
カスタマイズ
 ユーザー出口
 グローバル制御ブロックの構造 462
 独自のコンパイラー出口の作成 461
 SYSUEXIT の変更 461
カタログ式プロシージャ
 コンパイル、バインド、および実行 144
 コンパイル、プリリンク、およびリンク・エディット 146

カタログ式プロシージャ (続き)
 コンパイル、プリリンク、リンク・エディット、および実行 148
 コンパイル、プリリンク、ロード、および実行 150
 コンパイルおよびバインド 142
 コンパイルのみ 140
 コンパイル用入力データ 144, 148
 説明 139
 複数呼び出し 153
 変更
 DD ステートメント 155
 EXEC ステートメント 154
 呼び出し 152
 リスト 152
 OS/390 の下での
 変更する 154
 呼び出す 152
 IBM 提供の 139
可変長レコード
 ソート・プログラム 350
 フォーマット 202
紙送り制御文字 48, 235
環境変数
 z/OS UNIX での設定 218
漢字ストリング定数のコンパイル 37
関数
 ILC で C 関数を使用する 355
キー
 代替索引
 固有 293
 非固有 293
 REGIONAL(1) データ・セット 260
 ダミー・レコード 261
 VSAM
 索引付きデータ・セット 275
 相対バイト・アドレス 275
 相対レコード 275
 キー索引付き VSAM データ・セット 275
 キー順データ・セット
 ステートメントとオプション 286
 ロード 288
 DIRECT ファイルを使ったアクセス 290
 SEQUENTIAL ファイルを使ったアクセス 289
記述子 467
記述子域、SQL 120
記述子リスト、引数の引き渡し 467
記述子ロケーター、引数の引き渡し 468

行	構文図の読み方 xv	コンパイラー・オプション (compiler options) (続き)
長さ 236	構文レコード、SYSADATA 498	デフォルト 5
メッセージ内の数 37	固定長レコード 201	AGGREGATE 8
記録済みキー	コメント	ARCH 9, 310
領域データ・セット 260	オプション内 89	ATTRIBUTES 10
句読点	混合文字ストリング定数のコンパイル 37	BACKREG 10
自動プロンプト	コンパイラー	BLANK 12
指定変更 170	一時作業ファイル (SYSUT1) 162	CEESTART 12
使用 170	エラー条件の重大度 17	CHECK 13
OS/390	オプションの説明 5	CMPAT 14
継続文字 171	概要 157	CODEPAGE 15
端末での ENDFILE の入力 172	漢字ストリング定数 37	COMPACT 16
長い入力行 171	混合ストリング定数 37	COMPILE 17
GET DATA ステートメント 171	削減、ストレージ要件の 55	CSECT 18
GET EDIT での自動埋め込み 171	呼び出し 157	CURRENCY 19
GET LIST ステートメント 171	リスト	DBCS 19
SKIP オプション 172	印刷オプション 162	DD 20
PRINT ファイルからの出力 169	組み込みソース・プログラム 42	DDSQL 20
組み込み	コンパイラーへの入力 95	DEFAULT 22, 312
CICS ステートメント 136	集合長さテーブル 97	DISPLAY 31
SQL ステートメント 121	使用 94	DLLINIT 32
グラフィック・データ 227	使用されるスタック・ストレージ 75	EXIT 32
グローバル制御ブロック	ステートメント・オフセット・アドレス 97	EXTRN 32
終了プロシージャの作成 465	ソース・プログラム 73	FLAG 33
初期化プロシージャの作成 463	相互参照テーブル 97	FLOAT 33
データ入力フィールド 462	属性テーブル 96	GOFF 36
メッセージ・フィルター操作プロシージャの作成 463	ファイル参照テーブル 101	GONUMBER 37, 310
グローバル制御ブロックの構造	プリプロセッサへの入力 95	GRAPHIC 37
終了プロシージャの作成 465	ブロック・レベル 95	HGPR 38
初期化プロシージャの作成 463	見出し情報 94	INCAFTER 38
メッセージ・フィルター操作プロシージャの作成 463	メッセージ 102	INCDIR 39
言語環境プログラム ライブラリー xiv	戻りコード 102	INSOURCE 42
言語間通信	DO レベル 95	INTERRUPT 42
リンケージの考慮事項 362	SOURCE オプション・プログラム 95	LANGLVL 43
C との 353	SYSPRINT 162	LIMITS 44
構造化データ・タイプ 354	DBCS ID 37	LINECOUNT 45
出力の共用 364	JCL ステートメントの使用 160	LINEDIR 45
ストリング・パラメーター・タイプの一一致 359	OS/390 バッチの下での 160	LIST 45
データ・タイプ 353	PROCESS ステートメント 90	LISTVIEW 46
パラメーターの一一致 357	% ステートメント 91	MACRO 47
ファイル・タイプ 355	コンパイラー・オプション	MAP 47
C 関数を使用する 355	BIFPREC 11	MARGINI 47
ENTRY を戻す関数 361	BLKOFF 12	MARGINS 48
enum データ・タイプ 354	COPYRIGHT 18	MAXMEM 49
コーディング	CSECTCUT 19	MAXMSG 50
パフォーマンスの向上 316	FLOATINMATH 36	MAXNEST 50
CICS ステートメント 136	NOMARGINS 48	MAXSTMT 51
SQL ステートメント 119	RESEXP 65	MAXTEMP 51
交換コード 201	STATIC 74	MDECK 51
更新	コンパイラー・オプション (compiler options)	NAME 52
相対レコード・データ・セット 304	省略語 5	NAMES 52
ESDS 285	説明 5	NATLANG 52
REGIONAL(1) データ・セット 266		NEST 53
		NOT 53
		NUMBER 53

コンパイラー・オプション (compiler

options) (続き)

OBJECT 54

OFFSET 54

OPTIMIZE 55, 309

OPTIONS 56

OR 56

PP 57

PPCICS 58

PPINCLUDE 58

PPMACRO 59

PPSQL 60

PPTRACE 60

PRECTYPE 60

PREFIX 61, 311

PROCEED 61

QUOTE 63

REDUCE 63, 310

RENT 64

RESPECT 65

RULES 66, 310

SEMANTIC 72

SERVICE 73

SOURCE 73

SPILL 73

STD SYS 74

STMT 74

STORAGE 75

STRING OF GRAPHIC 75

SYNTAX 75

SYSPARM 76

SYSTEM 77

TERMINAL 78

TEST 78

TUNE 82

USAGE 82

WIDECHAR 83

WINDOW 84

WRITABLE 84

XINFO 85

XML 88

XREF 88

コンパイラー・ユーザー出口の初期化プロ
シージャー 463

コンパイル

ユーザー出口

カスタマイズ 461

活動化 460

プロシージャー 459

IBMUEXIT 460

z/OS UNIX の下での 157

コンパイル、プリリンク、およびリンク・
エディット用の入力データ 146

コンパイルおよびバインド用の入力データ
142

コンパイル時オプション

z/OS UNIX の下での 158

コンパイル時オプションの指定

フラグの使用 159

[サ行]

再始動

要求 457

要求する

システム障害後の自動 457

据え置き再始動 458

取り消す 457

プログラム内の自動 458

変更する 458

RESTART パラメーター 458

最適コーディング

コーディング・スタイル 316

コンパイラー・オプション (compiler
options) 309

作業データ・セット、ソート用の 338

索引付き ESDS (入力順データ・セット)

ロード 288

DIRECT ファイル 290

SEQUENTIAL ファイル 289

索引データ・セット

索引順次データ・セット 203

削減、ストレージ要件の 55

サマリー・レコード、SYSADATA 477

サンプル・プログラム、実行 372, 376,
381, 385

システム

障害 457

障害後の再始動 457

SYSTEM コンパイラー・オプション

パラメーター・リストのタイプ 77

SYSTEM(CICS) 77

SYSTEM(IMS) 77

SYSTEM(MVS) 77

SYSTEM(OS) 77

SYSTEM(TSO) 77

自動

埋め込み 171

プロンプト

指定変更 170

使用 170

RESTART

システム障害後 457

チェックポイント/再始動機能 455

プログラム内の 458

シフト・コード・コンパイル 37

集合

長さテーブル 97

終了

コンパイル 17

終了プロシージャー

構文

グローバル 462

特有の 465

コンパイラー・ユーザー出口 465

プロシージャー特有の制御ブロックの
例 465

主記憶域、ソート用の 334

出力

ストリーム・ファイル用のデータ・セ
ットの定義 228

ソート用データ 339

ソート用の骨組みコード 343

ソート・データ・セット 339

ソート・プログラム用のルーチン 339

プリプロセッサ出力の制限 51

リダイレクト 219

PLISRTA 用のデータ 344

SEQUENTIAL 249

SYSLIN 162

SYSPUNCH 162

順次アクセス

REGIONAL(1) データ・セット 264

順次データ・セット 203

条件付きコンパイル 17

条件付きサブパラメーター 205

情報交換コード 201

初期ボリューム・ラベル 204

序数エレメント・レコード、

SYSADATA 481

序数タイプ・レコード、SYSADATA 480

処理ルーチン

ソート用データ

可変長レコード 350

出力 (ソート出口 E35) 343

成功かどうかの判別 337

入力 (ソート出口 E15) 340

PLISRTB 346

PLISRTC 347

PLISRTD 348

シンボル・テーブル 78

シンボル・レコード、SYSADATA 481

据え置き再始動 458

ステートメント 91

オフセット・アドレス 97

ネスト・レベル 95

ステップの異常終了 205

ストリーム入出力

データ伝送のレコード・フォーマット
211

データ・セット

アクセス 233

作成 230

レコード・フォーマット 234

ファイル

定義 228

ストリーム入出力 (続き)	ソース・ステートメント・ライブラリー	ダミー・レコード
ファイル (続き)	163	REGIONAL(1) データ・セット 261
PRINT ファイル 235	ソース・レコード、SYSADATA 497	VSAM 276
SYSIN および SYSPRINT ファイル 240	ソート	ダンプ
連続データ・セット 227	最大レコード長 334	先頭の識別 440
DD ステートメント 232, 235	準備 328	定義、データ・セット
ENVIRONMENT オプション 228	ストレージ	論理レコード長 440
ストリーム・ファイルおよびレコード・ファイル 242, 244	主記憶域 334	DD ステートメント 440
ストリング	補助 334	PLIDUMP 組み込みサブルーチン 439
漢字ストリング定数のコンパイル 37	説明 327	PLIDUMP の呼び出し 439
ストリング記述子	ソート・タイプの選択 328	SNAP 440
ストリング記述子 468, 471	ソート・フィールド 331	z/OS 言語環境プログラム ダンプの生成 439
ストリングの割り当て 317	ソート・プログラムの呼び出し 334	ダンプ・トレースバック・テーブル内のプログラム単位名 441
ストレージ	データ 327	端末
印刷ファイルのブロック化 236	データの入出力 339	出力 243
ソート・プログラム 334	入出力ルーチンの作成 339	大文字と小文字 244
主記憶域 334	評価結果 337	ストリーム・ファイルおよびレコード・ファイル 244
補助記憶域 334	CHKPT オプション 332	PRINT ファイルのフォーマット 243
標準データ・セット 160	CKPT オプション 332	PUT EDIT コマンドからの出力 244
要件の削減 55	DYNALLOC オプション 332	入力 241
ライブラリー・データ・セット 222	E15 入力処理ルーチン 340	大文字と小文字 243
リスト内のレポート 75	EQUALS オプション 332	ストリーム・ファイルおよびレコード・ファイル 242
スラッシュ (/) 193	FILSZ オプション 332	データのフォーマット 241
制御	PLISRT 328	ファイル終わり 243
域 272	PLISRTA(x) コマンド 344, 350	GET ステートメントの COPY オプション 243
インターバル 272	RECORD ステートメント 340	チェックポイント/再始動
CONTROL オプション	RETURN ステートメント 340	据え置き再始動 458
EXEC ステートメント 164	SKIPREC オプション 332	PLICANC ステートメント 458
制御ブロック	SORTCKPT 339	チェックポイント/再始動機能
機能専用 460	SORTCNTL 339	活動の変更 458
グローバル制御 461	SORTIN 338	再始動の要求 457
制御文字	SORTLIB 338	説明 455
印刷 48, 235	SORTOUT 339	チェックポイント・データ・セット 456
紙送り機構 48, 235	SORTWK 334, 338	チェックポイント・レコードの要求 455
ゼロ値 212	ソート用のフィールド 331	戻りコード 456
宣言	ソート用のフローチャート 340	CALL PLIREST ステートメント 458
ホスト変数、SQL プリプロセッサ 122	相互参照テーブル	PLICKPT 組み込みサブルーチン 455
OS/390 におけるファイルの 187	コンパイラー・リスト 97	RESTART パラメーター 458
ソース	XREF オプションの使用 96	チェックポイント・データ
キー	相対バイト・アドレス (RBA) 275	ソート用 339
REGIONAL(1) データ・セット内の 261	相対レコード 275	チェックポイント・データの定義、
プログラム	相対レコード・データ・セット	PLICKPT 組み込みサブオプション 456
外部テキストのシフト 47	ステートメントとオプション 299	直接データ・セット 203, 204
コンパイラー・リスト 95	ロード 301	データ
コンパイラー・リストに組み込まれた 42	DIRECT ファイルを使ったアクセス 304	ソート
データ・セット 161	SEQUENTIAL ファイルを使ったアクセス 303	説明 327
プリプロセッサ 47	属性テーブル 96	ソート・プログラム 339
リスト 73		
ID 10		
リスト		
位置 48		

[タ行]

大量順次挿入 292
 対話式プログラム
 アテンション割り込み 42
 タブ制御テーブル 238

データ (続き)

- PLISRT(x) コマンド 344
- タイプ
 - Java と PL/I の同等な 385
 - SQL と PL/I の同等な 125
- ファイル
 - z/OS UNIX での作成 194
 - z/OS UNIX での変換 191
- データ指示入出力 317
 - パフォーマンスのためのコーディング 316
- データ・セット
 - 一時的な 162
 - カタログ式プロシージャでの入力 140
 - 区分 221
 - クローズ 208
 - 索引
 - 順次 203
 - 順次 203
 - 使用 187
 - 条件付きサブパラメーターの特性 205
 - 情報交換コード 201
 - ストリーム・ファイル 227
 - ソース・ステートメント・ライブラリー 163
 - ソート 338
 - SORTWK 334
 - ソート・プログラム
 - 出力データ・セット 339
 - ソート作業データ・セット 338
 - チェックポイント・データ・セット 339
 - 入力データ・セット 338
 - 相対レコードの定義 302
 - タイプ
 - 比較 217
 - 編成 203
 - PL/I レコード入出力によって使用される 217
 - ダンプ用の定義
 - 論理レコード長 440
 - DD ステートメント 440
 - チェックポイント 456
 - 直接 203, 204
 - データ・セット制御ブロック (DSCB) 204
 - データ・セットとファイルとの関連付け 187
 - 特性の設定 200
 - ファイルとの関連付け解除 208
 - 複数のデータ・セットと 1 つのファイルの関連付け 190
 - ブロックおよびレコード 200
 - 編成
 - 条件付きサブパラメーター 205

データ・セット (続き)

- 編成 (続き)
 - タイプ 203
- データ定義 (DD) ステートメント 204
- ライブラリー
 - 更新 224
 - 使用 222
 - 情報の取り出し 226
 - タイプ 221
 - SPACE パラメーター 222
- ラベル 204, 221
- ラベルなし 204
- ラベルの変更 207
- 領域の 257
- レコード 200
- レコード・フォーマット
 - 可変長 202
 - 固定長 201
 - 不定長 203
- レコード・フォーマットのデフォルト値 210
- 連続ストリーム指向データ 227
- DD 名 160
- OS/390 でのデータ・セットの定義 187
- PL/I ファイルとの関連付け解除 190
- PL/I ファイルの関連付け
 - ファイルのオープン 207
 - ファイルのクローズ 208
- ENVIRONMENT 属性での特性の指定 208
- REGIONAL(1) 261
 - アクセスと更新 263
 - 作成 261
- SPACE パラメーター 160
- VSAM
 - キー 274
 - 索引付きデータ・セット 286
 - 大量順次挿入 292
 - ダミー・データ・セット 276
 - データ・セット・タイプ 275
 - 定義 282
 - パフォーマンス・オプション 281
 - ファイル属性 276
 - ファイルの定義 277
 - プログラムの実行 271
 - ブロック化 272
 - 編成 272
 - ENVIRONMENT オプションの指定 278
 - VSAM オプション 281
- データ・セット、OS/390 での
 - 連結 190
 - 1 つのデータ・セットと複数のファイルの関連付け 189

データ・セット、OS/390 での (続き)

- HFS 190
- データ・セット、z/OS UNIX での
 - 出力での拡張 194
 - 出力の再作成 194
 - デフォルトの識別 191
 - 特性の設定
 - DD_DDNAME 環境変数 194
 - パスの設定 194
 - 領域数 197
 - 領域の最大値 197
 - DD_DDNAME 環境変数 191
- PL/I ファイルとデータ・セットの関連付け
 - 環境変数の使用 191
 - 関連付けされていないファイルの使用 193
 - OPEN ステートメントの TITLE オプションの使用 192
 - PL/I によるデータ・セットの検索方法 194
- データ・セットの定義
 - 特性の指定 208
 - ファイルのオープン 207
 - ファイルのクローズ 208
 - 複数のデータ・セットと 1 つのファイルの関連付け 190
 - 複数のデータ・セットの連結 190
 - 複数のファイルと 1 つのデータ・セットの関連付け 189
 - ENVIRONMENT 属性 208
 - ESDS 284
- 定義、ファイル
 - 特性の指定 208
 - ファイルのオープン 207
 - ファイルのクローズ 208
 - 複数のデータ・セットの連結 190
 - 複数のファイルと 1 つのデータ・セットの関連付け 189
 - 領域データ・セット 259
 - キー 260
 - ENV オプション 260
 - ENVIRONMENT 属性 208
 - VSAM データ・セット 277
- 定義、OS/390 におけるファイル
 - 複数のデータ・セットと 1 つのファイルの関連付け 190
- 出口 (E15) 入力処理ルーチン 340
- 出口 (E35) 出力処理ルーチン 343
- トークン・レコード、SYSADATA 497
- 通し番号ボリューム・ラベル 204
- トレースバック・テーブル
 - 内のプログラム単位名 441
- トレーラー・ラベル 204

[ナ行]

長さ、レコードの
 z/OS UNIX での指定 197
名前付き定数
 対静的変数 320
 定義 320
入出力
 カタログ式プロシージャでの 140
 コンパイラー
 データ・セット 161
 コンパイル、プリリンク、およびリン
 ク・エディット用のデータ 146
 コンパイルおよびバインド用のデータ
 142
 ソート用の骨組みコード 341
 ソート・データ・セット 338
 OS/390、長い行の句読法 171
入力
 ストリーム・ファイル用のデータ・セ
 ットの定義 228
 ソート用データ 339
 ソート用の骨組みコード 343
 ソート・データ・セット 339
 ソート・プログラム用のルーチン 339
 リダイレクト 219
 PLISRTA 用のデータ 344
 SEQUENTIAL 249
入力順データ・セット
 更新 285
 定義 285
 VSAM 273
 ステートメントとオプション 283
 ESDS のロード 284
 SEQUENTIAL ファイル 284

[ハ行]

配列記述子
 配列記述子 470, 471
バッチ・コンパイル
 OS/390 157, 160
パフォーマンス
 VSAM オプション 281
パフォーマンスの向上
 コンパイラー・オプションの選択
 ARCH 310
 DEFAULT 312
 GONUMBER 310
 OPTIMIZE 309
 PREFIX 311
 REDUCE 310
 RULES 310
 パフォーマンスのためのコーディング
 ストリングの割り当て 317
 名前付き定数対静的変数 320

パフォーマンスの向上 (続き)
 パフォーマンスのためのコーディング
 (続き)
 入力専用パラメーター 317
 ライブラリー・ルーチンの呼び出し
 の回避 322
 ライブラリー・ルーチンの呼び出し
 のプリロード 323
 ループ制御変数 318
 DATA ディレクティブ入出力 316
 DEFINED 対 UNION 320
 GOTO ステートメント 317
 PACKAGE 対ネストされた
 PROCEDURE 318
 REDUCIBLE 関数 319
パラメーター引き渡し
 引数の引き渡し 467
 CMPAT(LE) 記述子 470
 CMPAT(V*) 記述子 468
引数
 ソート・プログラム 334
引数の引き渡し 467
 記述子リストによる 467
 記述子ロケーターによる 468
標識変数、SQL 132
標準データ・セット 160
標準ファイル (SYSPRINT と
 SYSIN) 219
ファイル
 クローズ 208
 データ・セットとファイルとの関連付
 け 187
 特性の設定 200
 OS/390 でのデータ・セットの定義
 187
 ファイル・レコード、SYSADATA 478
フォーマット表記規則 xv
複数
 呼び出し
 カタログ式プロシージャ 153
復帰 - 改行 (CR - LF) 199
フック
 位置サブオプション 78
不定長レコード 203
負の値
 ブロック・サイズ 213
 レコード長 212
フラグ、コンパイル時オプションの指定
 159
プリプロセス
 ソース・プログラム 47
 入力 95
 80 バイトまでに出力を制限 51
 MACRO を使った 47
 %INCLUDE ステートメント 92

プリプロセッサー
 組み込み 106
 マクロ・プリプロセッサー 107
 CICS オプション 136
 PL/I とともに提供される 105
 SQL オプション 112
 SQL プリプロセッサー 110
プロシージャ
 カタログ式、OS/390 の下での用法
 139
 コンパイル、バインド、および実行
 (IBMZCBG) 144
 コンパイル、プリリンク、リンク・エ
 ディット、および実行
 (IBMZCPLG) 148
 コンパイル、プリリンク、ロード、お
 よび実行 (IBMZCPG) 150
 コンパイルおよびバインド
 (IBMZCB) 142
 コンパイルおよびリンク・エディット
 (IBMZCPL) 146
 コンパイルのみ (IBMZC) 140
ブロック
 およびレコード 200
 サイズ
 オブジェクト・モジュール 162
 最大 213
 指定 200
 領域データ・セット 269
 レコード長 213
 連続データ・セット 249
 PRINT ファイル 237
ブロック間ギャップ (IBG) 200
プロンプト
 自動、指定変更 170
 自動、使用 170
ヘッダー・ラベル 204
補助記憶域、ソート用の 334
ホスト
 構造体 131
 変数、SQL ステートメント内での使用
 122
 ホスト変数の使用、SQL プリプロセッサ
 ー 122
 保存されたオプション・ストリング、
 PLIDUMP 内の 448
 ボリューム通し番号
 直接アクセス・ボリューム 204
 領域データ・セット 267

[マ行]

マクロ・プリプロセッサー
 マクロ定義 107
マルチタスキング
 PLIDUMP のオプション 439

未参照 ID 10
メッセージ
印刷フォーマット 240
コンパイラー・ユーザー出口での変更 461
コンパイラー・リスト 102
フィルター機能 463
ランタイム・メッセージ行番号 37
メッセージのフィルター操作 460
メッセージ・レコード、SYSADATA 479
文字
印刷制御 48, 235
紙送り制御 48, 235
文字ストリング属性テーブル 96
モジュール
オブジェクト・モジュールの作成と保管 54
戻りコード
コンパイラー・リスト 102
チェックポイント/再始動ルーチン 456
PLIRETC 338

[ヤ行]

ユーザー出口
カスタマイズ
グローバル制御ブロックの構造 462
独自のコンパイラー出口の作成 461
SYSUEXIT の変更 461
機能 460
コンパイラー 459
ソート 330
容量レコード
REGIONAL(1) 261
呼び出し
カタログ式プロシージャ 152
複数呼び出し 153
マルチタスキング・プログラムのリンク・エディット 154
予備ファイル 162

[ラ行]

ラージ・オブジェクト (LOB) サポート、SQL プリプロセッサ 128
ライブラリー
概要 204
構造体 226
コンパイルされたオブジェクト・モジュール 224
作成に必要な情報 222
作成例 224

ライブラリー (続き)
システム・プロシージャ
(SYS1.PROCLIB) 221
使用 221
ソース・ステートメント 163
ソース・ステートメント・ライブラリー 157
タイプ 221
データ・セット・ライブラリーの作成 222
ディレクトリー 222
メンバーの作成 225
用法 222
ライブラリー・ディレクトリーにある情報の取り出し 226
ライブラリー・メンバーの更新 226
ライブラリー・メンバーの作成と更新 223
ロード・モジュールの配置 225
SPACE パラメーター 222
ライブラリー・ルーチンのプリロード 323
ライブラリー・ルーチンの呼び出しの回避 322
ラベル
データ・セットの 204
ランタイム
メッセージ行番号 37
OS/390 の考慮事項
自動プロンプト 170
長い入力行の句読法 171
フォーマット設定規則 168
GET EDIT ステートメント 171
GET LIST ステートメントと GET DATA ステートメント 171
SKIP オプション 172
PLIXOPT の使用 168
リスト
カタログ式プロシージャ 152
コンパイラー
オプション 95
集合長さテーブル 97
ステートメントのネスト・レベル 95
ステートメント・オフセット・アドレス 97
ストレージ・オフセット・リスト 100
ファイル参照テーブル 101
プリプロセッサ入力 95
見出し情報 94
メッセージ 102
戻りコード 102
ATTRIBUTE と相互参照テーブル 96
DD 名リスト 5

リスト (続き)
コンパイラー (続き)
SOURCE オプション・プログラム 95
ステートメント・オフセット・アドレス 97
ストレージ・オフセット・リスト 100
ソース・プログラム 73
OS/390 バッチ・コンパイル 157, 162
SYSPRINT 162
リテラル・レコード、SYSADATA 478
領域
サイズ、EXEC ステートメント 160
REGION パラメーター 154
領域、z/OS UNIX での 197
領域データ・セット
オペレーティング・システム要件 267
ファイルの定義
キーの用法 260
領域データ・セット 259
ENVIRONMENT オプションの指定 260
DD ステートメント
アクセス 269
作成 268
REGIONAL(1) データ・セット
アクセスと更新 263
作成 261
使用 261
リンク・エディット
説明 167
ループ
制御変数 318
例
PLIDUMP の呼び出し 439
レコード
コンパイラー入力の最大サイズ 161
ソート・プログラム 333
チェックポイント 455
データ・セット 456
ブロック化解除 201
z/OS UNIX での長さ 197
レコード長
値 212
指定 200
領域データ・セット 257
レコード入力
データ伝送 244
データ・セット
アクセス 250
作成 249
タイプ 217
連続データ・セット 252
フォーマット 210
レコード・フォーマット 246
ENVIRONMENT オプション 246

レコードのブロック化解除 201

レコード・フォーマット

オプション 228

可変長レコード 202

固定長レコード 201

指定する 246

ストリーム入出力 234

タイプ 201

不定長レコード 203

連結

外部参照 188

データ・セット 190

連続データ・セット

ストリーム指向データ伝送 227

データ・セットの作成 230

データ・セットへのアクセス 233

ファイルの定義 228

ENVIRONMENT オプションの指定
228

PRINT ファイルの用法 235

SYSIN ファイルおよび SYSPRINT
ファイルの使用方法 240

端末からの入力 241

端末からの入力の制御

大文字と小文字 243

条件のフォーマット 241

ストリーム・ファイルおよびレコー
ド・ファイル 242

データのフォーマット 241

ファイルの終わり 243

GET ステートメントの COPY オ
プション 243

端末への出力 243

端末への出力の制御

大文字と小文字 244

条件 243

ストリーム・ファイルおよびレコー
ド・ファイル 244

PRINT ファイルのフォーマット
243

PUT EDIT コマンドの出力 244

定義と用法 227

レコード単位データ伝送

使用できるステートメントとオプシ
ョン 244

データ・セットのアクセスと更新
250

データ・セットの作成 249

ファイルの定義 246

ENVIRONMENT オプションの指定
246

レコード単位入出力 244

連絡域、SQL 120

ローダー・プログラムの使用 150

論理否定 53

論理和 56

[ワ行]

割り込み

主要な説明 453

対話式システムでのアテンション割り
込み 42

デバッグ・ツール 454

ATTENTION ON ユニット 454

A

ACCT EXEC ステートメント・パラメー
ター 154

AGGREGATE コンパイラー・オプション
8

ALIGNED コンパイラー・サブオプション
30

ALL オプション

フック位置サブオプション 78

ALLOCATE ステートメント 97

AMP パラメーター 271

ANS

コンパイラー・サブオプション 24

APPEND オプション、z/OS UNIX での
194

ARCH コンパイラー・オプション 9,
310

ASCII

コンパイラー・サブオプション
説明 24

ASSIGNABLE コンパイラー・サブオプシ
ョン 24

ATTENTION ON ユニット 454

ATTRIBUTES オプション 10

B

BACKREG コンパイラー・オプション
10

BIFPREC コンパイラー・オプション 11

BINIARG コンパイラー・サブオプション
27

BKWD オプション 209, 279

BLANK コンパイラー・オプション 12

BLKOFF コンパイラー・オプション 12

BLKSIZE

サブパラメーター 205

連続データ・セット 249

ENVIRONMENT 209

レコード入出力用の 212

DCB サブパラメーターとの比較
210

ENVIRONMENT オプションの
ストリーム入出力用の 228

BUFFERS オプション

ストリーム入出力用の 228

BUFND オプション 279

BUFNI オプション 279

BUFSIZE オプション、z/OS UNIX での
195

BUFSP オプション 280

BYADDR

説明 312

パフォーマンスへの影響 313

DEFAULT オプションでの用法 24

BYVALUE

説明 312

パフォーマンスへの影響 313

DEFAULT オプションでの用法 24

C

C ルーチン

FETCH 182

CEESTART コンパイラー・オプション
12

CHECK コンパイラー・オプション 13

CHKPT ソート・オプション 332

CICS

サポート 134

プリプロセッサ・オプション 136

CKPT ソート・オプション 332

CMPAT コンパイラー・オプション 14

COBOL

マップ構造 97

CODE サブパラメーター 205

CODEPAGE コンパイラー・オプション
15

COMPACT コンパイラー・オプション
16

COMPILE コンパイラー・オプション 17

COND EXEC ステートメント・パラメー
ター 154

CONNECTED コンパイラー・サブオプシ
ョン
説明 25

パフォーマンスへの影響 314

CONSECUTIVE

ENVIRONMENT オプションの 228,
246

COPY オプション 243

COPYRIGHT コンパイラー・オプション
18

CSECT コンパイラー・オプション 18

CSECTCUT コンパイラー・オプション
19

CTLASA オプションと CTL360 オプショ
ン

ENVIRONMENT オプション

連続データ・セット用の 247

SCALARVARYING 216

CURRENCY コンパイラー・オプション
19
CYLOFL サブパラメーター
DCB パラメーター 205

D

DBCS ID のコンパイル 37
DBCS コンパイラー・オプション 19
DCB サブパラメーター 208, 210
 主な説明 205
 カタログ式プロシージャー内の一時変
 更 155
 同等の ENVIRONMENT オプション
 210
 領域データ・セット 269
DD コンパイラー・オプション 20
DD 情報、z/OS UNIX での
 TITLE ステートメント 192
DD ステートメント 204
 カタログ式プロシージャー内の入力デ
 ータ・セット 140
 カタログ式プロシージャーの変更
 154, 155
 カタログ式プロシージャーへの追加
 155
 チェックポイント/再始動 456
 標準データ・セット 160
 出力 (SYSLIN, SYSPUNCH) 162
 入力 (SYSIN) 161
 ライブラリーの作成 222
 領域データ・セット 268
 OS/390 パッチ・コンパイル 160
 %INCLUDE 92
DD (データ定義) 情報、z/OS UNIX での
 191
DD 名
 標準データ・セット 160
 %INCLUDE 92
DDSQL コンパイラー・オプション 20
DD_DDNAME 環境変数
 APPEND 194
 DELAY 196
 DELIMIT 196
 LRECL 196
 LRMSKIP 196
 PROMPT 197
 PUTPAGE 197
 RECCOUNT 197
 RECSIZE 197
 SAMELINE 198
 SKIP0 198
 TYPE 199
 z/OS UNIX での代替 dd 名 193
 z/OS UNIX での特性の指定 194

DECLARE
 STATEMENT 定義 134
DEFAULT コンパイラー・オプション
 サブオプション
 ALIGNED 30
 ASCII または EBCDIC 24
 ASSIGNABLE または
 NONASSIGNABLE 24
 BINIARG または
 NOBINIARG 27
 BYADDR または BYVALUE 24
 CONNECTED または
 NONCONNECTED 25
 DESCLIST または
 DESCLOCATOR 28
 DESCRIPTOR または
 NODESCRIPTOR 25
 DUMMY 29
 E 31
 EVENDEC または
 NOEVENDEC 26
 HEXADEC 30
 IBM または ANS 24
 INITFILL または NOINITFILL 28
 INLINE または NOINLINE 26
 LINKAGE 26
 LOWERINC | UPPERINC 29
 NATIVE または NONNATIVE 25
 NATIVEADDR または
 NONNATIVEADDR 25
 NULLSTRADDR または
 NONNULLSTRADDR 27
 NULLSYS または NULL370 27
 ORDER または REORDER 26
 ORDINAL(MIN|MAX) 30
 OVERLAP | NOOVERLAP 30
 RECURSIVE または
 NONRECURSIVE 28
 RETCODE 29
 RETURNS 28
 SHORT 29
 説明および構文 22
DEFINED
 対 UNION 320
DELAY オプション、z/OS UNIX での
 説明 196
DELIMIT オプション、z/OS UNIX での
 説明 196
DESCLIST コンパイラー・サブオプシ
 ョン 28
DESCLOCATOR コンパイラー・サブオブ
 ション 28
DESCRIPTOR コンパイラー・オプション
 パフォーマンスへの影響 314

DESCRIPTOR コンパイラー・サブオプシ
 ョン
 説明 25
DFSORT 327
DIRECT ファイル
 RRDS
 データ・セットへのアクセス 304
VSAM での索引付き ESDS
 データ・セットの更新 292
 データ・セットへのアクセス 290
DISP パラメーター
 データ・セットを削除する 221
 連続データ・セット 251
 連続データ・セット用の 249
DISPLAY コンパイラー・オプション 31
DLL
 リンクの考慮事項およびサイド・デッ
 ク 167
 DYNAM=DLL リンカー・オプション
 175
 RENT コンパイラー・オプションおよ
 びフェッチ 174
DLLINIT コンパイラー・オプション 32
DSA
 各ブロックごとの PLIDUMP 組み込み
 サブルーチンに保存された 441
DSCB (データ・セット制御ブロック)
 204, 223
DSNAME パラメーター
 連続データ・セット用の 249, 251
DSORG サブパラメーター 205
DUMMY コンパイラー・サブオプション
 29
DYNALLOC ソート・オプション 332

E

E コンパイラー・サブオプション 31
E コンパイラー・メッセージ 102
E15 入力処理ルーチン 340
E35 出力処理ルーチン 343
EBCDIC
 コンパイラー・サブオプション 24
EBCDIC (拡張 2 進化 10 進コード) 201
ENDFILE
 OS/390 の下での 172
Enterprise PL/I ライブラリー
 言語環境プログラム ライブラリー
 xiv
Enterprise PL/I for z/OS ライブラリー
 xiv
ENVIRONMENT オプション
 同等の DCB サブパラメーター 210
 編成オプション 210
 領域データ・セット 260

ENVIRONMENT オプション (続き)
レコード・フォーマット・オプション 228
BUFFERS オプション
DCB サブパラメーターとの比較 210
CONSECUTIVE 228, 246
CTLASA および CTL360 247
DCB サブパラメーターとの比較 210
GRAPHIC オプション 230
KEYLENGTH オプション
DCB サブパラメーターとの比較 210
LEAVE および REREAD 248
RECSIZE オプション
使用法 229
レコード・フォーマット 229
DCB サブパラメーターとの比較 210
VSAM
BKWD オプション 279
BUFND オプション 279
BUFNI オプション 279
BUFSP オプション 280
GENKEY オプション 280
PASSWORD オプション 280
REUSE オプション 280
SKIP オプション 281
VSAM オプション 281
ENVIRONMENT 属性
リスト 208
z/OS UNIX での特性の指定
BUFSIZE 195
ENVIRONMENT の F オプション
ストリーム入出力用の 228
レコード入出力用の 211
ENVIRONMENT の FB オプション
ストリーム入出力用の 228
レコード入出力用の 211
ENVIRONMENT の FBS オプション
ストリーム入出力用の 228
レコード入出力用の 211
ENVIRONMENT の FS オプション
ストリーム入出力用の 228
レコード入出力用の 211
ENVIRONMENT の REGIONAL オプション 260
ENVIRONMENT の U オプション
ストリーム入出力用の 228
レコード入出力用の 211
ENVIRONMENT の V オプション
ストリーム入出力用の 228
レコード入出力用の 211
ENVIRONMENT の VB オプション
ストリーム入出力用の 228

ENVIRONMENT の VB オプション (続き)
レコード入出力用の 211
ENVIRONMENT の VBS オプション
ストリーム入出力用の 228
ENVIRONMENT の VS オプション
ストリーム入出力用の 228
EQUALS ソート・オプション 332
ESDS (入力順データ・セット)
更新 285
固有キー代替索引パス 293
定義 285
非固有キー代替索引パス 294
VSAM 273
ステートメントとオプション 283
ロード 284
EVENDEC コンパイラー・サブオプション 26
EXEC SQL ステートメント 110
EXEC ステートメント
オプションの指定 164
オプション・リストの最大長 164
カタログ式プロシージャの変更 154
コンパイラー 160
最小領域サイズ 160
紹介 160
OS/390 パッチ・コンパイル 157, 160
PARM パラメーター 163
EXIT コンパイラー・オプション 32
export コマンド 194
EXTERNAL 属性 96
EXTRN コンパイラー・オプション 32

F

F フォーマット・レコード 201
FB フォーマット・レコード 201
FETCH
アセンブラー・ルーチン 182
Enterprise PL/I ルーチン 174
OS/390 C ルーチン 182
FILE 属性 96
filespec 194
FILLERS、タブ制御テーブルの 238
FILSZ ソート・オプション 332
FIXED
z/OS UNIX での TYPE オプション 199
FLAG コンパイラー・オプション 33
FLOAT オプション 33
FLOATINMATH コンパイラー・オプション 36
FUNC サブパラメーター
使用法 205

G

GENKEY オプション
キーの分類 214
使用法 209
VSAM 278
GET DATA ステートメント 171
GET EDIT ステートメント 171
GET LIST ステートメント 171
GOFF コンパイラー・オプション 36
GONUMBER コンパイラー・オプション 310
定義 37
GOTO ステートメント 317
GRAPHIC オプション
コンパイラー 37
ストリーム入出力 228
ENVIRONMENT の 209, 230

H

HEXADEC コンパイラー・サブオプション 30
HGPR コンパイラー・オプション 38

I

I コンパイラー・メッセージ 102
IBM コンパイラー・サブオプション 24
IBMUEXIT コンパイラー・出口 460
IBMZC カタログ式プロシージャ 140
IBMZCB カタログ式プロシージャ 142
IBMZCBG カタログ式プロシージャ 144
IBMZCPG カタログ式プロシージャ 150
IBMZCPL カタログ式プロシージャ 146
IBMZCPLG カタログ式プロシージャ 148
ID
参照されない 10
ソース・プログラム 10
IEC225I 206, 268
ILC
リンケージの考慮事項 362
C との 353
構造化データ・タイプ 354
出力の共用 364
ストリング・パラメーター・タイプ
の一致 359
データ・タイプ 353
パラメーターの一致 357
ファイル・タイプ 355
ENTRY を戻す関数 361
enum データ・タイプ 354

ILC でのリンケージ考慮事項 362
INCAFTER コンパイラー・オプション
38
INCDIR コンパイラー・オプション 39
INCLUDE ステートメント
コンパイラー 92
INDEXAREA オプション 209
INITFILL コンパイラー・サブオプション
28
INLINE コンパイラー・サブオプション
26
INSOURCE オプション 42
INTERNAL 属性 96
INTERRUPT コンパイラー・オプション
42

J

JAVA 367
Java 368, 369, 371, 372, 373, 374, 375,
376, 379, 380, 381, 382, 385
Java コード、コンパイル 369, 373, 379,
382
Java コード、作成 368, 372, 376, 381
JCL (ジョブ制御言語)
高効率化 139
コンパイル中の使用 160
jni
JNI サンプル・プログラム 368, 372,
376, 381

K

KEYLEN サブパラメーター 205
KEYLENGTH オプション 209, 217
KEYLOC オプション
使用法 209
KEYTO オプション
VSAM の下での 284
KSDS (キー順データ・セット)
更新 290
固有キー代替索引パス 294
定義とロード 288
VSAM
ロード 288
DIRECT ファイル 290
SEQUENTIAL ファイル 289

L

LANGLVL コンパイラー・オプション
43
LEAVE および REREAD オプション
ENVIRONMENT オプション
連続データ・セット用の 248

LIMCT サブパラメーター 205, 269
LIMITS コンパイラー・オプション 44
LINE オプション 228, 236
LINECOUNT コンパイラー・オプション
45
LINEDIR コンパイラー・オプション 45
LINESIZE オプション
タブ制御テーブルの 238
OPEN ステートメント 229
LINKAGE コンパイラー・サブオプション
構文 26
パフォーマンスへの影響 315
LIST コンパイラー・オプション 45
LISTVIEW コンパイラー・オプション
46
LOWERINC コンパイラー・サブオプション
29
LRECL オプション、z/OS UNIX での
196
LRECL サブパラメーター 200, 205
LRMSKIP オプション、z/OS UNIX での
196

M

MACRO オプション 47
MAP コンパイラー・オプション 47
MARGINI コンパイラー・オプション 47
MARGINS コンパイラー・オプション
48
MAXMEM コンパイラー・オプション
49
MAXMSG コンパイラー・オプション 50
MAXSTMT コンパイラー・オプション
51
MAXTEMP コンパイラー・オプション
51
MDECK コンパイラー・オプション
説明 51
MODE サブパラメーター
使用法 205

N

NAME コンパイラー・オプション 52
NAMES コンパイラー・オプション 52
NATIVE コンパイラー・サブオプション
説明 25
NATIVEADDR コンパイラー・サブオプション
25
NATLANG コンパイラー・オプション
52
NEST オプション 53
NOBIN1ARG コンパイラー・サブオプション
27

NODESCRIPTOR コンパイラー・サブオプション
25
NOEQUALS ソート・オプション 332
NOEVENDEC コンパイラー・サブオプション
26
NOINITFILL コンパイラー・サブオプション
28
NOINLINE コンパイラー・サブオプション
26
NOINTERRUPT コンパイラー・オプション
42
NOMAP オプション 97
NOMARGINS コンパイラー・オプション
48
NONASSIGNABLE コンパイラー・サブオプション
24
NONCONNECTED コンパイラー・サブオプション
25
NONE、フック位置サブオプション 78
NONNATIVE コンパイラー・サブオプション
25
NONNATIVEADDR コンパイラー・サブオプション
25
NONRECURSIVE コンパイラー・サブオプション
28
NONNULLSTRADDR コンパイラー・サブオプション
27
NOOVERLAP コンパイラー・サブオプション
30
パフォーマンスへの影響 315
NOSYNTAX コンパイラー・オプション
75
NOT コンパイラー・オプション 53
note ステートメント 102
NTM サブパラメーター
使用法 205
NULL370 コンパイラー・サブオプション
27
NULLSTRADDR コンパイラー・サブオプション
27
NULLSYS コンパイラー・サブオプション
27
NUMBER コンパイラー・オプション 53

O

OBJECT コンパイラー・オプション
定義 54
OFFSET コンパイラー・オプション 54
OPEN ステートメント
PL/I ライブラリーのサブルーチン
207
TITLE オプション 206
OPTCD サブパラメーター 205
OPTIMIZE オプション 55

OPTIMIZE コンパイラー・オプション
309
OPTIONS オプション 56
OR コンパイラー・オプション 56
ORDER コンパイラー・サブオプション
説明 26
パフォーマンスへの影響 315
ORDINAL コンパイラー・サブオプション
30
ORGANIZATION オプション 217
使用法 209
OS/390
一般コンパイル 157
バッチ・コンパイル
一時作業ファイル (SYSUT1) 162
オプションの指定 163
ソース・ステートメント・ライブラ
リー (SYSLIB) 163
リスト (SYSPRINT) 162
DD ステートメント 160
EXEC ステートメント 160, 164
OVERLAP コンパイラー・サブオプショ
ン 30

P

PACKAGE 対ネストされた
PROCEDURE 318
PAGE オプション 228
PAGELENGTH、タブ制御テーブルの
238
PAGESIZE、タブ制御テーブルの 238
PARM パラメーター
オプションの指定 164
カタログ式プロシーチャー 154
PASSWORD オプション 280
PLICANC ステートメント、およびチェッ
クポイント/要求 458
PLICKPT 組み込みサブルーチン 455
PLIDUMP 組み込みサブルーチン
各ブロックごとの DSA に保存された
441
構文 439
出力
変数の検出 441
ダンプ・トレースバック・テーブル内
のプログラム単位名 441
変数
AUTOMATIC 変数の検出 441
CONTROLLED 変数の検出 444
PLIDUMP 出力内での検出 441
STATIC 変数の検出 442
保存されたオプション・ストリング
448
保存されたロード・モジュール のタイ
ム・スタンプ 447
PLIDUMP 組み込みサブルーチン (続き)
ユーザー ID 440
H オプション 440
z/OS 言語環境プログラム ダンプを生
成するための呼び出し 439
PLIREST ステートメント 458
PLIRETC 組み込みサブルーチン
ソートでの戻りコード 338
PLISAXA 389, 390
PLISAXB 389, 390
PLISAXC 419, 420
PLISRTA インターフェース 344
PLISRTB インターフェース 346
PLISRTC インターフェース 347
PLISRTD インターフェース 348
PLITABS 外部構造
制御セクション 239
宣言 169
PLIXOPT 変数 168
PL/I
コンパイラー
ユーザー出口のプロシーチャー
460
ファイル
z/OS UNIX でデータ・セットと関
連付ける 191
PL/I コード、コンパイル 371, 375, 380,
385
PL/I コード、作成 369, 374, 379, 382
PL/I コード、リンク 371, 375, 380, 385
PP コンパイラー・オプション 57
PPCICS コンパイラー・オプション 58
PPINCLUDE コンパイラー・オプション
58
PPMACRO コンパイラー・オプション
59
PPSQL コンパイラー・オプション 60
PPTRACE コンパイラー・オプション 60
PRECTYPE コンパイラー・オプション
60
PREFIX コンパイラー・オプション 61,
311
デフォルトのサブオプションの使用
312
PRINT ファイル
出力に句読点を付ける 169
レコード入出力 252
フォーマット設定規則 168
PROCEED コンパイラー・オプション
61
PROCESS ステートメント
オプション・デフォルトの指定変更
163
説明 90
PROMPT オプション、z/OS UNIX での
197

PRTSP サブパラメーター
使用法 205
PUT EDIT コマンド 244
PUTPAGE オプション、z/OS UNIX での
197

Q

QUOTE コンパイラー・オプション 63

R

REAL 属性 96
RECCOUNT オプション、z/OS UNIX で
の 197
RECFM サブパラメーター 205
使用法 205
データ・セットの編成における 205
RECORD ステートメント 333
RECSIZE オプション
ストリーム入出力用の 228, 229
定義 211
デフォルト値 229
連続データ・セット 229
z/OS UNIX での記述 197
RECURSIVE コンパイラー・サブオプシ
ョン 28
REDUCE コンパイラー・オプション 63
パフォーマンスへの影響 310
REDUCIBLE 関数 319
REGION サイズ、最低限必要な 140
RENT コンパイラー・オプション 64
REORDER コンパイラー・サブオプショ
ン
説明 26
パフォーマンスへの影響 315
RESEXP コンパイラー・オプション 65
RESPECT コンパイラー・オプション 65
RETCODE コンパイラー・サブオプショ
ン 29
RETURNS コンパイラー・サブオプショ
ン 28, 315
REUSE オプション 209, 280
RRDS (相対レコード・データ・セット)
更新 304
定義 302
ロード・ステートメントとオプション
299
VSAM
ロード 301
DIRECT ファイル 304
SEQUENTIAL ファイル 303
VSAM でのロード 301
RULES コンパイラー・オプション 66
パフォーマンスへの影響 310

S

S コンパイラー・メッセージ 102
SAMELINE オプション、z/OS UNIX での 198
SAX パーサー 389, 419
SCALARVARYING オプション 216
SEMANTIC コンパイラー・オプション 72
SEQUENTIAL ファイル
 RRDS、アクセス・データ・セット 303
 VSAM での ESDS
 更新 285
 定義とロード 284
 VSAM での索引付き ESDS
 データ・セットへのアクセス 289
SERVICE コンパイラー・オプション 73
SHORT コンパイラー・サブオプション 29
SKIP オプション 281
 ストリーム入出力における 228
 OS/390 の下での 172
SKIPO オプション、z/OS UNIX での 198
SKIPREC ソート・オプション 332
SOURCE コンパイラー・オプション 73
SPACE パラメーター
 標準データ・セット 160
 ライブラリー 222
SPILL コンパイラー・オプション 73
SQL プリプロセッサ
 オプション 112
 記述子域 120
 標識変数の使用 132
 ホスト構造体の使用 131
 ホスト変数の使用 122
 ラージ・オブジェクト・サポート 128
 連絡域 120
 EXEC SQL ステートメント 110
 IBMUEXIT の使用 133
SQLCA 120
SQLDA 120
STACK サブパラメーター
 使用法 205
STATIC コンパイラー・オプション 74
STD SYS コンパイラー・オプション 74
STMT コンパイラー・オプション 74
STMT サブオプション、テストの 78
STORAGE コンパイラー・オプション 75
STREAM 属性 227
STRINGOFGRAPHIC コンパイラー・オプション 75
SUB 制御文字 201
SYNTAX オプション 75

SYS1.PROCLIB (システム・プロシージャ
ー・ライブラリー) 221
SYSADATA 情報、カウンター・レコード 477
SYSADATA 情報、構文情報 497
SYSADATA 情報、構文レコード 498
SYSADATA 情報、サマリー・レコード 477
SYSADATA 情報、序数エレメント・レコード 481
SYSADATA 情報、序数タイプ・レコード 480
SYSADATA 情報、シンボル情報 479
SYSADATA 情報、シンボル・レコード 481
SYSADATA 情報、ソース・レコード 497
SYSADATA 情報、トークン・レコード 497
SYSADATA 情報、ファイル・レコード 478
SYSADATA 情報、メッセージ・レコード 479
SYSADATA 情報、リテラル・レコード 478
SYSADATA 情報の概要 475
SYSCHK デフォルト 456
SYSIN 161, 219
SYSIN および SYSPRINT ファイル 240
SYSLIB
 プリプロセス 163
 %INCLUDE 92
SYSLIN 162
SYSOUT 338
SYSPARM コンパイラー・オプション 76
SYSPRINT 219
 以前の PL/I との共有 173
 エンクレーブ間での共用 172
 および z/OS UNIX 219
 コンパイラー・リストの書き込み先 162
 必須 DD ステートメント 160
 C との共用 364
 DD オプションでの指定 20
 PUT ステートメントでの使用 240
 STD SYS オプションの作用 74
SYSPUNCH 162
SYSUT1 コンパイラー・データ・セット 162

T

TERMINAL コンパイラー・オプション 78

TEST コンパイラー・オプション
 定義 78
TIME パラメーター 154
TIMESTAMP
 PLIDUMP 内の保存されたロード・モ
 ジュールのタイム・スタンプ 447
TITLE オプション
 使用 206
 標準 SYSPRINT ファイルの関係付け 173
 z/OS UNIX での記述 192
TITLE オプション、OS/390 での
 文字ストリング値 187
TITLE オプション、z/OS UNIX での
 データ・セットに関連付けられてい
 ないファイルの使用 193
TUNE コンパイラー・オプション 82

U

U コンパイラー・メッセージ 102
U フォーマット 203
UNDEFINEDFILE 条件
 BLKSIZE エラー 213
 OPEN での行サイズの矛盾 237
 z/OS UNIX でのファイル・オープン時
 に発生する 200
UNDEFINEDFILE 条件、OS/390 での
 DD ステートメント・エラー 188
UNDEFINEDFILE 条件、z/OS UNIX での
 データ・セットに関連付けられてい
 ないファイルの使用 200
UNIT パラメーター
 連続データ・セット 251
UPPERINC コンパイラー・サブオプショ
ン 29
USAGE コンパイラー・オプション 82

V

VB フォーマット・レコード 202
VOLUME パラメーター
 連続データ・セット 249, 251
VSAM (仮想記憶アクセス方式)
 索引付きデータ・セット
 ロード・ステートメントとオプショ
 ン 286
 相対レコード・データ・セット 302
 大量順次挿入 292
 データ・セット
 キー 274
 キー順および索引付き入力順 286
 使用 271
 相対レコード 299
 代替索引 292

VSAM (仮想記憶アクセス方式) (続き)
データ・セット (続き)
 代替索引パス 281
 タイプの選択 275
 ダミー・データ・セット 276
 定義 282
 入力順 283
 パフォーマンス・オプション 281
 ファイル属性 276
 ファイルの定義 277
 プログラムの実行 271
 ブロック化 272
 編成 272
 ENVIRONMENT オプションの指定 278
 ファイルの定義 277
 パフォーマンス・オプション 281
 ENV オプション 278
 VSAM オプション 281
VSAM での KSDS
 VSAM での索引付き ESDS
 データ・セットの更新 292
 データ・セットへのアクセス 289, 290
VTOC 204

W

W コンパイラー・メッセージ 102
WIDECHAR コンパイラー・オプション 83
WINDOW コンパイラー・オプション 84
WRITABLE コンパイラー・オプション 84

X

XINFO コンパイラー・オプション 85
XML
 SAX パーサーでのサポート 389, 419
XML コンパイラー・オプション 88
XREF コンパイラー・オプション 88

Z

z/OS UNIX
 環境変数の設定 218
 コンパイル 157
 コンパイル時オプション
 指定 158
 コンパイル時オプションの指定
 コマンド行 158
 フラグの使用 159
 DD_DDNAME 環境変数 194
 export コマンド 194

z/OS UNIX での TYPE オプション 199
z/OS UNIX でのデータ定義 (DD) 情報 191
z/OS UNIXでの代替 dd 名、TITLE オプションの 193

〔特殊文字〕

*PROCESS、内のオプションの指定 90
/ (スラッシュ) 193
% ステートメント 91
%INCLUDE ステートメント 91, 163
 制御ステートメント 91
 ソース・ステートメント・ライブラリ
 ー 163
%NOPRINT 91
 制御ステートメント 91
%NOPRINT ステートメント 91
%OPTION 91
%OPTION ステートメント 91
%PAGE 91
 制御ステートメント 91
%PAGE ステートメント 91
%POP ステートメント 92
%PRINT 91
 制御ステートメント 91
%PRINT ステートメント 91
%PROCESS、内のオプションの指定 90
%PUSH ステートメント 92
%SKIP 91
 制御ステートメント 91
%SKIP ステートメント 91



プログラム番号: 5655-H31

Printed in Japan

Enterprise PL/I for z/OS ライブラリー

SC27-1456

Licensed Program Specifications

SC88-9123

プログラミング・ガイド

GC88-9124

コンパイラおよびランタイム 移行ガイド

SC88-9126

言語解説書

SC88-9127

メッセージおよびコード

SC88-9123-06



日本アイ・ビー・エム株式会社

〒106-8711 東京都港区六本木3-2-12