

Rational Developer for System z バージョン 7.6
PL/I for Windows



プログラミング・ガイド

バージョン 7.6

Rational Developer for System z バージョン 7.6
PL/I for Windows



プログラミング・ガイド

バージョン 7.6

— 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、489 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、Rational Developer for System z PL/I for Windows バージョン 7.6、および新しい版またはテクニカル・ニュースレターで明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。製品のレベルに合った正しい版をご使用ください。

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： SC18-9977-02

Rational Developer for System z, Version 7.6

PL/I for Windows

Programming Guide

Version 7.6

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第7版第1刷 2009.10

© Copyright International Business Machines Corporation 1998, 2009.

目次

図	xi
-------------	----

第 1 部 ワークステーション上での PL/I の導入 1

第 1 章 本書について	3
新機能	3

第 2 章 構文図の読み方	5
-------------------------	---

第 3 章 ご意見の送付方法	9
--------------------------	---

第 4 章 プラットフォーム間でのアプリケ ーションの移植 11

メインフレーム・アプリケーションのワークステ ーション上でのコンパイル	11
正しいコンパイル時オプションの選択	12
制限された言語	12
プログラムの移植を助けるマクロ機能の使用	15
メインフレーム・アプリケーションのワークステ ーション上での実行	16
リンクの相違	16
ランタイムの相違をもたらすデータ表記	16
移植性に影響する環境の相違	19
ランタイムの相違を引き起こす言語エレメント	20

第 2 部 プログラムのコンパイルおよ びリンク 23

第 5 章 プログラムのコンパイル 25

簡単な練習問題	25
HELLO プログラム	25
コンパイル時オプションの使用	26
製品同梱のサンプル・プログラムの使用	26
ソース・プログラムのコンパイルの準備	27
プログラム・ファイルの構造	27
プログラム・ファイルのフォーマット	29
コンパイル時環境変数の設定	30
IBM.OPTIONS	31
IBM.PPINCLUDE	31
IBM.PPMACRO	31
IBM.PPSQL	31
IBM.PPCICS	32
IBM.SOURCE	32
IBM.SYSLIB	32
IBM.PRINT	32
IBM.OBJECT	33
IBM.DECK	33
INCLUDE	33

TMP	33
コンパイラを呼び出す PLI コマンドの使用	33
コンパイル時オプションを指定する場所	34
IBM.OPTIONS と IBM.PPxxx 環境変数	34
PLI コマンド	34
%PROCESS ステートメント	35

第 6 章 コンパイル時オプションの説明 37

コンパイル時オプションの説明	37
コンパイル時オプションの使用規則	39
ADDEXT	40
AGGREGATE	40
ATTRIBUTES	41
BIFPREC	41
BLANK	42
CHECK	43
CMPAT	44
CODEPAGE	44
COMPILE	45
COPYRIGHT	45
CURRENCY	46
DBCS	46
DEFAULT	46
DLLINIT	54
EXIT	55
EXTRN	55
FLAG	55
FLOATINMATH	56
GONUMBER	56
GRAPHIC	57
IMPRECISE	57
INCAFTER	58
INCLUDE	58
INITAUTO	59
INITBASED	59
INITCTL	59
INITSTATIC	60
INSOURCE	60
LANGLVL	61
LIBS	61
LIMITS	62
LINECOUNT	63
LINEDIR	64
LIST	64
LISTVIEW	64
MACRO	66
MARGINI	66
MARGINS	66
MAXGEN	67
MAXMSG	68
MAXNEST	68

MAXSTMT	69
MAXTEMP	69
MDECK	69
MSG.	70
NAMES.	70
NATLANG.	71
NEST	71
NOT	72
NUMBER	72
OBJECT	73
OFFSET	73
ONSNAP	73
OPTIMIZE	74
OPTIONS	74
OR	75
PP	75
PPCICS	76
PPINCLUDE	77
PPMACRO.	77
PPSQL	78
PPTRACE	78
PRECTYPE	79
PREFIX.	79
PROBE	80
PROCEED	80
PROCESS	81
QUOTE.	81
REDUCE	82
RESEXP	82
RESPECT	83
RULES	83
SEMANTIC	90
SNAP	90
SOSI.	91
SOURCE	91
STATIC.	91
STMT	92
STORAGE	92
SYNTAX	92
SYS Parm	93
SYSTEM	93
TERMINAL	94
TEST	94
USAGE.	95
WIDECHAR	96
WINDOW	96
XINFO	96
XML.	99
XREF	99

第 7 章 PL/I プリプロセッサ 101

インクルード・プリプロセッサ	102
例:	102
インクルード・プリプロセッサ・オプション環 境変数	102
マクロ・プリプロセッサ	103

マクロ・プリプロセッサのオプション	103
マクロ機能オプション環境変数	105
SQL プリプロセッサ	106
プログラミングとコンパイルに関する考慮事項	106
SQL プリプロセッサのオプション	107
SQL プリプロセッサ・オプション環境変数	115
SQL プリプロセッサ BIND 環境変数	115
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	115
ラージ・オブジェクト (LOB) サポート	121
ユーザー定義関数のサンプル・プログラム	124
CICS サポート	132
プログラミングとコンパイルに関する考慮事項	132
CICS プリプロセッサのオプション	134
CICS プリプロセッサ・オプション環境変数	135
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	135
PL/I を使用した CICS トランザクションの作成	136
PL/I プログラムに使用される CICS 異常終了	137
CICS ランタイム・ユーザー出口	137

第 8 章 コンパイル出力 139

コンパイラ・リストの使用	139
コンパイラ出力ファイル	147

第 9 章 プログラムのリンク 149

リンカーの開始	149
静的リンク	149
コマンド行によるリンク	149
MAKE ファイルでリンク	151
入出力	152
検索規則	152
ディレクトリーの指定	153
ファイル名のデフォルト	153
オブジェクト・ファイルの指定	153
応答ファイルの使用	153
実行可能ファイルの出力タイプの指定	154
.EXE ファイルの作成	154
ダイナミック・リンク・ライブラリーの作成	155
実行可能ファイルのパッキング	156
マップ・ファイルの生成	156
リンカーの戻りコード	156

第 10 章 リンカー・オプションの設定 157

コマンド行でのオプションの設定	157
ILINK 環境変数でのオプションの設定	158
リンカーの使用	158
数値引数の指定	158
Windows リンカー・オプションの要約	160
Windows リンカー・オプション	161
/?	161
/ALIGNADDR	161
/ALIGNFILE	161
/BASE	161
/CODE	162
/DATA	162

/DBGPACK、/NODBGPACK	162
/DEBUG、/NODEBUG	163
/DEFAULTLIBRARYSEARCH	163
/DLL	164
/ENTRY	164
/EXECUTABLE	164
/EXTDICTIONARY、/NOEXTDICTIONARY	164
/FIXED、/NOFIXED	165
/FORCE	165
/HEAP	165
/HELP	165
/INCLUDE	166
/INFORMATION、/NOINFORMATION	166
/LINENUMBERS、/NOLINENUMBERS	166
/LOGO、/NOLOGO	166
/MAP、/NOMAP	167
/OUT	167
/PMTYPE	167
/SECTION	168
/SEGMENTS	168
/STACK	169
/STUB	169
/SUBSYSTEM	169
/VERBOSE	169
/VERSION	170

第 3 部 プログラムの実行およびデバッグ 171

第 11 章 ランタイム・オプションの使用 173

ランタイム環境変数の設定	173
PATH	173
DPATH	173
ランタイム・オプションの指定	173
ランタイム・オプションの指定場所	173
複数のランタイム・オプションまたはサブオプションの指定	175
ランタイム・オプション	175
NATLANG	175
ランタイム DLL の出荷	175

第 12 章 プログラムのテストとデバッグ 177

プログラムのテスト	177
一般的なデバッグのヒント	179
PL/I デバッグ技法	179
デバッグでのコンパイル時オプションの使用	179
デバッグでのフットプリントの使用	181
デバッグでのダンプの使用	182
デバッグでのエラーおよび条件処理の使用	186
エラー処理の概念	188
一般的なプログラミング・エラー	190
ソース・プログラムの論理エラー	190
PL/I の使用方法が無効	190

未初期化の入り口変数の呼び出し	190
ループおよびその他の予期しないエラー	190
予期しない入出力データ	191
予期しないプログラム終了	192
その他の予期しないプログラム結果	193
コンパイラまたはライブラリー・サブルーチンの障害	193
システム障害	193
ローパフォーマンス	194

第 4 部 入出力 195

第 13 章 データ・セットとファイルの使用 197

データ・セットのタイプ	198
ネイティブ・データ・セット	199
その他のデータ・セット	200
データ・セット特性の設定	201
レコード	201
レコード・フォーマット	202
データ・セットの編成	202
PL/I ENVIRONMENT 属性の使用による特性の指定	202
DD:ddname 環境変数の使用による特性の指定	209
PL/I ファイルとデータ・セットの関連付け	219
環境変数の使用	220
OPEN ステートメントの TITLE オプションの使用	220
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	221
PL/I によるデータ・セットの検索方法	221
PL/I ファイルのオープンとクローズ	221
ファイルのオープン	221
ファイルのクローズ	222
複数のデータ・セットと 1 つのファイルの関連付け	222
入出力ステートメント、属性、およびオプションの組み合わせ	222
DISPLAY ステートメントの入出力	224
PL/I 標準ファイル (SYSPRINT と SYSIN)	225
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	225

第 14 章 連続データ・セットの定義と使用 227

プリンター向けファイル	227
ストリーム指向データ伝送の使用	228
ストリーム入出力を用いたファイルの定義	229
ストリーム指向データ伝送用 ENVIRONMENT オプション	229
ストリーム入出力によるデータ・セットの作成	229
ストリーム入出力によるデータ・セットへのアクセス	232
PRINT ファイルの使用	234

SYSIN ファイルおよび SYSPRINT ファイルの 使用方法	239
コンソールからの入力の制御	239
会話型のファイルの使用	240
データのフォーマット	240
ストリーム・ファイルおよびレコード・ファイル	241
大文字と小文字	241
ファイルの終わり	242
コンソールへの出力の制御	242
PRINT ファイルのフォーマット	242
ストリーム・ファイルおよびレコード・ファイル	242
対話式プログラムの例	243
レコード単位入出力の使用	244
レコード入出力の使用によるファイルの定義	246
レコード単位データ伝送用 ENVIRONMENT オ プション	246
レコード入出力によるデータ・セットの作成	246
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	247

第 15 章 領域データ・セットの定義と

使用	253
領域データ・セット用のファイルの定義	255
ENVIRONMENT オプションの指定	256
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	256
領域データ・セットでのキーの使用	256
REGIONAL(1) データ・セットの使用	256
ダミー・レコード	257
REGIONAL(1) データ・セットの作成	257
例	257
REGIONAL(1) データ・セットへのアクセスと更 新	259
順次アクセス	259
直接アクセス	260
例	260

第 16 章 ワークステーション VSAM デ ータ・セットの定義と使用 265

ワークステーションとメインフレーム間のデータの 移動	266
ワークステーション VSAM 編成	266
ワークステーション VSAM データ・セットの作 成とアクセス	266
必要なワークステーション VSAM データ・セッ トの判別	267
ワークステーション VSAM データ・セット内の レコードへのアクセス	267
ワークステーション VSAM データ・セットのキー の使用	268
データ・セット・タイプの選択	269
ワークステーション VSAM データ・セットのファ イルの定義	269
PL/I ENVIRONMENT 属性のオプションの指定	270
既存のプログラムのワークステーション VSAM 向けの修正	270

ワークステーション VSAM 順次データ・セットの 使用	273
順次ファイルを用いたワークステーション VSAM 順次データ・セットへのアクセス	274
ワークステーション VSAM 順次データ・セット の定義とロード	275
ワークステーション VSAM キー順データ・セット	277
ワークステーション VSAM キー順データ・セッ トのロード	280
SEQUENTIAL ファイルを用いたワークステーシ ョン VSAM キー順データ・セットへのアクセス	282
DIRECT ファイルを用いたワークステーション VSAM キー順データ・セットへのアクセス	283
ワークステーション VSAM 直接データ・セット	286
ワークステーション VSAM 直接データ・セット のロード	289
SEQUENTIAL ファイルを使用したワークステー ション VSAM 直接データ・セットへのアクセス	291
DIRECT ファイルを使用したワークステーショ ン VSAM 直接データ・セットへのアクセス	292

第 5 部 データベースとの PL/I の 使用 295

第 17 章 Open Database

Connectivity	297
ODBC の紹介	297
背景	297
ODBC ドライバー・マネージャー	298
組み込み SQL または ODBC の選択	298
ODBC ドライバーの使用	298
オンライン・ヘルプ	299
環境固有の情報	299
データ・ソースへの接続	299
エラー・メッセージ	300
PL/I からの ODBC API	301
CALL インターフェースの規則	302
提供されるインクルード・ファイルの使用	302
ODBC C タイプのマッピング	304
ODBC ドライバー・マネージャー/ドライバの ライセンス情報の設定	304
提供されるインクルード・ファイルを使用したサン プル・プログラム	305

第 18 章 java Dclgen の使用 307

java Dclgen 用語の理解	307
PL/I java Dclgen のサポート	308
テーブル宣言とホスト構造の作成	309
データベースの選択	309
テーブルの選択と PL/I 宣言の生成	310
生成された PL/I 宣言の変更と保管	310
java Dclgen の終了	311
プログラムへのデータ宣言の組み込み	311

第 6 部 上級トピック 313

第 19 章 Program Maintenance

Utility NMAKE の使用 315

NMAKE を使用する理由	316
NMAKE の実行	316
コマンド行の使用	316
NMAKE コマンド・ファイルの使用	317
NMAKE オプション	318
エラー・ファイルの生成 (/X)	319
すべてのターゲットのビルド (/A)	319
メッセージの抑制 (/C)	319
変更日の表示 (/D)	319
環境変数のオーバーライド (/E)	319
記述ファイルの指定 (/F)	319
ヘルプの表示 (/HELP または /?)	320
終了コードの無視 (/I)	320
コマンドの表示 (/N)	320
サインオン・バナーの抑制 (/NOLOGO)	320
マクロ定義とターゲット定義の出力 (/P)	320
終了コードのリターン (/Q)	320
TOOLS.INI ファイルの無視 (/R)	321
コマンド表示の抑制 (/S)	321
ターゲット変更日の変更 (/T)	321
記述ファイル	321
記述ブロック	321
特殊機構	322
複数の記述ブロックでのターゲット	323
マクロの使用	323
マクロの例	324
特殊機構	324
記述ファイル内のマクロ	324
コマンド行でのマクロ	325
継承されたマクロ	325
定義されたマクロ	325
マクロ置換	326
特殊マクロ	326
特殊マクロの例	327
ファイル指定の各部分	328
特殊マクロを変更する文字	328
変更された特殊マクロの例	328
マクロ優先順位規則	329
推論規則	329
特殊機構	330
推論規則の例	331
推論規則パス指定	331
事前定義推論規則	331
ディレクティブ	332
ディレクティブの例	334
疑似ターゲット	334
事前定義疑似ターゲット	334
インライン・ファイル	336
インライン・ファイルの例	336
エスケープ文字	337
コマンドを変更する文字	337
エラー・チェックのオフへの切り替え (-)	338
ダッシュ・コマンド修飾子の例	338
コマンド表示の抑制 (@)	338

アットマーク (@) コマンド修飾子の例	338
従属ファイルに対するコマンド実行 (!)	339
感嘆符 (!) コマンド修飾子の例	339
EXTMAKE 構文	339
TOOLS.INI のマクロと推論規則	340
TOOLS.INI の例	340

第 20 章 パフォーマンスの向上 341

最適なパフォーマンスのためのコンパイル時オプションの選択	341
OPTIMIZE	342
IMPRECISE	342
GONUMBER	342
SNAP	342
RULES	343
PREFIX	343
DEFAULT	344
パフォーマンスを向上させるコンパイル時オプションの要約	348
パフォーマンス向上のためのコーディング	348
DATA 指示入出力	349
入力専用パラメーター	349
ストリングの割り当て	350
ループ制御変数	350
PACKAGE 対ネストされた PROCEDURE	351
REDUCIBLE 関数	352
DEFINED 対 UNION	352
名前付き定数対静的変数	352
ライブラリー・ルーチンの呼び出しの回避	354

第 21 章 ユーザー出口の用法 357

コンパイラー・ユーザー出口の使用	357
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	358
コンパイラー・ユーザー出口の活動化	358
IBM 提供のコンパイラー出口、IBMUEXIT	359
コンパイラー・ユーザー出口のカスタマイズ	359
CICS ランタイム・ユーザー出口の使用	364
プログラム起動前の動作	364
プログラム終了後の操作	365
CEEFXITA の変更	365
データ変換表の使用	365

第 22 章 ダイナミック・リンク・ライブラリーの構築 367

DLL ソース・ファイルの作成	367
DLL ソースのコンパイル	368
DLL のリンク準備	368
Windows におけるエクスポートする名前の指定	368
DLL のリンク	368
DLL の使用	369
DLL を構築するサンプル・プログラム	369
メインプログラムでの FETCH および RELEASE の使用	371
DLL からのデータのエクスポート	371

第 23 章 Windows における IBM Library Manager の使用 373

ILIB の実行	373
コマンド行の使用	374
ILIB 環境変数の使用	375
ILIB 応答ファイルの使用	375
ILIB パラメーターの指定例	376
ILIB 入力の制御	377
ILIB 出力の制御	377
ILIB 出力の制御	378
ILIB オブジェクト	379
ILIB オブジェクトの要約	379
Add/Replace	379
/EXTRACT	380
/REMOVE	381
ILIB オプション	381
ILIB オプションの要約	381
/?	382
/BACKUP	382
/DEF	383
/FREEFORMAT	383
/GENDEF	383
/GI	383
/HELP	384
/LIST	384
/NOEXT	384
/OUT	385
/QUIET	385
/WARN	385

第 24 章 呼び出し規則 387

リンケージについての考慮事項の理解	387
OPTLINK リンケージ	389
OPTLINK の機能	389
OPTLINK 使用のヒント	390
汎用レジスタの説明	390
パラメーター	390
パラメーターの受け渡し例	390
SYSTEM リンケージ	395
SYSTEM の機能	395
SYSTEM リンケージの使用例	396
STDCALL リンケージ (Windows のみ)	398
STDCALL の機能	398
STDCALL 規則の使用例	399
WinMain の使用 (Windows のみ)	400
CDECL リンケージ	400
CDECL の機能	401
CDECL 規則の使用例	401

第 25 章 混合言語アプリケーションでの PL/I の使用 405

データおよびリンケージの一致	405
受け渡されるデータの内容	406
データが受け渡される方法	408
データが受け渡される場所	409

使用する環境の保守	410
PL/I メインからの非 PL/I ルーチンの呼び出し	410
非 PL/I メインからの PL/I ルーチンの呼び出し	411
ON ANYCONDITION の使用	411

第 26 章 Java とのインターフェース 413

Java Native Interface (JNI) の概要	413
JNI サンプル・プログラム #1 - 「Hello World」	414
ステップ 1: Java プログラムの作成	414
ステップ 2: Java プログラムのコンパイル	415
ステップ 3: PL/I プログラムの作成	415
ステップ 4: PL/I プログラムのコンパイルとリンク	417
ステップ 5: サンプル・プログラムの実行	417
JNI サンプル・プログラム #2 - スtringの引き渡し	418
ステップ 1: Java プログラムの作成	418
ステップ 2: Java プログラムのコンパイル	419
ステップ 3: PL/I プログラムの作成	420
ステップ 4: PL/I プログラムのコンパイルとリンク	421
ステップ 5: サンプル・プログラムの実行	422
JNI サンプル・プログラム #3 - 整数の引き渡し	422
ステップ 1: Java プログラムの作成	422
ステップ 2: Java プログラムのコンパイル	424
ステップ 3: PL/I プログラムの作成	424
ステップ 4: PL/I プログラムのコンパイルとリンク	425
ステップ 5: サンプル・プログラムの実行	426
Java および PL/I の同等なデータ型の判別	426

第 27 章 ソート・ルーチンの使用 . . . 427

S/390 とワークステーションのソート・プログラム の比較	427
ソート・プログラムの使用準備	429
ソート・タイプの選択	429
ソート・フィールドの指定	432
ソートするレコードの指定	433
ソート・プログラムの呼び出し	434
PLISRT の例	434
ソートが成功したかどうかの判別	435
ソート・データの入出力	435
ソート・データの処理ルーチン	436
E15 - 入力処理ルーチン (ソート出口 E15)	436
E35 - 出力処理ルーチン (ソート出口 E35)	438
PLISRТА の呼び出し	441
PLISRTB の呼び出し	442
PLISRTC の呼び出し	444
PLISRTD の呼び出し、例 1	445
PLISRTD の呼び出し、例 2	446

第 28 章 SAX パーサーの使用 447

概要	447
PLISAXA 組み込みサブルーチン	448
PLISAXB 組み込みサブルーチン	448
SAX イベント構造体	448

start_of_document	449
version_information	449
encoding_declaration	449
standalone_declaration	449
document_type_declaration	449
end_of_document	449
start_of_element	450
attribute_name	450
attribute_characters	450
attribute_predefined_reference	450
attribute_character_reference	450
end_of_element	450
start_of_CDATA_section	450
end_of_CDATA_section	451
content_characters	451
content_predefined_reference	451
content_character_reference	451
processing_instruction	451
comment	452
unknown_attribute_reference	452
unknown_content_reference	452
start_of_prefix_mapping	452
end_of_prefix_mapping	452
exception	452
イベント関数に渡されるパラメーター	452
XML 文書のコード化文字セット	453
サポートされる EBCDIC コード・ページ	454
サポートされる ASCII コード・ページ	454
コード・ページの指定	454
例外	455
例	456
継続可能な例外コード	468
例外コードの終了	472

第 29 章 アプリケーションにおける PL/I MLE の使用 477

属性およびオプションの適用	477
DATE 属性	477
RESPECT コンパイル時オプション	478
WINDOW コンパイル時オプション	478
RULES コンパイル時オプション	479
日付パターンの理解	479
パターンおよびウィンドウ操作	480
MLE による組み込み関数の使用	481
DAYS	481
DAYSTODATE	482
日付の計算および比較の実行	482
明示的な日付計算	482
暗黙的な日付計算	483
暗黙的な日付比較	483
暗黙的な DATE の代入	484
SQL プリプロセッサによる MLE の使用	485

第 7 部 付録 487

特記事項 489

プログラミング・インターフェース情報	490
ユーザー用のマクロ	490
商標	491

参考文献 493

Enterprise PL/I 資料	493
DB2 UDB for OS/390 および z/OS	493
CICS Transaction Server	493

用語集 495

索引 511



1. SQLCA の PL/I 宣言	116	27. PL/I コンパイラー・ユーザー出口のプロシ ジャー	358
2. SQL 記述子域の PL/I 宣言	117	28. IBMUEXIT.INF ファイルの例	360
3. CHIMES プログラムのコンパイラー・リスト	139	29. Java サンプル・プログラム #2 - ストリング の引き渡し	419
4. MAKE ファイルの例	151	30. PL/I サンプル・プログラム #2 - ストリング の引き渡し	421
5. 定様式ダンプを生成する PL/I コード	184	31. Java サンプル・プログラム #3 - 整数の引き 渡し	423
6. PLIDUMP 出力の例	185	32. PL/I サンプル・プログラム #3 - 整数の引き 渡し	425
7. 静的および動的派生プロシージャ	187	33. ソート・プログラムの制御の流れ	431
8. 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	34. 入力プロシージャ用の骨組みコード	437
9. 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	234	35. E15 が、PLISRTx を呼び出すプロシージャ の外部にある場合	438
10. ストリーム・データ伝送による印刷ファイル の作成	236	36. 出力処理プロシージャ用の骨組みコード	440
11. PLITABS の宣言	238	37. PLISRTA-入力データ・セットから出力デー タ・セットへのソート	441
12. 設定済みのタブ設定を変更する場合の PL/I 構 造体 PLITABS	239	38. PLISRTB-入力処理ルーチンから出力データ・ セットへのソート	442
13. 対話式プログラムのサンプル	244	39. PLISRTC-入力データ・セットから出力処理ル ーチンへのソート	444
14. 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	249	40. PLISRTD-入力処理ルーチンから出力処理ルー チンへのソート	445
15. レコード単位データ伝送の印刷	252	41. PLISRTD-入力処理ルーチンから出力処理ルー チンへのソート	446
16. REGIONAL(1) データ・セットの作成	258	42. サンプル XML 文書	449
17. REGIONAL(1) データ・セットの更新	261	43. PLISAXA のコーディング例 - 型宣言	457
18. ワークステーション VSAM キー順データ・セ ットの作成	272	44. PLISAXA のコーディング例 - イベント構造 体	458
19. ワークステーション VSAM 順次データ・セッ トの定義とロード	276	45. PLISAXA のコーディング例 - メインルーチ ン	459
20. ワークステーション VSAM キー順データ・セ ットの定義とロード	281	46. PLISAXA のコーディング例 - イベント・ル ーチン	460
21. ワークステーション VSAM キー順データ・セ ットの更新	284	47. PLISAXA のコーディング例 - プログラム出 力	468
22. ワークステーション VSAM 直接データ・セッ トのロード	290		
23. キーによるワークステーション VSAM 直接デ ータ・セットの更新	293		
24. データベースの選択	309		
25. 修飾子によって作成されたテーブルの表示	310		
26. 生成された PL/I 宣言	311		

第 1 部 ワークステーション上での PL/I の導入

第 1 章 本書について

新機能 3

このプログラミング・ガイドは、PL/I for Windows コンパイラーを使って PL/I プログラムをコーディングしコンパイルするときの手助けとなることを目的に作成されています。

これまでは主にメインフレーム PL/I を使用しており、そのプログラムを Windows プラットフォームに移動させることを検討している場合は、特に 11 ページの『第 4 章 プラットフォーム間でのアプリケーションの移植』が参考になります。本書には他に、Windows の基本機能の理解に役立つ情報や、PL/I プログラムのコンパイル、リンク、および実行方法の説明があります。

新機能

PL/I ワークステーション・コンパイラーに追加された最新機能の一部を以下に示します。

- PL/I と Java を一緒に使用方法のヒント
- Windows 環境で dclgen を使用方法
- Open Database Connectivity 使用に関するヘルプ

第 2 章 構文図の読み方

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。

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、--->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、大文字で示されます。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。

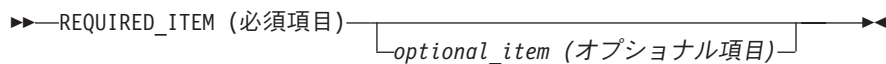
必須項目

必須項目は横線（主経路）に示します。

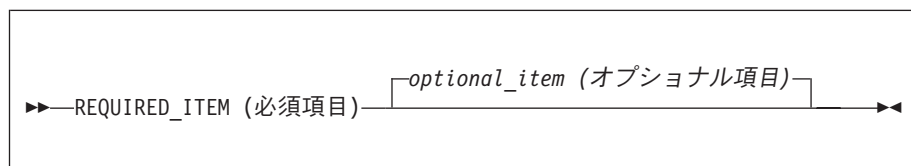


オプション項目

オプション項目は、主経路の下に示します。

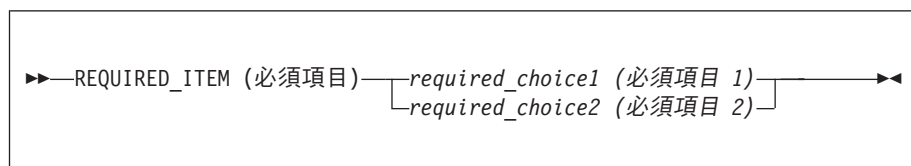


主経路より上にオプション項目を示すこともあります。これは見やすくするため、ステートメントの実行には影響を及ぼしません。

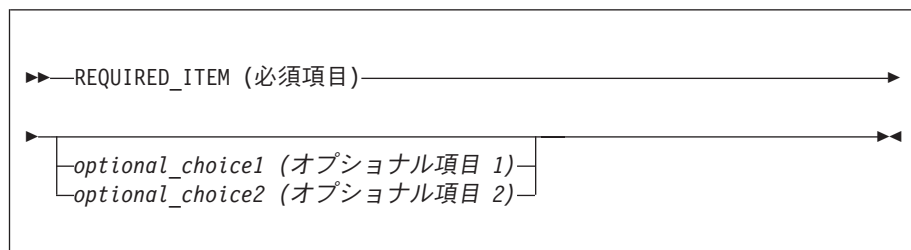


複数の必須項目またはオプション項目

2 つ以上の項目から選択できる場合は、それらの項目は縦に並べたスタックで表されます。その項目の 1 つを選択しなければならない場合は、スタックのうちの 1 つの項目が主経路上に置かれます。

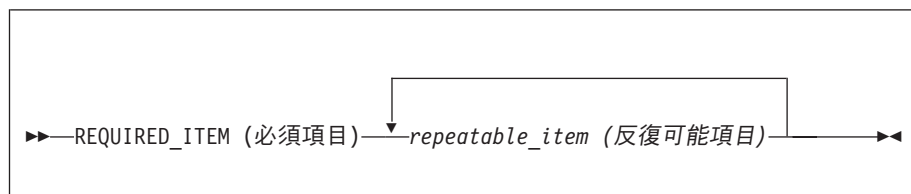


項目がオプションである場合、主経路の下にある支線の上に縦に並んだ項目として示されます。

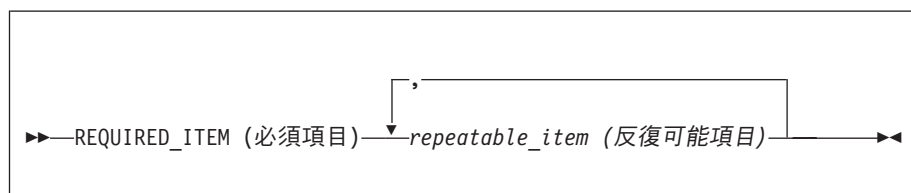


反復可能項目

主経路の上方を通過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。



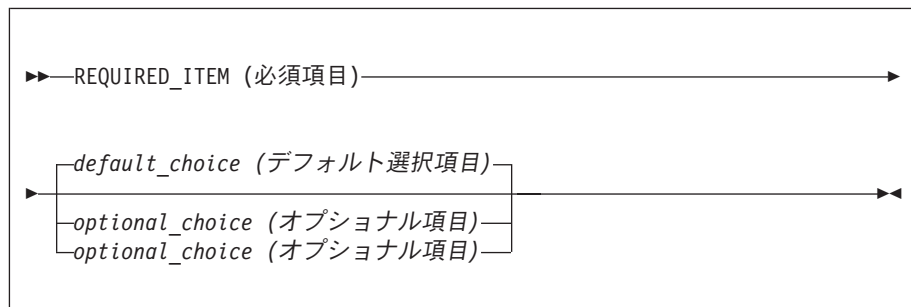
反復矢印の途中にコンマがある場合は、反復される項目をコンマで区切らなければなりません。



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。

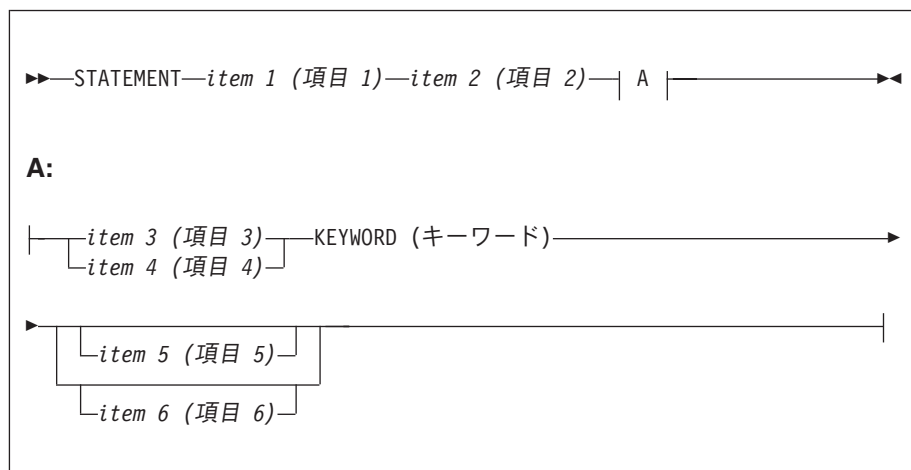
デフォルト・キーワード

デフォルト・キーワードは主経路より上に示され、それ以外の選択項目は主経路より下に示されます。



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。



第 3 章 ご意見の送付方法

本書または Debug Tool の他のマニュアルについてご意見がありましたら、IBM 発行のマニュアルに関する情報の Web ページ (<http://www.ibm.com/jp/manuals/>) よりお送りください。今後の参考にさせていただきます。(URL は、変更になる場合があります)

第 4 章 プラットフォーム間でのアプリケーションの移植

メインフレーム・アプリケーションのワークステーション上でのコンパイル	11	iSUB の定義	15
正しいコンパイル時オプションの選択	12	DBCS	15
制限された言語	12	マクロ・プリプロセッサ	15
RECORD I/O	12	プログラムの移植を助けるマクロ機能の使用	15
STREAM I/O	13	メインフレーム・アプリケーションのワークステーション上での実行	16
構造式	13	リンクの相違	16
配列式	13	ランタイムの相違をもたらすデータ表記	16
DEFAULT ステートメント	14	移植性に影響する環境の相違	19
自動変数のエクステンツ	14	ランタイムの相違を引き起こす言語エレメント	20
組み込み関数	14		

IBM メインフレーム環境には、AIX システムまたはパーソナル・コンピュータ (PC) よりも多くの、さまざまなハードウェアおよびオペレーティング・システム・アーキテクチャがあります。メインフレーム以外のオペレーティング・システムは、しばしば、ワークステーション・プラットフォームと呼ばれます。本書では、AIX および Windows のオペレーティング・システムを呼ぶのにワークステーションという用語を使用します。

基本的なプラットフォーム間の相違、および OS PL/I コンパイラと PL/I for Windows コンパイラ間の相違のために、PL/I プログラムをメインフレームとワークステーション環境の間で移動するときに、いくつかの問題が発生する場合があります。本章では、開発プラットフォーム間でのこれらの相違について説明し、以下の作業を行う際に、問題を最小化するための説明を行います。

- ワークステーション上で、エラーを出さずに、メインフレーム・アプリケーションをコンパイルする。
- ワークステーション上でメインフレーム・アプリケーションを実行する (および同じ結果を得る)。
- 後からメインフレーム上で実動モードで実行するアプリケーションを、ワークステーション上で作成、コンパイル、およびテストする。

メインフレーム・アプリケーションのワークステーション上でのコンパイル

メインフレームからワークステーションにプログラムを移動するときの最初の目標の 1 つは、これまで使用してきたアプリケーションを、エラーなしに新しい環境でコンパイルすることです。

メインフレームとワークステーションで使用される文字セットは異なります。このため、コンパイルのときにいくつかの問題が発生する場合があります。

組み込み制御文字

ソース・ファイルに '20'x より小さな 16 進値を持った文字があると、ワークステーション・コンパイラは、そのファイルの行サイズを誤って解釈する場合があります。また、ファイル自体のサイズを誤って解釈する場合さえあります。このような値をエンコードするには、16 進文字定数を使用する必要があります。

ホストから、16 進エディターで入力された値に初期化された変数を持つソース・ファイルをダウンロードする場合、それらのいくつかの値は、ホストではより大きな値を持つにもかかわらず、'20'x より小さな 16 進値を持つ場合があります。

国別文字およびその他の記号

プラットフォーム間でプログラムを移植する場合、特定のコード・ページで国別文字およびその他の記号 (PL/I コンテキストにおいて) を使用すると、エラーが発生する場合があります。これは、論理「NOT」(¬) および論理「OR」(∨) の記号、通貨記号、および PL/I ID における次のアルファベット・エクステンダーの使用についても当てはまります。

\$

@

「NOT」、「OR」、および通貨記号に関連する問題を回避するには、NOT (72 ページの『NOT』を参照)、OR (75 ページの『OR』を参照)、および CURRENCY (46 ページの『CURRENCY』を参照) コンパイル時オプションを、*PROCESS ステートメントで使用します。その他の文字に関連する問題を回避するには、NAMES (70 ページの『NAMES』を参照) コンパイル時オプションを使用して、特別言語の文字および記号を定義してください。

正しいコンパイル時オプションの選択

特定のコンパイル時オプションを選択することにより、コンパイラとプラットフォーム間で、ソース・コードの移植性を高くすることができます。例えば、LANGVL(SAA) を選択すると、コンパイラは、pre-Enterprise PL/I でサポートされていないキーワードにフラグを立て、pre-Enterprise PL/I でサポートされていない組み込み関数を認識しません。

pre-Enterprise PL/I との互換性を高めたい場合は、次のオプションを指定できます。

- DEFAULT(DESCLOCATOR EVENDEC NULL370 RETURNS(BYADDR))
- LIMITS(EXTNAME(7) NAME(31))

オプション DEFAULT(RETURNS(BYADDR)) を使用する場合、RETURNS 記述で BYVALUE 属性が指定されていないと、ワークステーション上での非 PL/I 関数の呼び出しに失敗することに注意してください。

これらの (およびその他のすべてのコンパイラの) オプションは 37 ページの『第 6 章 コンパイル時オプションの説明』にアルファベット順でリストされており、詳しく説明されています。

制限された言語

指示されている場所以外では、コンパイラは、制限された言語の使用箇所にフラグを立てます。

RECORD I/O

RECORD I/O はサポートされていますが、次の制約事項があります。

- READ/WRITE ステートメントの EVENT 文節はサポートされません。
- UNLOCK ステートメントはサポートされません。

メインフレーム・アプリケーションのワークステーション上でのコンパイル

- 以下のファイル属性はサポートされません。
 - BACKWARDS
 - EXCLUSIVE
 - TRANSIENT
- ENVIRONMENT 属性の以下のオプションはサポートされませんが、それらの使用箇所には、LANGVLV(LNOEXT) のもとでのみフラグが付けられます。
 - ADDBUFF
 - ASCII
 - BUFFERS
 - BUFND
 - BUFNI
 - BUFOFF
 - INDEXAREA
 - LEAVE
 - NCP
 - NOWRITE
 - REGIONAL(2)
 - REGIONAL(3)
 - REREAD
 - SIS
 - SKIP
 - TOTAL
 - TP
 - TRKOFL

STREAM I/O

STREAM I/O はサポートされていますが、PUT/GET DATA ステートメントに次の制約事項が適用されます。

- DEFINED 変数が BIT または GRAPHIC の場合、または POSITION 属性を持つ場合、DEFINED はサポートされません。
- DEFINED は、その基本変数が配列スライス、または定義された変数とは異なる次元数を持った配列の場合、サポートされません。

構造式

次の条件が両方とも満たされない場合は、引数としての構造式はサポートされません。

- パラメーター記述がある。
- パラメーター記述がすべての定数エクステンントを指定する。

配列式

配列式は、既知のサイズのスカラーの配列でない場合、ユーザー関数への引数としては許可されません。したがって、算術型のスカラーの配列はユーザー関数に渡されますが、可変長ストリングの配列の場合は、問題が発生することがあります。

以下の例は、呼び出しでサポートされる数値配列式を示します。

```
dc1 x entry, (y(10),z(10)) fixed bin(31);  
  
call x(y + z);
```

次のプロトタイプ化されていない呼び出しは、不明サイズのストリング式を必要とするため、フラグが付けられます。

```
dc1 a1 entry;  
dc1 (b(10),c(10)) char(20) var;  
  
call a1(b || c);
```

しかし、次のプロトタイプ化された呼び出しには、フラグは付きません。

```
dc1 a2 entry(char(30) var);  
dc1 (b(10),c(10)) char(20) var;  
  
call a2(b || c);
```

DEFAULT ステートメント

因数化されたデフォルト指定はサポートされません。

次のようなステートメントはサポートされません。

```
default ( range(a:h), range(p:z) ) fixed bin;
```

しかし、上記のステートメントを次のようにして、サポートされるステートメントに変更することができます。

```
default range(a:h) fixed bin, range(p:z) fixed bin;
```

DEFAULT キーワード後の「(」の使用は、ANSI 規格のもとにおけるのと同じ目的で予約されています。この規格では、DEFAULT キーワード後に、属性の論理述部を括弧に入れることが許可されています。

自動変数のエクステンツ

自動変数のエクステンツは、自動変数が宣言されるプロシーチャー内にネストされた関数によっては設定できません。また、自動変数の前にエンタリー変数を宣言しない限り、エンタリー変数によっても設定できません。

組み込み関数

組み込み関数のサポートには、以下の例外または制約事項があります。

- PLITEST 組み込み関数はサポートされません。
- 疑似変数は、以下においてはサポートされません。
 - PUT ステートメントの STRING オプション
- DO ループで許可される疑似変数は、以下のものに制限されます。
 - IMAG
 - REAL
 - SUBSTR
 - UNSPEC
- POLY 組み込み関数には、以下の制約事項があります。
 - 最初の引数は REAL FLOAT でなければなりません。
 - 2 番目の引数はスカラーでなければなりません。
- COMPLEX 疑似変数はサポートされません。
- IMS 組み込みサブルーチン PLICANC、PLICKPT、および PLIREST は、サポートされません。

iSUB の定義

iSUB 定義のサポートは、スカラーの配列に制限されています。

DBCS

DBCS は、以下のものにおいてのみ使用できます。

- G および M の定数
- ID
- コメント

G リテラルは、後に DBCS G または SBCS G が付いた DBCS 引用符で開始および終了できます。

マクロ・プリプロセッサ

ストリング定数に続く接尾部は、正しい PL/I の接尾部であろうとなかろうと、ストリングの終了引用符と接尾部の最初の文字の間に区切り文字を挿入しない限り、マクロ・プリプロセッサによっては置き換わりません。

OS PL/I V2R1 コンパイラでこの変更が導入されたため、これは、PL/I for MVS & VM コンパイラと、PL/I for MVS & VM コンパイラまたは OS PL/I V2Rx コンパイラのいずれかとの間の相違ではないことに注意してください。したがって、この制約事項にはフラグは付きません。

例として、以下を考えてみます。

```
%DCL (GX, XX) CHAR;  
%GX='||FX';  
%XX='||ZZ';  
DATA = 'STRING'GX;  
DATA = 'STRING'XX;  
DATA = 'STRING' GX;  
DATA = 'STRING' XX;
```

OS PL/I V1 のもとでは、これは次のソースを生成します。

```
DATA = 'STRING' ||FX;  
DATA = 'STRING' ||ZZ;  
DATA = 'STRING' ||FX;  
DATA = 'STRING' ||ZZ;
```

それに対し、PL/I for MVS & VM のもとでは、それは以下を生成します。

```
DATA = 'STRING'GX;  
DATA = 'STRING'XX;  
DATA = 'STRING' ||FX;  
DATA = 'STRING' ||ZZ;
```

プログラムの移植を助けるマクロ機能の使用

多くの場合、移植性の問題は、マクロ機能を使用することによって回避できます。マクロ機能には、プラットフォーム固有のコードを分離する機能があるためです。例えば、特定プラットフォームのコンパイルにプラットフォーム固有コードを組み込み、別のプラットフォームのコンパイルからそれを除外することができます。

メインフレーム・アプリケーションのワークステーション上でのコンパイル

PL/I for Windows マクロ機能の COMPILETIME 組み込み関数は、「DD.MMM.YY」の形式を使用して日付を戻します。他方、OS PL/I マクロ機能の COMPILETIME 組み込み関数は、「DD MMM YY」の形式を使用して日付を戻します。

これにより、PL/I for Windows コンパイラおよびすべてのバージョンのメインフレーム PL/I コンパイラによって正しくコンパイルされる、次のような条件付きシステム依存コードを含むコードを作成できます。

```
%dcl compiletime builtin;  
  
%if substr(compiletime,3,1) = '.' %then  
  %do;  
    /* Windows PL/I code */  
  %end;  
%else  
  %do;  
    /* OS PL/I code */  
  %end;
```

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。

メインフレーム・アプリケーションのワークステーション上での実行

メインフレームからソース・プログラムをダウンロードし、ワークステーション・コンパイラでエラーなしにコンパイルしたら、次は、そのプログラムを実行します。ワークステーション上において、メインフレーム上におけるのと同じ結果を得たい場合は、基盤となるハードウェアまたはソフトウェアのアーキテクチャーによって変化する PL/I 言語の要素および動作について知っておく必要があります。

リンクの相違

作成するすべての .EXE には、メインルーチン (OPTIONS(MAIN) を含むプロシージャ) がちょうど 1 つ含まれる必要があります。メインルーチンが存在しない場合、リンカーは、プログラムに開始アドレスが存在しないというエラーを報告します。メインルーチンが複数存在する場合、リンカーは、名前 main に対して重複する参照があるというエラーを報告します。

作成するすべての .DLL には、DLLINIT コンパイル時オプションでコンパイルされたモジュールが少なくとも 1 つなければなりません (54 ページの『DLLINIT』を参照)。

ランタイムの相違をもたらすデータ表記

ほとんどのプログラムは、データ表記を気にしなくても、同じように動作しますが、使用するプログラムでもそうなるようにするためには、以下のセクションで説明する相違について理解しておく必要があります。

ワークステーション・コンパイラは、データおよび浮動小数点の操作を、メインフレームと同じように処理するようオペレーティング・システムに指示するオプションをサポートします。コードをワークステーションに移すときに変更しなければならない場合のあるすべてのメインフレーム・アプリケーションには、指定すべき DEFAULT コンパイル時オプションのサブオプションがあります。

- ASCII の代わりにの DEFAULT(EBCDIC)
- IEEE の代わりにの DEFAULT(HEXADEC)
- DFT(E(IEEE)) の代わりにの DEFAULT(E(HEXADEC))
- NATIVE の代わりにの DEFAULT (NONNATIVE)
- NATIVEADDR の代わりにの DEFAULT (NONNATIVEADDR)

コンパイル時オプションの詳細については、46 ページの『DEFAULT』を参照してください。

ASCII 対 EBCDIC

ワークステーション・オペレーティング・システムは ASCII 文字セットを使用するのに対し、メインフレームは EBCDIC 文字セットを使用します。つまり、ほとんどの文字は異なる 16 進値を持ちます。例えば、ブランクの 16 進値は、ASCII 文字セットでは '20'x であり、EBCDIC 文字セットでは '40'x です。

したがって、EBCDIC 16 進値の文字データに依存するコードは、ASCII を使用して実行すると、論理的に失敗する可能性があります。例えば、'40'x と比較してある文字がブランクかどうかをテストするコードは、ASCII を使用して実行すると失敗します。同じように、'OR' および '80'b4 を使用して文字を大文字に変更するコードは、ASCII を使用して実行すると失敗します。(ただし、TRANSLATE 組み込み関数を使用して大文字に変換するコードは、失敗しません。)

ASCII 文字セットでは、数字は、'30'x から '39'x の 16 進値を持ちます。ASCII の小文字「a」は 16 進値 '61'x を持ち、大文字「A」は 16 進値 '41'x を持ちます。EBCDIC 文字セットでは、数字は、'F0'x から 'F9'x の 16 進値を持ちます。EBCDIC では、小文字「a」は 16 進値 '81'x を持ち、大文字「A」は 16 進値 'C1'x を持ちます。これらの相違によって、以下の興味深い結果がもたらされます。

EBCDIC では「a」<「A」が成り立つが、ASCII では成り立たない。

EBCDIC では「A」<「1」が成り立つが、ASCII では成り立たない。

EBCDIC では、 $x \geq 0$ のときは、ほとんどの場合、 x は数字を意味するが、ASCII ではこれは当てはまらない。

このような相違があるため、文字ストリングのソートの結果は EBCDIC と ASCII では異なります。多くのプログラムでは、このことの影響はありませんが、プログラムが、いくつかの文字ストリングをソートする正確な順序に依存する場合は、論理エラーが発生する可能性に注意する必要があります。

ASCII から EBCDIC への変換については、365 ページの『データ変換表の使用』を参照してください。

ネイティブ 対 非ネイティブ

パーソナル・コンピュータ (PC) は、メインフレームまたは AIX の形式と比較すると、バイトが反転された形式で整数を保持します。このことは、例えば、値 258 を保持する FIXED BIN(15) 変数 (256+2 に等しい) は、Windows のストレージでは '0201'x、AIX またはメインフレームでは '0102'x として保持されることを意味します。同じ値を持った FIXED BIN(31) 変数は、Windows では '02010000'x、AIX またはメインフレームでは '00000102'x として保持されます。

AIX およびメインフレームの表記は、ビッグ・エンディアン (Big End In) と呼ばれます。

Windows の表記は、逆に、リトル・エンディアン (Little End In) と呼ばれます。

内部表記のこの相違は、以下のものに影響します。

- 2 バイト以上を必要とする FIXED BIN 変数
- OFFSET 変数
- VARYING スtringの長さ接頭部
- 序数データおよびエリア・データ

ほとんどのプログラムで、この相違は問題を引き起こしません。ただし、使用するプログラムが 16 進値の整数に依存する場合は、論理エラーの可能性に注意する必要があります。そのような従属関係は、FIXED BINARY 引数を持った UNSPEC 組み込み関数を使用する場合、または BIT 変数が FIXED BINARY 変数のアドレスに基づく場合に存在する可能性があります。

プログラムが、ポインターを整数と同様に扱う場合は、データ表記の相違が問題を引き起こす場合があります。DEFAULT(NONNATIVE) を指定するときは、ほとんどの場合、DEFAULT(NONNATIVEADDR) も指定する必要があります。

選択した宣言で NONNATIVE 属性を指定できます。例えば、以下のステートメントの代入によって、構造体のすべての FIXED BIN 値は非ネイティブからネイティブに変換されます。

```

dcl
  1 a1 native,
    2 b   fixed bin(31),
    2 c   fixed dec(8,4),
    2 d   fixed bin(31),
    2 e   bit(32),
    2 f   fixed bin(31);
dcl
  1 a2 nonnative,
    2 b   fixed bin(31),
    2 c   fixed dec(8,4),
    2 d   fixed bin(31),
    2 e   bit(32),
    2 f   fixed bin(31);

a1 = a2;

```

IEEE 対 HEXADEC

ワークステーション・オペレーティング・システムは、IEEE 形式を使用して浮動小数点データを表しますが、メインフレームは伝統的に、16 進形式を使用します。

表 1 は、正規化浮動小数点の IEEE と 16 進との相違の要約です。

表 1. 正規化 IEEE 対正規化 16 進

仕様	IEEE (AIX)	IEEE (PC)	16 進数
値のおよその範囲	±10E-308 から ±10E+308	±3.30E-4932 から ±1.21E+4932	±10E-78 から ±10E+75
FLOAT DECIMAL の最大精度	32	18	33
FLOAT BINARY の最大精度	106	64	109
FLOAT DECIMAL 指数の最大桁数	4	4	2
FLOAT BINARY 指数の最大桁数	5	5	3

16 進浮動小数点は、short 浮動小数点、long 浮動小数点、および拡張浮動小数点で同じ最大および最小の指数値を持ちますが、IEEE 浮動小数点は、short 浮動小数点、long 浮動小数点、および拡張浮動小数点で異なる最大および最小の指数値を持ちます。つまり、1E74 (PL/I では、属性 FLOAT DEC(1) を持つ必要がある) は有効な 16 進の short 浮動小数点ですが、有効な IEEE の short 浮動小数点ではありません。

ほとんどのプログラムで、FIXED BIN 変数の表記の相違が問題を引き起こさないのとまったく同じように、これらの相違は問題を引き起こしません。ただし、使用するプログラムが、16 進値の浮動小数点値に依存する場合は、コーディングの際に注意が必要です。

また、FIXED BIN の計算は、上で説明した内部表記に依存しない同じ結果をもたらしますが、浮動小数点の計算は、必ずしも同じ結果をもたらすとは限りません。これは、浮動小数点値を表現する方法に相違があるためです。このことは、特に、short 浮動小数点および拡張浮動小数点について当てはまります。

EBCDIC DBCS 対 ASCII DBCS

EBCDIC DBCS スtringはシフト・コードで囲みますが、ASCII DBCS スtringはシフト・コードでは囲みません。同じ文字を表現するために使用する 16 進値も異なります。

この場合も、ほとんどのプログラムでこのことは問題になりません。使用するプログラムが、16 進値の GRAPHIC スtringに依存する場合、または混合文字およびグラフィック・データを含む文字スtringに依存する場合は、コーディングの際に注意が必要です。

移植性に影響する環境の相違

ワークステーションとメインフレームのプラットフォームの間には、データ表記以外に、プログラムの移植性に影響を与える可能性のある相違がいくつかあります。このセクションでは、これらの相違のいくつかについて説明します。

ファイル名

PC 上のファイル命名規則は、メインフレームのものと大きく異なります。例えば、次のファイル名は PC では有効ですが、メインフレームでは無効です。

```
d:¥programs¥data¥myfile.dat
```

PL/I ソース内のファイル名を、OPEN および FETCH ステートメントの TITLE オプションの一部として使用すると、このことが移植性に影響を与えます。

ファイル属性

PL/I では、ファイル宣言において、多くのファイル属性を、ENVIRONMENT 属性の一部として指定することができます。これらの属性の多くはワークステーションでは意味を持ちません。その場合、コンパイラはそれらを無視します。使用するプログラムが、これらの属性を無視しないことを前提としている場合、そのプログラムは、正常に移植されない可能性があります。

制御コード

メインフレームで特定の意味を持たないいくつかの文字は、ワークステーションによって制御文字として解釈されます。LF、LFEof、CRLF、または CRLFEOF

メインフレーム・アプリケーションのワークステーション上での実行

のいずれかの TYPE を持つデータ・ファイルは、このことによって、誤って処理される可能性があります。そのようなファイルには、次の文字を入れないようにしてください。

```
'0A'x (「LF - 改行」)  
'0D'x (「CR - 復帰」)  
'1A'x (「EOF - ファイルの終わり」)
```

例えば、下のコード内のファイルが TYPE(CRLF) を持つ場合、2573 は 16 進値 '0D0A'x を持つため、WRITE ステートメントはオン・コード 1041 の ERROR 条件を発生します。ファイルが FIXED、VARLS、または VARMS のいずれかの TYPE を持つ場合は、このようにはなりません。

```
dc1  
  1 a native,  
  2 b char(10),  
  2 c fixed bin(15),  
  2 d char(10);  
  
dc1 f file output;  
  
a.b = 'alpha';  
a.c = 2573;  
a.d = 'omega';  
  
write file(f) from(a);
```

装置依存の制御コード

プログラムまたはファイルにおいて装置依存の (プラットフォーム固有の) 制御コードを使用すると、制御コードを必ずしもサポートしないその他のプラットフォームにそれらを移植するときに、エラーが発生する場合があります。

プラットフォームに対する固有性が非常に強い他のすべてのコードと同じように、アプリケーションを別のプラットフォームに移すときに簡単に置き換えることができるように、そのようなコードをできる限り分離することが最善の方法です。

ランタイムの相違を引き起こす言語エレメント

コンパイラーによる言語のインプリメンテーションの相違のために、PL/I for Windows と OS PL/I のもとで、プログラムが異なって実行される原因となるいくつかの言語エレメントもあります。次のそれぞれの項目は、PL/I for Windows の動作の観点から説明します。

p <= 7 の場合、FIXED BIN(p) は 1 バイトにマップされる

精度が 7 以下で、FIXED BIN として宣言された変数がある場合、それらは、OS PL/I では 2 バイトを占有しますが、PL/I for Windows では 1 バイトのストレージを占有します。この変数が構造体の一部の場合は、通常、このことにより、構造体のマップ方法が変更されるため、プログラムの実行方法にも影響がでます。例えば、メインフレームで作成されたファイルから構造体を読み込まれる場合は、少ないバイト数が読み込まれることになります。

この相違を回避するには、変数の精度を 8 から 15 まで (両端を含む) の値に変更してください。

AREA の INITIAL 属性は無視される

PL/I for Windows 製品では AREA の INITIAL 属性が無視されてしまうので、それを避けて INITIAL 文節は代入ステートメントに変換します。

メインフレーム・アプリケーションのワークステーション上での実行

例えば、次のコード断片では、配列のエレメントは a1、a2、a3、および a4 には初期化されません。

```
dc1 (a1,a2,a3,a4) area;  
dc1 a(4) area init( a1, a2, a3, a4 );
```

しかし、コードを次のように書き直せば、希望するように、配列を初期化できます。

```
dc1 (a1,a2,a3,a4) area;  
dc1 a(4) area;  
  
a(1) = a1;  
a(2) = a2;  
a(3) = a3;  
a(4) = a4;
```

ERROR メッセージの発行

ERROR 条件が発生した場合、以下の 2 つの条件が満たされると、PL/I for Windows では ERROR メッセージは発行されません。

- ERROR ON ユニットが確立している。
- ブロック外に制御を移すために GOTO を使用して、ERROR ON ユニットがその条件からリカバリーしている。

ERROR メッセージは、SYSPRINT データ・セットにではなく、STDERR に送られます。デフォルトでは、これが端末になります。SYSPRINT が端末に送られる場合は、SYSPRINT バッファ内すべての出力 (まだ SYSPRINT には書き込まれていない) は、ERROR メッセージが書き込まれる前に書き込まれます。

ADD、DIVIDE、および MULTIPLY はスケールされた FIXED BIN を戻さない

デフォルトである RULES(IBM) コンパイル時オプションでは、変数は、非ゼロのスケール因数を持った FIXED BIN として宣言できます。2 項演算、接頭部演算、および比較演算は、メインフレームと同じように、スケールされた FIXED BIN に関して実行されます。しかし、ADD、DIVIDE、または MULTIPLY 組み込み関数が、非ゼロ因数を持つ引数を持つか、非ゼロのスケール因数を持つ結果を指定する場合、PL/I for Windows コンパイラは、メインフレーム・コンパイラと同じように、組み込み関数を FIXED BIN としてではなく、FIXED DEC として評価します。

例えば、PL/I for Windows コンパイラは、下の代入ステートメントの DIVIDE 組み込み関数を FIXED DEC 式として評価します。

```
dc1 (i,j) fixed bin(15);  
dc1 x      fixed bin(15,2);  
.  
.  
.  
x = divide(i,j,15,2)
```

OVERFLOW および ZERODIVIDE の使用可能化

OVERFLOW および ZERODIVIDE の場合、ERROR 条件は、次の条件のもとで発生します。

- OVERFLOW または ZERODIVIDE が発生しており、対応する ON ユニットが入力される。
- GOTO ステートメントによって、制御が ON ユニットを離れない。

第 2 部 プログラムのコンパイルおよびリンク

第 5 章 プログラムのコンパイル

簡単な練習問題	25	IBM.PPINCLUDE	31
HELLO プログラム	25	IBM.PPMACRO	31
コンパイル時オプションの使用	26	IBM.PPSQL	31
製品同梱のサンプル・プログラムの使用	26	IBM.PPCICS	32
ソース・プログラムのコンパイルの準備	27	IBM.SOURCE	32
プログラム・ファイルの構造	27	IBM.SYSLIB	32
PROCESS との PROCEDURE ステートメント の併用	27	IBM.PRINT	32
INCLUDE 処理	28	IBM.OBJECT	33
%OPTION ディレクティブ	28	IBM.DECK	33
%LINE ディレクティブ	29	INCLUDE	33
マージン	29	TMP	33
プログラム・ファイルのフォーマット	29	コンパイラーを呼び出す PLI コマンドの使用	33
行継続	29	コンパイル時オプションを指定する場所	34
コンパイル時環境変数の設定	30	IBM.OPTIONS と IBM.PPxxx 環境変数	34
IBM.OPTIONS	31	PLI コマンド	34
		%PROCESS ステートメント	35

この章では、最初に、単純な PL/I プログラムのコンパイル、リンク、および実行の各方法を説明します。残りの部分は、コンパイル環境の設定の詳細説明に充てられています。

簡単な練習問題

Windows 環境での PL/I の使用方法を把握するため、単純なプログラムをコンパイル、リンク、および実行してみます。

HELLO プログラム

以下は、コンピューター画面上に文字ストリング「Hello!」を表示するプログラムを作成する手順です。

1. ソース・プログラムを作成します。

次の PL/I ステートメントを含むファイル HELLO.PLI を作成します。

```
Hello: proc options(main);  
        display('Hello!');  
end Hello;
```

各行の 1 桁目を空白にします。デフォルトでは、コンパイラーは、列 2 から 72 にある文字しか認識しません。(詳しくは、66 ページの『MARGINS』を参照してください。)

ファイルをディスクに保管します。

2. プログラムをコンパイルします。

ウィンドウまたはフルスクリーン・セッションで、HELLO.PLI ファイルを格納しているディレクトリーに進み、次のコマンドを入力します。

```
pli hello
```

コンパイラーによって、画面上にコンパイルに関する情報が表示され、現行ディレクトリーにオブジェクト・ファイル (HELLO.OBJ) が作成されます。

3. プログラムをリンクします。

ディレクトリを変えないで、次のコマンドを入力します。

```
ilink hello.obj
```

このコマンドによって、必要なライブラリー・ファイル (LIBS コンパイル時オプションによって指定したもの) にファイル HELLO.OBJ が結合され、同じディレクトリ内にファイル HELLO.EXE (実行可能プログラム) が作成されます。

リンク・コマンドでは、パラメーターが指定されていないため、デフォルトが使用されます。(リンク・コマンドで使用可能なオプションは、149 ページの『第 9 章 プログラムのリンク』に説明されています。)

4. プログラムを実行します。

ディレクトリを変えないで、次のコマンドを入力します。

```
hello
```

このコマンドによって、モニター上に Hello! を表示する HELLO.EXE プログラムが起動されます。

プログラマーは通常、作業をより簡単にするために、コンパイル、リンク、および実行用のコマンドをコマンド・ファイル (CMD) にまとめて記載します。

コンパイル時オプションの使用

プログラムのコンパイル準備を行う場合、使用可能なコンパイル時オプションのサブセットを検討します。オプションの省略形も含め、コンパイル時オプションの完全な説明については、37 ページの『第 6 章 コンパイル時オプションの説明』を参照してください。

以下の例は、コンパイル・コマンドの一部として、オプションを指定する方法を具体的に示したものです。

```
pli filename (source attributes(full))
```

source

このオプションによって、ソース・コードとコンパイラー・メッセージが、コンパイラー・リスト・ファイル (例えば、HELLO.LST など) に保管されます。

attributes(full)

このオプションによって、實際上、各プログラマー定義 ID に対するすべての属性が、コンパイラー・リストに記載されることになります。

製品同梱のサンプル・プログラムの使用

製品には、複数のサンプル・プログラムが同梱され、その中の一部が、このマニュアルの別の箇所にそれぞれ記載されています。

Windows の場合、サンプル・プログラムは、.¥SAMPLES ディレクトリにインストールされています。README ファイル smwread.me が、サンプル・プログラムに添付されています。

ソース・プログラムのコンパイルの準備

ソース・プログラムのコンパイル前に、コンパイラーが要求するソース・プログラム・ファイルの構造と形式を確認しておく必要があります。

プログラム・ファイルの構造

PL/I アプリケーションは、複数のコンパイル単位から構成される場合があります。各コンパイル単位を別々にコンパイルしてから、結果のオブジェクト・ファイルをリンクして、アプリケーション全体を作成する必要があります。

コンパイル単位は、1 つのメイン・ソース・ファイルと任意の数のインクルード・ファイルから構成されています。インクルード・ファイルは、コンパイル時にメインプログラムに実際に組み込まれるため、別々にコンパイルすることはしません。コンパイラーでは、ソース・ファイル名またはインクルード・ファイル名に DBCS を使用することは許されていません。

%PROCESS ステートメントまたは *PROCESS ステートメントを必要とするプログラムの場合、同ステートメントは、ソース・ファイル内の 1 番目の行に置く必要があります。ブランク行やコメント行を除いて、そのステートメントの後の最初の行は、PACKAGE ステートメントまたは PROCEDURE ステートメントにする必要があります。ブランク行やコメント行を除いて、ソース・ファイルの最終行は、PACKAGE ステートメントまたは PROCEDURE ステートメントと対になる END ステートメントにする必要があります。

以下の例では、ソース・ファイルを正しくフォーマット設定する方法について示します。

PROCESS との PROCEDURE ステートメントの併用

```
%PROCESS ;
%PROCESS ;
%PROCESS ;

/* optional comments */

procedure_Name: proc( ... ) options( ... );
...
end procedure_Name;
```

PROCESS との PACKAGE ステートメントの併用:

```
*PROCESS ;
*PROCESS ;
*PROCESS ;

/* optional comments */

package_Name: package exports( ... ) options( ... );
...
end package_Name;
```

コンパイルするソース・ファイルは、*PROCESS ステートメントで区切られた複数のプログラムから構成できます。*PROCESS ステートメントの最初のセット以外はすべて無視され、コンパイラーは、最初のプロシージャーの前に PACKAGE EXPORTS(*) ステートメントがあると仮定します。

INCLUDE 処理

追加の PL/I ファイルは、コンパイル単位内の指定したポイントに、%INCLUDE ステートメントを使用してインクルードすることができます。%INCLUDE ステートメントの構文については、PL/I 言語解説書を参照してください。

ストリングを使用してインクルードするファイルを指定すると、コンパイラによって、使用したストリングで指定した名前とまったく同じ名前のファイルが検索されます。より伝統的な PL/I のいずれかの方法を使用し、*ddname* と *member name* または *member name* のみを用いて、インクルード・ファイルを指定すると、ファイル拡張子が、コンパイラによって、*member name* に追加されます。

メンバー名に追加されるファイル拡張子は、INCLUDE コンパイラ・オプションを使用して指定できます。例えば、INCLUDE(EXT(CPY)) と INCLUDE オプションを指定した場合、コンパイラは、以下のいずれかのステートメントを認識すると、ファイル *member.cpy* をインクルードしようとします。

```
%include member;  
%include ddname(member);
```

コンパイラは、以下の順序で、上記のファイルを検索します。

1. %include ステートメントで *ddname* を指定した場合、環境変数 IBM.DDNAME に指定されているディレクトリー
2. 環境変数 IBM.SYSLIB に指定されているディレクトリー
3. 環境変数 INCLUDE に指定されているディレクトリー
4. 現行ディレクトリー

INCLUDE コンパイラ・オプションに複数の拡張子を指定した場合、コンパイラは、上記のディレクトリーのすべてを、1 番目の拡張子を使用して、次に、2 番目の拡張子を使用して、すべてのディレクトリーをというように検索していきます。

%OPTION ディレクティブ

%OPTION ディレクティブは、ソース・コードの 1 つのセグメントに対して、コンパイル時オプションから選択された一部のオプションのいずれかを指定するために使用します。指定したオプションは、以下のいずれかの場合が成立するまで有効となります。

- 別の %OPTION ディレクティブによって、最初のオプションをオーバーライドする代替のコンパイル時オプションが指定された場合。
- %PUSH ディレクティブを使用して保管されたコンパイル時オプションが、%POP ディレクティブを使用してリストアされた場合。

%OPTION ディレクティブとともに使用できるコンパイル時オプションまたはディレクティブが、以下をインクルードする場合。

- LANTLRVL(SAA)
- LANTLRVL(SAA2)

オプションの説明については、37 ページの『第 6 章 コンパイル時オプションの説明』を参照してください。

%LINE ディレクティブ

%LINE ディレクティブは、デバッグ用に生成されたメッセージおよび情報の中で、その次の行を指定された行およびファイルとして取り扱うように指定します。

%LINE ディレクティブが認識されるためには、文字 '%LINE' が、入力行の 1 列から 5 列に存在する必要があります (また、反対に、この 5 文字で始まる行はすべて、%LINE ディレクティブとして取り扱われます)。line-number は 7 桁以下の整数値とし、file-specification は引用符で囲んではいけません。セミコロンの後に指定された文字はすべて、無視されます。

このような行が、実際にどのような外観をしているかは、オプションの PPTRACE MACRO や MDECK を使用してプログラムをコンパイルすると分かります。

マージン

デフォルトでは、コンパイラーは、ソース・プログラム・ファイルの 1 列にあるデータを無視し、左から 72 列分のスペースを右マージンとして設定します。

デフォルトのマージン設定は変更することができます (66 ページの『MARGINS』を参照してください)。デフォルト設定を受け入れた場合は、ソース・コードは、2 番目の列から始める必要があります。

注: %PROCESS (または *PROCESS) ステートメントは、マージン規則に対する例外であり、1 番目の列から始める必要があります。%PROCESS ステートメントの詳細については、35 ページの『%PROCESS ステートメント』を参照してください。

プログラム・ファイルのフォーマット

Windows オペレーティング・システムの下で動作するコンパイラーは、ソース・ファイルの内容が、ASCII フォーマットかつ CR-LF タイプ¹から構成されていることを要求します。ワークステーション上で作成されたファイルの場合、そのフォーマットは正しくなります。ただし、別のマシン環境からファイルを転送する場合には、ファイル転送ユーティリティが必要な変換 (ASCII と CR-LF への変換) を行うことを確認します。

コンパイラーは、X'00' から X'1F' の範囲の文字を制御コードとして解釈します。プログラム内でこの範囲内にある文字を使用すると、結果は予測不能となります。

行継続

コンパイル時、MARGINS オプションで定義された右側マージン設定値よりも短いソース行は、その長さが右側マージン値と等しくなるように、右側にブランク文字が埋め込まれます。例えば、IBM-default MARGINS (2,72) を使用した場合、72 文字長より短い行は、72 文字長になるように右側にブランク文字が埋め込まれます。

長い ID 名が右マージンを越える場合は、その名前を 2 行に分割しないで、名前全体を次行に移動するようにします。

1. CR-LF タイプ・ファイルは、各行が CR-LF 文字で区切られている可変長の行から構成されています。CR と LF は、「復帰」と「改行」(16 進数値でそれぞれ、0D と 0A) を示す特別な ASCII 文字です。コンパイラーは、CR-LF、LF-CR、CR、または LF をレコード区切り文字と解釈します。16 進数値 1A は、ファイルの終了を示します。

プログラムの行が、右側マージンと正確に重なる場合は、その行の最終文字は、次行の右側マージン内の最初の文字と、ブランク文字を挟まずに連結されます。

右側マージン設定を越えて延びるストリングの場合は、そのストリングのテキストを次行 (複数行も可) に持ち越すことができます。長いストリングは、(各ストリングが 1 行に収まるように) 一連の短いストリングに分割し、連結することをお勧めします。例えば、次のようなコーディングは避けます。

```
do;
  if x > 200 then
    display ('This is a long string and requires more than one line to
      type it into my program');
  else
    display ('This is a short string');
end;
```

次のような一連のステートメントを使用します。

```
do;
  if x > 200 then
    display ('This is a long string and requires more than '
      ||'one line to type it into my program');
  else
    display ('This is a short string');
end;
```

コンパイル時環境変数の設定

コンパイル時環境変数の設定方法は、使用中のオペレーティング・システムに依存します。

Windows では、環境変数は「システム」ウィンドウで設定されています (「メイン」をダブルクリックしてから、「コントロール パネル」をダブルクリックして、進みます)。「システム」ウィンドウで、「設定」をクリックして、「ユーザー環境変数」のリストに新規の項目を追加します。Windows の「システム」ウィンドウで設定したオプションは、.CMD ファイルを使用するか、コマンド行でオプションを指定するかしてオーバーライドしない限り、コンピューターをブートすると有効になります。

環境変数およびその使用方法の詳細については、お手元のシステムに関する資料を参照してください。

コンパイラーには、複数の環境変数が用意されています。次のデフォルト設定をカスタマイズする場合に使用できます。

- コンパイラーの入力と出力の位置
- コンパイル時オプション。

コンパイラーの入力と出力のデフォルトの位置は、現行ディレクトリーとなります。各コンパイル時オプションに対する IBM 提供のデフォルトは、37 ページの『第 6 章 コンパイル時オプションの説明』に指定されています。

一部のコンパイラー環境変数は、ディレクトリー・パスを指定します (ファイル名や拡張子は含みません)。パスがコンパイラー入力用の場合は、個々のパスを (最後のパスを除いて) セミコロンで区切る必要があります。パスがコンパイラー出力用の場合は、特定のディレクトリーに対するパスのみが許されます。

IBM.OPTIONS

IBM.OPTIONS 環境変数には、コンパイラ・オプション設定を指定します。次に例を示します。

```
set ibm.options=xref attributes
```

IBM.OPTIONS 環境変数への文字ストリングの代入構文は、PLI コマンドで指定するコンパイル時オプションに要求されるものと同じです (33 ページの『コンパイラを呼び出す PLI コマンドの使用』を参照してください)。

この環境変数を使用して適用する変更内容とともにデフォルトが、新規のデフォルトになります。PLI コマンドまたはソース・プログラムで指定したオプションがある場合、これらのデフォルトはオーバーライドされます。

IBM.PPINCLUDE

IBM.PPINCLUDE 環境変数には、インクルード・プリプロセッサ・オプション設定を指定します。次に例を示します。

```
set ibm.ppinclude=id(++include)
```

IBM.PPINCLUDE 環境変数への文字ストリングの代入構文は、PLI コマンドで指定するコンパイル時オプションに要求されるものと同じです (33 ページの『コンパイラを呼び出す PLI コマンドの使用』を参照してください)。

この環境変数を使用して適用する変更内容とともにデフォルトが、新規のデフォルトになります。IBM. OPTIONS 環境変数、PLI コマンド、またはソース・プログラム内に PP(INCLUDE) オプションで指定したオプションがある場合、それらのデフォルトは、オーバーライドされます。

IBM.PPMACRO

IBM.PPMACRO 環境変数には、マクロ機能オプション設定を指定します。次に例を示します。

```
set ibm.ppmacro=xref print
```

IBM.PPMACRO 環境変数への文字ストリングの代入構文は、PLI コマンドで指定するコンパイル時オプションに要求されるものと同じです (33 ページの『コンパイラを呼び出す PLI コマンドの使用』を参照してください)。

この環境変数を使用して適用する変更内容とともにデフォルトが、新規のデフォルトになります。IBM. OPTIONS 環境変数、PLI コマンド、またはソース・プログラム内に PP(MACRO) オプションで指定したオプションがある場合、それらのデフォルトは、オーバーライドされます。

IBM.PPSQL

IBM.PPSQL 環境変数には、SQL プリプロセッサ・オプション設定を指定します。次に例を示します。

```
set ibm.ppsql=dbname(employee)
```

コンパイル時環境変数の設定

IBM.PPSQL 環境変数への文字ストリングの代入構文は、PLI コマンドで指定するコンパイル時オプションに要求されるものと同じです (33 ページの『コンパイラーを呼び出す PLI コマンドの使用』を参照してください)。

この環境変数を使用して適用する変更内容とともにデフォルトが、新規のデフォルトになります。IBM. OPTIONS 環境変数、PLI コマンド、またはソース・プログラム内に PP(SQL) オプションで指定したオプションがある場合、それらのデフォルトは、オーバーライドされます。

IBM.PPCICS

IBM.PPCICS 環境変数には、CICS プリプロセッサ・オプション設定を指定します。次に例を示します。

```
set ibm.ppcics=source edf
```

IBM.PPCICS 環境変数への文字ストリングの代入構文は、PLI コマンドで指定するコンパイル時オプションに要求されるものと同じです (33 ページの『コンパイラーを呼び出す PLI コマンドの使用』を参照してください)。

この環境変数を使用して適用する変更内容とともにデフォルトが、新規のデフォルトになります。IBM. OPTIONS 環境変数、PLI コマンド、またはソース・プログラム内に PP(CICS) オプションで指定したオプションがある場合、それらのデフォルトは、オーバーライドされます。

IBM.SOURCE

IBM.SOURCE 環境変数には、ソース・プログラム・ファイルに対するパスを指定します。次に例を示します。

```
set ibm.source=c:¥pli¥project¥updates;¥pli¥system
```

IBM.SYSLIB

IBM.SYSLIB 環境変数には、ソース・プログラム内の %INCLUDE ステートメントで指定されているインクルード・ファイルに対する 1 次入力ディレクトリー検索パスを指定します。次に例を示します。

```
set ibm.syslib=c:¥pli¥project¥updates;¥pli¥system
```

上記のディレクトリーは、INCLUDE 環境変数に指定されているディレクトリーの前に検索されます。

IBM.PRINT

IBM.PRINT 環境変数には、リスト・ファイルを書き込むパスを指定します。次に例を示します。

```
set ibm.print=c:¥pli¥project¥updates
```

リスト・ファイルは、ソース・プログラム・ファイルと同名ですが、拡張子は、アセンブラー・リストの場合 ASM、他のリスト情報の場合 LST となります。

デフォルトでは、診断メッセージと戻りコードは、画面上に表示されます。

IBM.OBJECT

IBM.OBJECT 環境変数には、ソース・プログラム・ファイルと同名で、拡張子がそれぞれ OBJ および DEF となるオブジェクト・ファイルと定義ファイル用の出力ディレクトリーを指定します。次に例を示します。

```
set ibm.object=c:¥pli¥project¥updates
```

オブジェクト・ファイルには、PL/I ソース・ステートメントから変換されたマシン・コードが格納されます。オブジェクト・ファイルを実行可能にするには、プログラムを構成する残りの OBJ ファイルのすべてと適切なライブラリー・ファイルをリンクする必要があります。プログラムのリンク方法の要点については、149 ページの『第 9 章 プログラムのリンク』を参照してください。

IBM.DECK

IBM.DECK 環境変数には、マクロ機能の生成する変更後のソース・ファイル用の出力ディレクトリーを指定します。このファイルは、MDECK コンパイル時オプションが有効な場合に限り生成されます。次に例を示します。

```
set ibm.deck=c:¥pli¥project¥updates
```

出力ファイルは、1 次ソース・プログラム・ファイルと同名ですが、拡張子は DEK となります。このファイルは、後で行うコンパイルへの入力として使用できます。

INCLUDE

INCLUDE 環境変数には、ソース・プログラム内の %INCLUDE ステートメントで指定されているインクルード・ファイルに対する 2 次入力ディレクトリー検索パスを指定します。次に例を示します。

```
set include=c:¥pli¥program
```

上記のディレクトリーは、IBM.SYSLIB 環境変数に指定されているディレクトリーの後に検索されます。

TMP

TMP 環境変数には、コンパイラが必要とする一時的な作業ファイル用の入力・出力ディレクトリーを指定します。次に例を示します。

```
set tmp=c:¥pli¥project¥updates
```

ローカル・エリア・ネットワーク (LAN) 上に常駐するディレクトリーを指定してはいけません。大規模なプログラムを扱う作業をしている場合は、この変数には、十分なフリー・スペースを持つディレクトリーを設定します。

コンパイラを呼び出す PLI コマンドの使用

PLI コマンドを使用して、コンパイラを呼び出します。PLI コマンドは、コマンド行または CMD ファイルで入力できます。

pli_program_file_specification

1 次ソース・プログラム用の Windows ファイルの指定。ファイル指定から拡張

コンパイラーを呼び出す PLI コマンドの使用

子を省略すると、コンパイラーは拡張子 `PLI` を想定します。パスを完全に省略すると、`IBM.SOURCE` を使用して別のディレクトリーを指定していない限り、現行ディレクトリーが仮定されます。

compiler_option

37 ページの『第 6 章 コンパイル時オプションの説明』に説明されている 1 つ以上のコンパイル時オプション。

以下は、`PLI` コマンドの例です。

```
pli hello (source
```

応答ファイルを使用して、共通オプションを 1 つのファイルに収納できます。そして、作成したファイルを使用して、さまざまなプログラムをコンパイルすることができます。例えば、ファイル `pli.opt` が、一連のオプションを格納している場合は、以下のように、`towers` サンプル・プログラムをコンパイルできます。

```
pli towers.pli ( @pli.opt
```

応答ファイルの使用時には、以下のガイドラインに注意します。

- ・ ソース・ファイルの名前およびオプションを、応答ファイルの名前の前に置くことができますが、応答ファイルの後には、何も続けないようにします。
- ・ 応答ファイルは、別の応答ファイルを指すことができます。

コンパイル時オプションを指定する場所

以下のセクションで説明するように、コンパイル時オプションは、3 箇所に指定できます。デフォルトをベースとして、連続した各場所で指定されたオプションは、その直前の場所で指定されたオプションをオーバーライドします。

注: `PL/I` が、コンパイルに使用するコンパイル時オプション設定を決定した後も、プログラム内の個々のソース・ステートメントが、さまざまなコンパイル時オプションの効果を変更する場合があります。例えば、プログラムで `OPTION(BYVALUE)` を指定すると、`DEFAULT(BYVALUE)` コンパイル時オプションがオーバーライドされます。

IBM.OPTIONS と IBM.PPxxx 環境変数

オプションを指定する第一の方法は、コンパイル時オプションに対して `IBM.OPTIONS` 環境変数およびプリプロセッサ・オプションに対して `IBM.PPxxx` 環境変数を設定することです。30 ページの『コンパイル時環境変数の設定』を参照してください。それらの環境変数でコンパイル時オプションを制御すると、通常のオプション・デフォルト値がオーバーライドされます。

PLI コマンド

コンパイル時オプションを指定する第二の方法 (オプションのデフォルト値、`IBM.OPTIONS`、および `IBM.PPxxx` をオーバーライドする方法) は、コンパイラーを呼び出すときに `PLI` コマンドを使用することです (33 ページの『コンパイラーを呼び出す `PLI` コマンドの使用』を参照してください)。これらのオプションは、現在のコンパイルのみに適用されます。

%PROCESS ステートメント

コンパイル時オプションを指定する第三かつ最後の方法 (オプションのデフォルト値、IBM.OPTIONS、IBM.PPxxx、および PLI コマンドをオーバーライドする方法) は、PL/I ソース・プログラム内で %PROCESS (または *PROCESS) ステートメントを使用することです。これらのオプションは、現在のコンパイルのみに適用されます。

以下の例は、%PROCESS ステートメントの使用を具体的に示します。

```
%process source margins(1,80);
Hello: proc options(main);
        display('Hello!');
end Hello;
```

1 つ以上の %PROCESS ステートメントを指定できますが、ブランク行も含めて、他のすべての PL/I ソース・ステートメントの前に来る必要があります。

PROCESS ステートメントのパーセント記号 (またはアスタリスク) は、ソース・ファイルの第 1 列にコーディングする必要があります。キーワード PROCESS は、次の列か任意の数のブランクの後に置くことができます。%PROCESS ステートメントのコンパイル時オプションのリストは、デフォルトの右側マージンを越えてはいけません。%PROCESS ステートメントは、次行に続けることができますが、続ける場合には、キーワードまたは値を 2 行に分割しないように気をつけます。ステートメントを折り返して 2 行にするよりも、1 行に 1 ステートメントとして、複数 %PROCESS ステートメントをコーディングすることを推奨します。

すべての %PROCESS ステートメントの解釈が終了すると、プログラムの残りの部分が、PLI コマンドと %PROCESS ステートメントの考慮後に決定されたマージン設定を使用して読み取られます。これは、前に示したサンプル %PROCESS ステートメントが、コンパイル時にデフォルトの MARGINS(2,72) が有効であるという前提のもとに、正しく処理されることを意味します。

コンパイル時オプションを指定する場所

第 6 章 コンパイル時オプションの説明

この章では、省略形、デフォルト、および関連のコーディング・サンプルと合わせて、コンパイル時オプションを詳細に説明します。

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 2 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 2 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 139 ページの『コンパイラー・リストの使用』にあります。

表 2. 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	Windows デフォルト
ADDEXT NOADDEXT	-	ADDEXT
AGGREGATE (DECIMAL HEXADEC) NOAGGREGATE	AG NAG	NOAGGREGATE
ATTRIBUTES[(FULL SHORT)] NOATTRIBUTES	A NA	NA [(FULL)] ¹
BIFPREC(15 31)	-	BIFPREC(31)
BLANK('c')	-	BLANK('t') ²
CHECK(STORAGE NOSTORAGE, CONFORMANCE NOCONFORMANCE)	-	CHECK(NSTG, NOCONFORMANCE)
CMPAT(LE V1 V2)	-	CMPAT(V2)
CODEPAGE(n)	CP	CODEPAGE(00819)
COMPILE NOCOMPILE[(W E S)]	C NC	NOCOMPILE(S)
COPYRIGHT('string') NOCOPYRIGHT	-	NOCOPYRIGHT
CURRENCY('c')	CURR	CURRENCY(\$)

コンパイル時オプション

表 2. 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	Windows デフォルト
NODBCS DBCS(JPN CHS CHT KOR)	-	NODBCS
DEFAULT(<i>attribute</i> <i>option</i>)	DFT	46 ページを参照。
DLLINIT NODLLINIT	-	NODLLINIT
EXIT NOEXIT	-	NOEXIT
EXTRN(FULL SHORT)	-	EXTRN(FULL)
FLAG[(I W E S)]	F	FLAG(W)
FLOATINMATH(ASIS LONG EXTENDED)	-	FLOATINMATH(ASIS)
GONUMBER NOGONUMBER	GN NGN	NOGONUMBER
GRAPHIC NOGRAPHIC	GR NGR	NOGRAPHIC
IMPRECISE NOIMPRECISE	-	IMPRECISE
INCAFTER([PROCESS(<i>filename</i>)])	-	INCAFTER()
INCDIR(<i>directory name</i>)	-	INCDIR()
INCLUDE[(EXT('include extension'))]	INC	INC(EXT('inc'))
INITAUTO NOINITAUTO	-	NOINITAUTO
INITBASED NOINITBASED	-	NOINITBASED
INITCTL NOINITCTL	-	NOINITCTL
INITSTATIC NOINITSTATIC	-	NOINITSTATIC
INSOURCE[(FULL SHORT)] NOINSOURCE	IS NIS	NOINSOURCE
LANGVL(SAA SAA2[,NOEXT OS])	-	LANGVL(SAA2,OS)
LIBS	-	61 ページを参照。
LIMITS(<i>options</i>)	-	62 ページを参照。
LINECOUNT(<i>n</i>)	LC	LINECOUNT(60)
LINEDIR NOLINEDIR	-	NOLINEDIR
LIST NOLIST	-	NOLIST
LISTVIEW(SOURCE AFTERMACRO AFTERCICS AFTERSQL AFTERALL)	-	LISTVIEW(SOURCE)
MACRO NOMACRO	M NM	NOMACRO
MARGIN('c') NOMARGIN	MI NMI	NOMARGIN
MARGINS(m,n[,c]) NOMARGINS	MAR(m,n)	MARGINS F-format: (2,72) V-format: (10,100)
MAXGEN(n)	-	MAXGEN(100000)
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(1000)
MDECK NOMDECK	MD NMD	NOMDECK
MSG(390 *)	-	MSG(*)
NAMES('lower'[,upper])	-	NAMES('#@\$', '#@\$')
NATLANG(ENU CHS CHT DEU ESP FRA JPN PTB)	-	NATLANG(ENU)
NEST NONEST	-	NONEST
NOT	-	NOT('~')
NUMBER NONUMBER	NUM NNUM	NUMBER
OBJECT NOOBJECT	OBJ NOBJ	OBJECT
OFFSET NOOFFSET	OF NOF	NOOFFSET
ONSNAP(STRINRANGE STRINGSIZE) NOONSNAP	-	NOONSNAP
OPTIMIZE(0 2 3) NOOPTIMIZE	OPT NOPT	OPT(0)
OPTIONS[(ALL DOC)] NOOPTIONS	OP NOP	NOOPTIONS
OR('c')	-	OR(' ')

表 2. 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	Windows デフォルト
PP(pp-name) NOPP	-	NOPP
PPCICS('string') NOPPCICS	-	NOPPCICS
PPINCLUDE('string') NOPPINCLUDE	-	NOPPINCLUDE
PPMACRO('string') NOPPMACRO	-	NOPPMACRO
PPSQL('string') NOPPSQL	-	NOPPSQL
PPTRACE NOPTRACE	-	NOPTRACE
PRECTYPE (ANS DECDIGIT DECRERESULT)	-	PRECTYPE(ANS)
PREFIX(condition)	-	79 ページを参照。
PROBE NOPROBE	-	PROBE
PROCEED NOPROCEED[(W E S)]	PRO NPRO	NOPROCEED(S)
PROCESS[(KEEP DELETE)] NOPROCESS	-	PROCESS(DELETE)
QUOTE("")	-	QUOTE("")
REDUCE NOREDUCE	-	REDUCE
RESEXP NORESEXP	-	RESEXP
RESPECT([DATE])	-	RESPECT()
RULES(options)	-	83 ページを参照。
SEMANTIC NOSEMANTIC[(W E S)]	SEM NSEM	NOSEMANTIC(S)
SNAP NOSNAP	-	NOSNAP
SOSI NOSOSI	-	NOSOSI
SOURCE NOSOURCE	S NS	NOSOURCE
STATIC(FULL SHORT)	-	STATIC(SHORT)
STMT NOSTMT	-	NOSTMT
STORAGE NOSTORAGE	STG NSTG	NOSTORAGE
SYNTAX NOSYNTAX[(W E S)]	SYN NSYN	NOSYNTAX(S)
SYSPARM('string')	-	SYSPARM("")
SYSTEM(WINDOWS CICS IMS PENTIUM S486)	-	SYSTEM(WINDOWS)
TERMINAL NOTERMINAL	TERM NTERM	TERMINAL
TEST(ALL NONE STMT,SYM ,NOSYM) NOTEST	-	NOTEST(ALL,SYM) ³
USAGE(options)	-	95 ページを参照。
WIDECHAR(BIGENDIAN LITTLEENDIAN)	WCHAR	WIDECHAR(BIGENDIAN)
WINDOW(w)	-	WINDOW(1950)
XINFO(options)	-	XINFO(NODEF,NOMSG, NOSYMNOSYN,NOXMI, 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。

コンパイル時オプションの使用規則

1. 相互排他的コンパイル時オプションまたはサブオプションを指定すると、最後に指定したオプションが有効になります。
2. 必須のストリングが PL/I ID 規則に従っている場合は、引用符で囲む必要はありません。コンパイラーは、それらのストリングを大文字に変換します。

以下のオプションは、指定するストリングが特殊文字またはランタイム・オプションを含むため、ストリングを指定する場合は、引用符で囲む方が賢明です。

コンパイル時オプション

- CURRENCY
 - DEFAULT(INITFILL)
 - MARGINI
 - NAMES
 - NOT
 - OR
3. オプションのストリングを引用符で囲む場合、ストリング自体には、引用符を含めることはできません。
 4. オプションのストリングを引用符で囲む場合、ストリングは、16 進ストリングとして、例えば、NOT('aa'x) のように指定することができます。
 5. コンパイル時オプションの指定が正しくなかった場合 (例えば、NEST ではなく、NEXT を指定した場合など)、OPTIONS コンパイル時オプションは、自動的に OPTIONS に設定されます。これにより、コンパイル作業に有効なすべてのコンパイル時オプションのリストが表示されます。

ADDEXT

このオプションでは、ファイル拡張子を持たないソース・ファイル名およびインクルード・ファイル名に、コンパイラーが、ファイル拡張子 (.pli 、 .cpy) を追加するかどうかを指定します。



NOADDEXT

コンパイラーは、ファイル拡張子を追加しません。リモートの編集/コンパイル環境内からワークステーション上で、チェックアウト・コンパイルを行う場合、マッピングが行われないため、NOADDEXT を指定する必要があります。

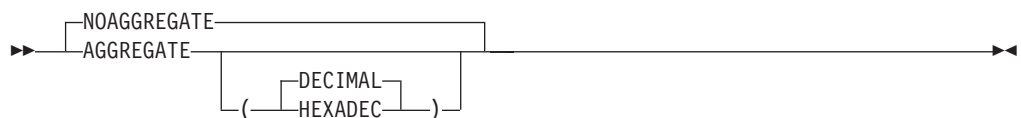
ADDEXT

コンパイラーは、ファイル拡張子を追加します。ADDEXT を指定すると、コンパイラーは、コマンド『pli hello』を『pli hello.pli』と解釈します。

ADDEXT と NOADDEXT は、コンパイラーがファイル検索を行うときに、ファイル名にファイル拡張子を追加するかどうかのみに影響します。コンパイラー出力ファイルは、このオプションの設定内容に関係なく、ファイル拡張子を持ちます。

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: NAG、AG

AGGREGATE オプションのサブオプションは、サブ要素のオフセットが集合長さテーブルに表示される方法を決定します。

DECIMAL

AGGREGATE リスト内のすべてのオフセットが 10 進数で表示されます。

HEXADEC

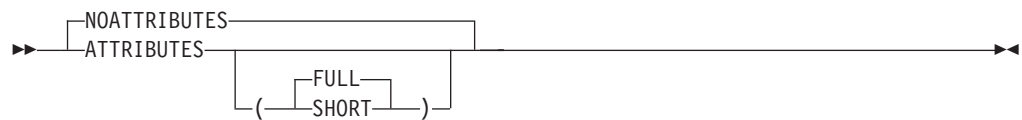
AGGREGATE リスト内のすべてのオフセットが 16 進数で表示されます。

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。

139 ページの『コンパイラー・リストの使用』に、サンプル・リストを示します。

ATTRIBUTES

このオプションを指定すると、ソース・プログラム ID とその属性のテーブルが、コンパイラー・リストに含まれます。



省略形: NA、A

FULL

すべての ID と属性をリストに含めます。ATTRIBUTES(FULL) を選択した場合のテーブルの例については、139 ページの『コンパイラー・リストの使用』を参照してください。

SHORT

未参照の ID を省略します。

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。



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

コンパイル時オプション

- 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。

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—(—'——'—)——▶▶

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、'09'X です。

char

単一の SBCS 文字。

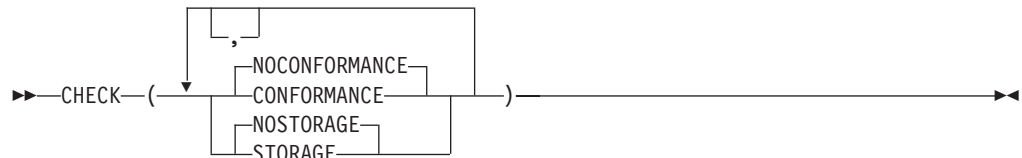
英字、数字、および「PL/I 言語解説書」に定義されている特殊文字はどれも指定できません。

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。

デフォルト: BLANK('09'x)

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、コンパイラーは以下の状況下で、プロシージャ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に一致するかどうかを実行時に検査するコードを生成します。

- 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) の場合、渡された引数の長さが一致しないと、STRINGSIZE 条件が引き起こされます。
- パラメーターがストリング (またはストリングの配列) の場合、引数の長さタイプ (VARYING、NONVARYING、または VARYINGZ) が同じでないと、STRINGSIZE 条件が引き起こされます。
- パラメーターが (スカラーまたは構造体の) 配列の場合、定数の境界が、渡された引数の境界に一致しないと、SUBSCRIPTRANGE 条件が引き起こされます。また、すべてのエクステントが一定で、引数内の配列エレメントのサイズおよび余白がパラメーター内の配列のサイズおよび余白に一致しない場合でも、SUBSCRIPTRANGE 条件が引き起こされます。構造体の内部にある配列は検査されません。
- パラメーターが、一定のエクステントを持つ構造体または共用体の場合、最後のエレメントのオフセットが、渡された引数のオフセットに一致しないと、SUBSCRIPTRANGE 条件が引き起こされます。
- プロシージャに RETURNS BYADDR 属性があり、その属性がストリング型を指定している場合、RETURNS 値に対して渡されたストリングの長さが一致しないと、STRINGSIZE 条件が引き起こされます。

このエクストラ・コードは、NODESCRIPTOR オプションがプロシージャに適用される場合、またはブロックに ENTRY ステートメントが含まれる場合、または CMPAT(LE) オプションが有効になっている場合は生成されません。

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・ルーチンを呼び出します

コンパイル時オプション

(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

CMPAT

CMPAT オプションは、コンパイラーが生成する記述子に使用されるフォーマットを指定します。

▶▶CMPAT—(

LE
V2
V1

)————▶▶

LE

CMPAT(LE) の場合、コンパイラーは、言語処理環境製品で定義されているフォーマットで記述子を生成します。

V1 CMPAT(V1) の場合、コンパイラーは、OS PL/I バージョン 1 コンパイラーによる生成フォーマットで記述子を生成します。

V2 CMPAT(V2) の場合、コンパイラーは、CMPAT(V2) オプション指定時の OS PL/I バージョン 2 コンパイラーの生成フォーマットで記述子を生成します。

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。

DFT(DECLIST) オプションは CMPAT(V1) または CMPAT(V2) オプションと対立するので、CMPAT(V1) または CMPAT(V2) オプションと一緒に指定するとメッセージが出され、DFT(DESCLOCATOR) オプションがとられます。

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*)————▶▶

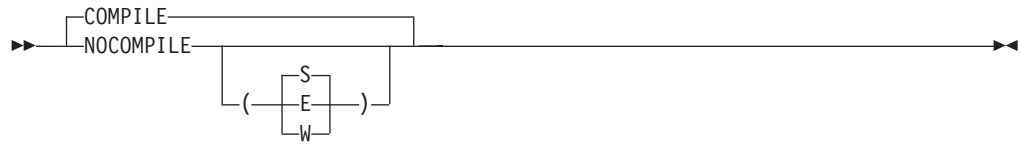
サポートされる CCSID は、次のとおりです。

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

デフォルト CCSID 00819 は、Latin 1 ASCII コード・ページです。

COMPILE

このオプションを指定すると、コード生成段階の実行は、この処理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: NC、C

NOCOMPILER

コンパイルは、セマンティック検査後に無条件に停止されます。

NOCOMPILER(S)

コンパイルは、重大エラーまたは回復不能エラーが検出された場合に停止されます。

NOCOMPILER(E)

コンパイルは、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に停止されます。

NOCOMPILER(W)

コンパイルは、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に停止されます。

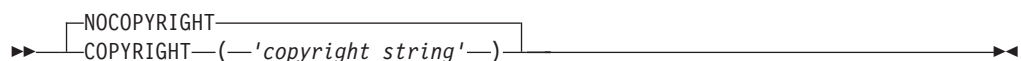
COMPILE

NOCOMPILER(S) と同等です。

コンパイルが、NOCOMPILER オプションによって強制終了された場合に、リストが生成されるかどうかは、コンパイルが停止した時点によって決まります。例えば、相互参照や属性のリストは、NOCOMPILER オプションでも普通生成されますが、セマンティック検査時にエラーが発生すると、生成されない可能性があります。

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。

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。

CURRENCY

このオプションは、ドル記号に固有の文字を使用するために使用できます。

►►—CURRENCY—(—x—)—————◄◄

- 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。

デフォルト: CURRENCY('\$')

DBCS

DBCS オプションは、存在するどのバイトを DBCS 形式として受け入れるかを決定します。

►►—

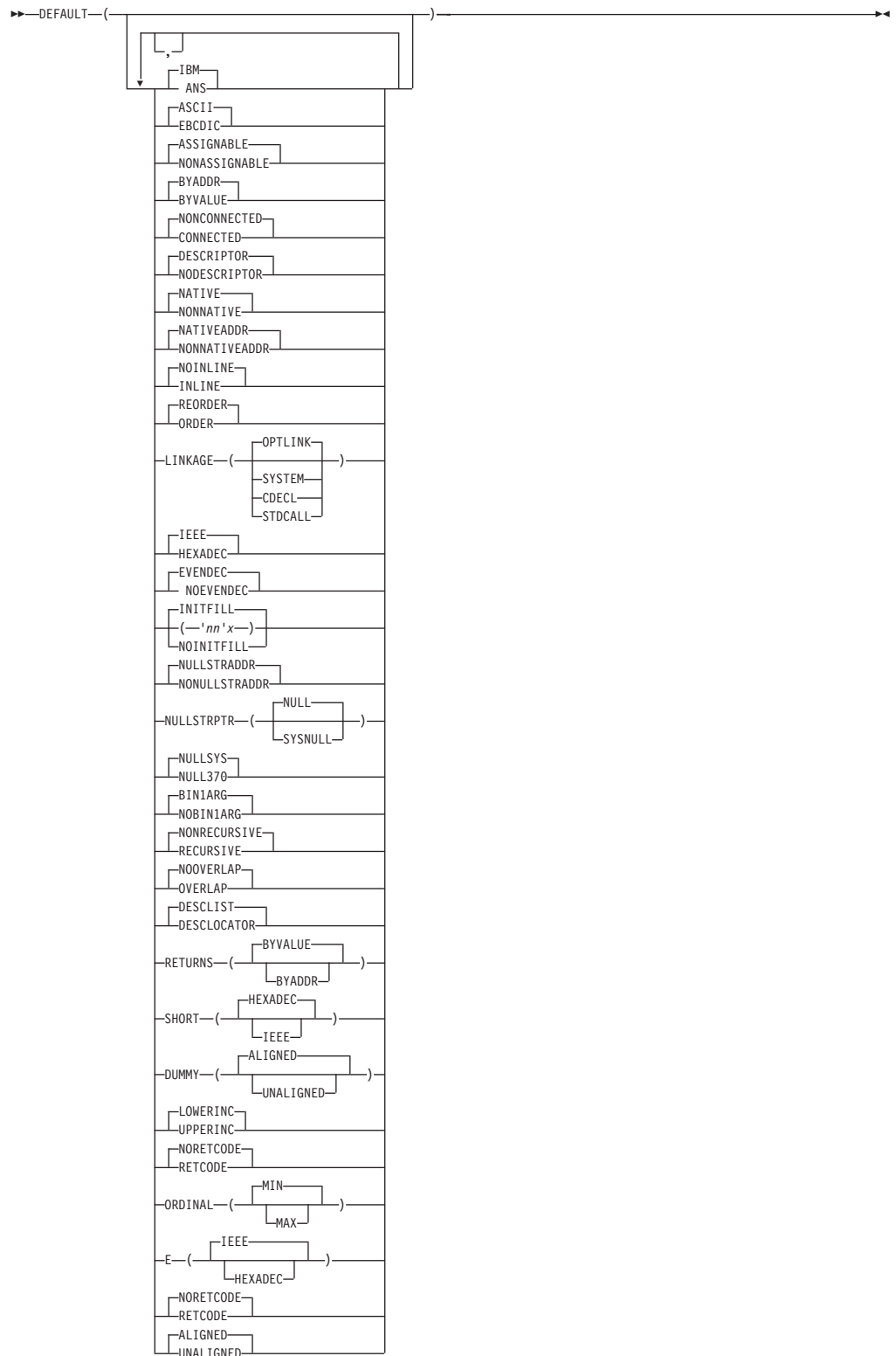
NODBCS
DBCS

—————◄◄

DBCS オプションを指定したときに、GRAPHIC オプションも指定した場合は、ID、リテラルなどにおいて、指示された言語から適切な DBCS 文字が受け入れられます。

DEFAULT

このオプションは、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、CONN、NONCONN

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。

ASCII または 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。

EBCDIC は、EBCDIC 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順序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 is the same as '313133'X
'123'E is the same as 'F1F1F3'X
```

ASSIGNABLE または NONASSIGNABLE

このオプションは、静的変数のみに適用されます。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立てます。メインフレームにコーディングを移植する場合 (プログラムが再入可能な場合)、このオプションにより、そうしない場合に記憶保護例外を起こす可能性のあるステートメントにフラグを立てます。

BYADDR または 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。

CONNECTED または 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。

DESCRIPTOR または 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、より効率の高いコーディングが生成されますが、以下の条件のもとでエラーが発生します。

- 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、または領域のサイズ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。

NATIVE または 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。

NATIVEADDR または 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。

INLINE または 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アプリケーションのパフォーマンスの改善方法の詳細については、341 ページの『第 20 章 パフォーマンスの向上』を参照してください。

REORDER または ORDER

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・コードの最適化が可能になります。341 ページの『第 20 章 パフォーマンスの向上』を参照してください。

ORDINAL (MAX または MIN)

ORDINAL(MAX) を指定すると、その定義が PRECISION 属性を含まない順次数すべてに、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。

OVERLAP または 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。341 ページの『第 20 章 パフォーマンスの向上』。

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

PL/I for Windows 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

パラメーターがすべてスタックに引き渡されますが、呼び出し側 関数により、スタックはクリーンアップされます。

STDCALL

Windows API の標準リンケージ規則。このリンケージ規則は、Windows 下で使用され、スタックにすべてのパラメーターを引き渡します。呼び出された側の 関数が、スタックをクリーンアップします。

CDECL

パラメーターがすべてスタックに引き渡されますが、呼び出し側 関数により、スタックはクリーンアップされます。外部名は、接頭部として _
applied を持ちます。

OPTIONS(COBOL) は、リンケージがエントリーの DCL ステートメントまたは PROC ステートメントで指定されていない場合、LINKAGE(SYSTEM) を暗黙指定します。

リンケージ規則の詳細については、387 ページの『第 24 章 呼び出し規則』を参照してください。

IEEE または HEXADEC

IEEE を指定すると、浮動小数点データは、ネイティブ IEEE フォーマットで記憶装置に格納されます。HEXADEC は、浮動小数点データの保管がメインフレーム環境と同じにすることを指示します。

EVENDEC または 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ー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 を指定すると、(メインフレーム PL/I を使用している場合とまったく同様に) SIZE 条件は発生しません。

EVENDEC がデフォルトです。

BIN1ARG または NOBIN1ARG

このサブオプションは、プロトタイプ化されていない関数に渡される 1 バイトの REAL FIXED BIN 引数をコンパイラーが処理する方法を制御します。

BIN1ARG が指定された場合、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 引数をそのまま渡します。

ただし、NOBIN1ARG が指定された場合、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る 1 バイトの REAL FIXED BIN 引数を、2 バイトの一時的な FIXED BIN 引数に割り当てて、その一時的な引数を代わりに渡します。

次の例を見てください。

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

call f1( 1b );
call f2( 1b );
```

DEFAULT(BIN1ARG) が指定された場合、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。ただし、DEFAULT(NOBIN1ARG) が指定された場合、コンパイラーは 2 バイトの FIXED BIN(15) 引数のアドレスを両方のルーチンに渡します。

ルーチン f1 が COBOL ルーチンの場合、1 バイトの整数は COBOL でサポートされていないため、1 バイトの整数の引数をこのルーチンに渡すと問題が発生することに注意してください。この場合、DEFAULT(NOBIN1ARG) を使用すると状況が改善される可能性があります。ただし、エントリーの宣言で引数の属性を指定する方がより望ましい結果となります。

BIN1ARG がデフォルトです。

INITFILL または 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。

16 進数値 (nn) とともに INITFILL を指定すると、16 進数値は複製され、すべての自動変数の記憶域がその値で充てんされます。16 進値を入力しない場合、デフォルトは '00'x です。NOINITFILL では、自動変数の初期化は行われません。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ため、実動プログラム内では指定しないでください。ただし、プログラム開発時には、未初期化の自動変数の検出に役立ちます。

NOINITFILL がデフォルトです。

LOWERINC または UPPERINC

LOWERINC を指定すると、コンパイラーは、INCLUDE ファイルとして、小文字のファイル名を受け入れます。UPPERINC を指定すると、コンパイラーは、INCLUDE ファイルとして、大文字のファイル名を受け入れます。

LOWERINC がデフォルトです。

NULLSTRADDR または NONNULLSTRADDR

このサブオプションは、ヌル・ストリングが引数として渡されるときに、コンパイラーがそのヌル・ストリングを処理する方法を制御します。

NULLSTRADDR を指定した場合、入りの呼び出し時に引数としてヌル・ストリングが指定されると、コンパイラーによって、自動ストレージの初期化部分のアドレスが渡されます。これは、OS PL/I コンパイラーおよび PL/I for MVS コンパイラーの動作と互換性があります。

ただし、NONNULLSTRADDR を指定した場合、入りの呼び出し時に引数としてヌル・ストリングが指定されると、コンパイラーによって、NULL ポインターが引数のアドレスとして渡されます。これは、Enterprise PL/I コンパイラーの初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。

NULLSTRPTR

このサブオプションは、ヌル・ストリングが POINTER に割り当てられるときに、コンパイラーがそのヌル・ストリングを処理する方法を制御します。

NULLSTRPTR(SYSNULL) を指定した場合、” を POINTER に割り当てると、結果は、SYSNULL() をそのポインターに割り当てた場合と同じになります。

NULLSTRPTR(NULL) を指定した場合、” を POINTER に割り当てると、結果は、NULL() をそのポインターに割り当てた場合と同じになります。

NULLSTRPTR(NULL) がデフォルトです。

NULLSYS または 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。メインフレーム PL/I の場合と同じく binvalue(null()) を 'ff_00_00_00'xn に等しくする場合は、NULL370 を指定します。

NULLSYS がデフォルトです。

RECURSIVE または 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 または DESCLOCATOR

DEFAULT(DECLIST) を指定すると、コンパイラーは、以前のワークステーション製品リリースとまったく同じようにして、コーディングを生成します (すべての記述子が、「隠れた」最後のパラメーターとしてリストで引き渡されます)。

DEFAULT(DESCLOCATOR) を指定すると、記述子を必要とするパラメーターは、メインフレーム PL/I と同じ方法で、ロケーターまたは記述子を使用して引き渡されます。この方法によって、古いコーディングが、ポインターの受け取りを期待しているルーチン間で構造体を引き渡す場合でも、機能することが可能になります。

DECLIST がデフォルトです。

RETURNS (BYVALUE または 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。

RETURNS(BYVALUE) がデフォルトです。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ー・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。

SHORT (HEXADEC または IEEE)

このサブオプションは、他の UNIX PL/I コンパイラーとの互換性を向上させます。SHORT (HEXADEC) を指定すると、 $p \leq 21$ の場合に、FLOAT BIN (p) が短い (4 バイトの) 浮動小数点数にマップされます。SHORT (IEEE) を指定すると、 $p \leq 24$ の場合に、FLOAT BIN (p) は、短い (4 バイト) 浮動小数点数にマップされます。

SHORT (HEXADEC) がデフォルトです。

DUMMY (ALIGNED または 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。

次の例を見てください。

```
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。

RETCODE または NORETCODE

RETCODE を指定すると、RETURNS 属性を持たない外部プロシージャーに対しても、整数値を戻すように指示できます。外部プロシージャーは、戻る直前に PLIRETV 組み込み関数を呼び出して取得した整数値を戻します。この指定により、外部プロシージャーを、メインフレーム上で COBOL から呼び出される似たようなプロシージャーのように動作させることができます。

NORETCODE を指定すると、RETURNS 属性を持たないプロシージャーから、特別なコードは生成されなくなります。

ALIGNED または 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。

E(IEEE または HEXADEC)

E サブオプションは、E 形式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 形式項目の指数として 4 桁の数字が使用されます。

E(HEXADEC) を指定すると、E 形式項目の指数として 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。

DFT(E(HEXADEC)) は、ワークステーション上で 390 アプリケーションの開発とテストを行う場合に役立ちます。ステートメント "put skip edit(x) (e(15,8));" は、390 上では何のメッセージも生成しませんが、デフォルトでは、Intel と AIX のもとで、フラグが立てられます。DFT(E(HEXADEC)) を指定すると、この状態が修正されます。

IEEE がデフォルトです。

DEFAULT (IBM ASCII ASSIGNABLE BYADDR NONCONNECTED DESCRIPTOR
NATIVE NATIVEADDR NOINLINE REORDER LINKAGE(OPTLINK) IEEE
EVENDEC BIN1ARG NOINITFILL ORDINAL(MIN) NOOVERLAP NULLSTRADDR
NULLSTRPTR(NULL) NULLSYS NONRECURSIVE DESCLIST
RETURNS(BYVALUE) SHORT(HEXADEC) DUMMY(ALIGNED) LOWERINC
NORETCODE ALIGNED E(IEEE)).

DLLINIT

このオプションは、結果のオブジェクト・ファイルを実行可能ファイル (.EXE) またはダイナミック・リンク・ライブラリー・ファイル (.DLL) のいずれかで使用するかを指定する場合に使用します。



NODLLINIT

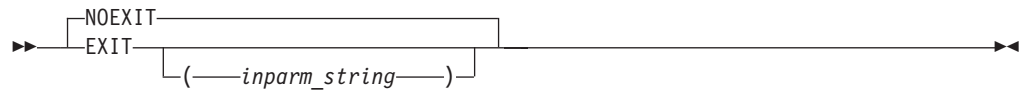
このオプションは、.EXE ファイルを作成するために実行する毎回のコンパイル時に有効にする必要があります。

DLLINIT

このオプションは、コンパイルで作成されるオブジェクト・ファイルを .DLL の作成に使用する場合に、実行するコンパイル作業の中で、少なくとも 1 回指定する必要があります。

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。

詳しくは、357 ページの『第 21 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。

SHORT

EXTRN は、参照された定数に対してだけ発行されます。これはデフォルトです。

FLAG

FLAG オプションは、その重大度を超えるとコンパイラー・リストにメッセージを含め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。

コンパイル時オプション

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口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。

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。

GONUMBER

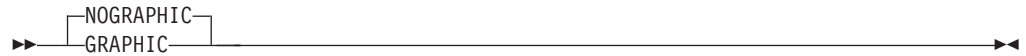
このオプションにより、オブジェクト・ファイルの一部として、ステートメント番号テーブルが作成されます。このテーブルは、デバッグ時に有効となります。



省略形: NGN、GN

GRAPHIC

このオプションは、ソース・プログラム内に 2 バイト文字が存在することを指定します。



省略形: NGR、GR

ソース・プログラムに以下のいずれかを使用する場合に、GRAPHIC を指定する必要があります。

- DBCS ID
- コメント内の DBCS
- 漢字ストリング定数
- 混合ストリング定数

IMPRECISE

このオプションによって、浮動小数点の演算結果の精度、および浮動小数点割り込みが報告される位置が決定されます。



省略形: IMP、NIMP

IMPRECISE

浮動小数点の演算結果の精度が、IEEE に準拠していない可能性があり、浮動小数点割り込みの位置も正確でない可能性があります。精度の損失は、ほとんどのアプリケーションにとって無視できるものです。割り込み位置は、割り込み点に近い場合も、遠い場合も、場合によっては別のブロックに存在することもあります。

このオプションを使用すると、より高速に実行される小型化されたオブジェクト・コードが生成されます。このオプションは、実動プログラムにお勧めします。

NOIMPRECISE

浮動小数点の演算結果の精度は、IEEE に準拠し、浮動小数点割り込みの正確な位置が要求されます。このオプションにより、実行速度の遅いコードが生成されるため、是非という場合には、プログラム開発時に限り使用することをお勧めします。

NOIMPRECISE の方が、IMPRECISE よりも優れた浮動小数点エラー検出を可能にするにもかかわらず、Windows オペレーティング・システムでは、浮動小数点例外の即時検出が許されていません。浮動小数点例外を発生させる可能性のあるステートメントがプログラム内にある場合は、そのステートメントを単独で BEGIN ブロック内に入れることにより、前記の検出の問題を回避できます。

➡➡INCAFTER—(PROCESS—(*filename*))—➡➡

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様です。

INCLUDE

```

graph LR
    INCLUDE["INCLUDE( "]
    subgraph List
        direction TB
        L1[" "]
        L2[" "]
        L3[" "]
        L1 --> L2
        L2 --> L3
        L3 --> L1
    end
    INCLUDE --> List
    List --> END[" ) "]
    style END fill:none,stroke:none

```

```
include ( ext(Cop Inc '2++' Mac) )
```

以下の例では、コンパイラーは、最初、ファイル名拡張子なしにインクルード・ファイルを検索した後、INC、CPY、および MAC というファイル名拡張子を持つインクルード・ファイルを検索します。

```
include (ext(' ',Inc,Cpy,Mac))
```

INITAUTO

INITAUTO では、コンパイラーは、INITIAL 属性を持たない AUTOMATIC 変数に INITIAL 属性を追加します。



コンパイラーは、変数のデータ属性により INITIAL 値を判別します。

- FIXED または FLOAT の場合は、INIT((*) 0)
- PICTURE、CHAR、BIT、GRAPHIC または WIDECHAR の場合は、INIT((*) ”)
- POINTER または OFFSET の場合は、INIT((*) sysnull())

NOINITAUTO がデフォルトです。

INITAUTO は、完全に初期化されていない (ただし DFT(INITFILL) オプションとは異なり、これらの変数には意味のある初期値が与えられます) AUTOMATIC 変数を含む各ブロックのプロローグでさらに多くのコードを生成するため、パフォーマンスに悪影響を与えます。

INITBASED

このオプションは、BASED 変数を除き、INITAUTO と同じ機能を実行します。



NOINITBASED がデフォルトです。

INITBASED は、完全に初期化されていない BASED 変数の ALLOCATE についてさらに多くのコードを生成するため、パフォーマンスに悪影響を与えます。

INITCTL

このオプションは、CONTROLLED 変数を除き、INITAUTO と同じ機能を実行します。



NOINITCTL がデフォルトです。

INITCTL は、完全に初期化されていない CONTROLLED 変数の ALLOCATE についてさらに多くのコードを生成するため、パフォーマンスに悪影響を与えます。

INITSTATIC

このオプションは、STATIC 変数を除き、INITAUTO と同じ機能を実行します。



NOINITSTATIC がデフォルトです。

INITSTATIC オプションでは、一部のオブジェクトが大きくなったり、一部のコンパイルでより多くの時間がかかったりすることがありますが、それ以外ではパフォーマンスに影響はありません。

INSOURCE

INSOURCE オプションは、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が組み込むことを指定します。



省略形: NIS、IS

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。

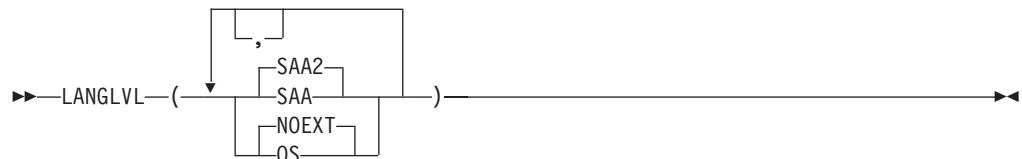
```
insource: proc options(main);
    %include member;
end;
```

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)。

LANGLVL

このオプションでは、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。コンパイラーは、指定された言語定義に対する違反すべてにフラグを立てます。



SAA

コンパイラーは、OS PL/I バージョン 2 リリース 3 のサポートしないキーワードと組み込み関数に対して、前者にはフラグを立て、後者は認識しません。

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

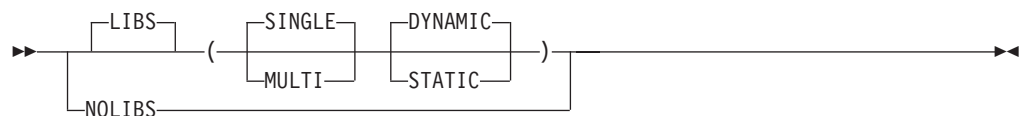
指定されている言語レベルを越えた拡張は許されていません。

Windows

Windows 環境に固有の ENVIRONMENT オプションは許されています。

LIBS

外部エントリやデータに対する参照を解決するためリンク時に検索されるデフォルト・ライブラリーの名前を記述する情報を、コンパイラーが、オブジェクト・ファイル内に生成するかどうかを、このオプションを使用して指定します。



LIBS

LIBS(SINGLE DYNAMIC) を指定する場合と同じです。

LIBS(SINGLE DYNAMIC)

指定すると、リンク時に検索されるデフォルト・ライブラリーが、シングルスレッドの PL/I ライブラリーとなります。

- Windows では、該当するライブラリーは、ibmws20i.lib、ibmwstbi.lib、hepws20i.lib、および kernel32.lib です。

LIBS(MULTI DYNAMIC)

指定すると、リンク時に検索されるデフォルト・ライブラリーが、マルチスレッドの PL/I ライブラリーとなります。

- Windows では、該当するライブラリーは、ibmwm20i.lib、ibmwmtbi.lib、hepwm20i.lib、および kernel32.lib です。

LIBS(SINGLE STATIC)

指定すると、リンク時に検索されるデフォルト・ライブラリーが、静的な非マルチスレッド・ライブラリーとなります。

- Windows では、該当するライブラリーは、ibmws20.lib、ibmws35.lib、ibmwstb.lib、hepws20.lib、および kernel32.lib です。

LIBS(MULTI STATIC)

指定すると、リンク時に検索されるデフォルト・ライブラリーが、静的なマルチスレッド・ライブラリーとなります。すなわち、ライブラリーが、ユーザー・モジュールに静的にリンクされることを意味します。

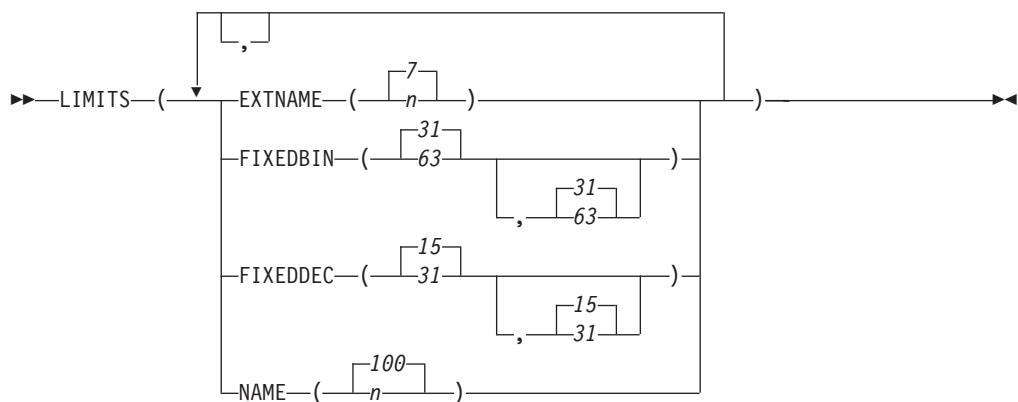
- Windows では、該当するライブラリーは、ibmwm20.lib、ibmwm35.lib、ibmwmtb.lib、hepwm20.lib、および kernel32.lib です。

SINGLE サブオプションは、アプリケーションが非マルチスレッド言語を使用する場合に限り指定し、MULTI サブオプションは、アプリケーションに、PL/I マルチスレッド言語が含まれている場合に限り指定します。

マルチスレッド言語が使用されていない場合も LIBS(MULTI) の使用は可能ですが、使用すると、LIBS(SINGLE) の場合よりも、アプリケーションの実行速度が遅くなります。

LIMITS

このオプションでは、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 を指定します。

FIXEDDEC(15,31) を指定した場合は、15 より大きい精度で FIXED DECIMAL 変数を宣言できますが、精度が 15 より大きいオペランドが式に含まれていない場合、最大精度として 15 を使用してすべての演算が行われます。

FIXEDDEC(15,31) は FIXEDDEC(31) よりはるかに優れたパフォーマンスを発揮します。

FIXEDDEC(15) と FIXEDDEC(15,15) は同等です。同様に、FIXEDDEC(31) と FIXEDDEC(31,31) も同等です。

FIXEDDEC(31,15) は許可されていません。

デフォルトは FIXEDDEC(15,15) です。

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、算術演算はすべて 4 バイト整数を使用して行われます。

FIXEDBIN(63,31) は指定できません。

デフォルトは FIXEDBIN(31,31) です。

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 7 です。

デフォルト: LIMITS(EXTNAME(100) FIXEDBIN(31,31) FIXEDDEC(15,15) NAME(100))

LINECOUNT

このオプションは、コンパイラ・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—(n)————▶▶

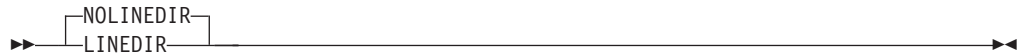
省略形: LC

n の値は、1 から 32,767 の範囲となります。

デフォルト: LINECOUNT(60)

LINEDIR

このオプションは、コンパイラーが `%LINE` ディレクティブを受け入れることを指定します。



`LINEDIR` オプションが指定された場合、コンパイラーはすべての `%INCLUDE` ステートメントをリジェクトします。

デフォルト: `NOLINEDIR`

LIST

このオプションを指定すると、オブジェクト・モジュール・リストが生成されます。このリストは、アセンブラー言語命令に類似した形式になります。

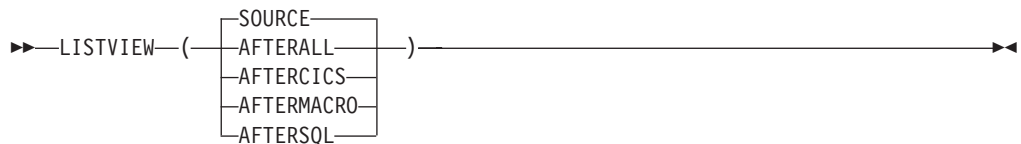


オブジェクト・リストは、拡張子 `.asm` の別個のファイルに生成されます。

アセンブラー・リストは、常にコンパイルされません。139 ページの『コンパイラー・リストの使用』に、サンプル・リストを示します。

LISTVIEW

`LISTVIEW` オプションは、コンパイラーが、ソースをソース・リストに表示するかどうか、またはソースがプリプロセッサの 1 つ以上によって処理された後に、そのソースを表示するかどうかを指定します。



SOURCE

ソース・リストによって純粋なソースが表示されます。また、おそらくより重要なこととして、`Debug Tool` によってこのソースがソース・ビューとして表示されます。

AFTERALL

ソース・リストによって、最後のプリプロセッサの最後の呼び出し (ある場合) での `MDECK` から得られたかのようにソースが表示されます。また、おそらくより重要なこととして、`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、`Debug Tool` によってこのソースがソース・ビューとして表示されます。

`AFTERALL` の省略形として、`AALL` を使用することができます。

AFTERCICS

ソース・リストによって、CICS プリプロセッサの最後の呼び出し (ある場合) での MDECK から得られたかのようにソースが表示されます。また、おそらくより重要なこととして、TEST コンパイラ・オプションの SEPARATE サブオプションも指定されている場合、Debug Tool によってこのソースがソース・ビューとして表示されます。

AFTERCICS の省略形として、ACICS を使用することができます。

AFTERMACRO

ソース・リストによって、MACRO プリプロセッサの最後の呼び出し (ある場合) での MDECK から得られたかのようにソースが表示されます。また、おそらくより重要なこととして、TEST コンパイラ・オプションの SEPERATE サブオプションも指定されている場合、Debug Tool によってこのソースがソース・ビューとして表示されます。

AFTERMACRO の省略形として、AMACRO を使用することができます。

AFTERSQL

ソース・リストによって、SQL プリプロセッサの最後の呼び出し (ある場合) での MDECK から得られたかのようにソースが表示されます。また、おそらくより重要なこととして、TEST コンパイラ・オプションの SEPARATE サブオプションも指定されている場合、Debug Tool によってこのソースがソース・ビューとして表示されます。

AFTERSQL の省略形として、ASQL を使用することができます。

TEST オプションが指定され、SOURCE 以外のサブオプションが LISTVIEW に指定されている場合、SEPARATE サブオプションも TEST オプションに指定する必要があります。

AFTERMACRO、AFTERSQL、および AFTERALL の各サブオプションによる異なる効果の一例として、PP オプションが PP(MACRO('INCONLY'), SQL, MACRO) であると仮定します。その場合には、以下のようになります。

- LISTVIEW(AFTERMACRO) を指定すると、リストと、Debug Tool のソース・ウィンドウ (TEST(SEP) が指定された場合) にある「ソース」は、MACRO プリプロセッサの 2 番目の呼び出しで生成された MDECK から得られたかのように表示されます。
- LISTVIEW(AFTERSQL) を指定すると、リストと、Debug Tool のソース・ウィンドウ (TEST(SEP) が指定された場合) にある「ソース」は、SQL プリプロセッサの呼び出しで生成された MDECK から得られたかのように表示されます (このため、%DCL およびその他のマクロ・ステートメントは表示されたままになります)。
- LISTVIEW(AFTERALL) を指定すると、MACRO プリプロセッサが PP オプションの最後にあるため、「ソース」は LISTVIEW(AFTERMACRO) オプションの場合と同様に表示されます。

MACRO

MACRO オプションを指定すると、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。MACRO オプションを使用すると、MACRO('macro-options') が、PP オプションに挿入されます。



省略形: NM、M

例えば、以下のように、コンパイル時オプションを指定します。

```
MDECK NOINSOURCE MACRO PP(MACRO SQL)
```


PP オプションは変更され、実際には、以下のようになります。

```
PP (MACRO MACRO SQL)
```

75 ページの『PP』も参照。

MARGINI

このオプションは、生成されるソース・リストに使用されるマージン・インディケータを指定します。



省略形: MI('char')

文字 *char* が、両側マージンに左端と右端に直ぐ近く的位置に挿入され、マージンの外側にはみ出たソース・コードを簡単に検出することが可能になります。

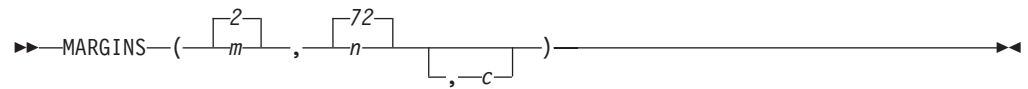
デフォルト: MARGINI(' ')

デフォルトを使用すると、左右のソース・マージンを空白の列によってリストに示すように指定されます。

サンプル・リストについては、139 ページの『コンパイラ・リストの使用』を参照してください。

MARGINS

このオプションにより、その内部にあるものをコンパイラがプログラム・ファイルのソース・コードとして解釈するマージンが設定されます。マージンの外側にあるデータは、ソース・コードとして解釈されませんが、ユーザーの要求次第でソース・リストに含められます。



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。*m* より大きくしますが、必ず、200 以下にします。

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。200 以下でなければならず、また *m* と *n* に指定された値の範囲外でなければなりません。*c* の値として 0 を指定すると、ANS 制御文字がないことを示します。使用できる制御文字は以下のみです。

(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替りの方法として、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)。

デフォルト: MARGINS (2 72)

MAXGEN



MAXGEN オプションは、どのユーザー・ステートメントに対しても生成される中間言語ステートメントの最大数を指定し、この最大数を超えたすべてのステートメントに、コンパイラーがフラグを立てるようにします。

任意のユーザー・ステートメントに対して生成される中間言語ステートメントの数は、コンパイラー・リリース、コンパイラー保守レベル、および有効なコンパイラー・オプションによって異なる可能性があります。このオプションは、過剰なコー

コンパイル時オプション

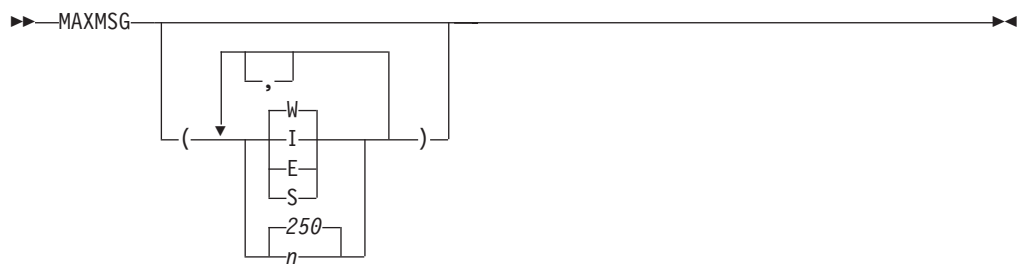
ドが生成されたため、コーディングが正常に行われていない可能性のあるステートメントを見つけるのに役立たせることのみを目的としています。

ただし、プリプロセッサを使用すると、一部のステートメントに対して生成される中間言語ステートメントの数が非常に多くなる可能性があることに注意してください。このような場合、MAXGEN しきい値をより高い値に設定するか、LISTVIEW(AFTERALL) オプションを使用する方がよい場合があります。

デフォルトは、MAXGEN(100000) 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。

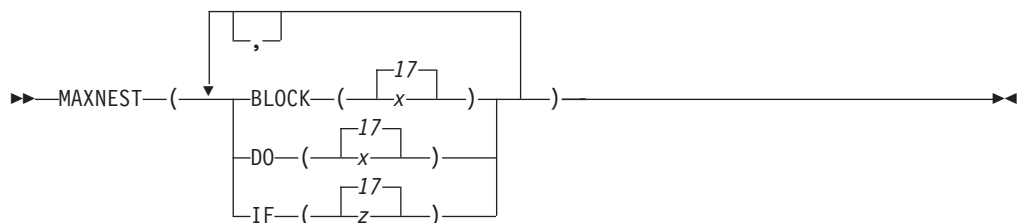


- I** すべてのメッセージを数えます。
- W** 情報メッセージを除くすべてのメッセージを数えます。
- E** 警告メッセージと通知メッセージを除くすべてのメッセージを数えます。
- S** 重大エラー・メッセージおよび回復不能エラー・メッセージだけを数えます。
- n** メッセージの数がこの値を超えた場合、コンパイルを終了します。指定の重大度より低いメッセージ、またはコンパイラ出口ルーチンによりフィルターに掛けられて取り除かれたメッセージは、カウントされません。n の値の範囲は 0 から 32767 です。0 を指定した場合、指定された重大度の最初のエラーが検出されるとコンパイルは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

デフォルト: MAXMSG(W 250)

MAXNEST

MAXNEST オプションは、プログラムが複雑すぎるとしてコンパイラがフラグを立てないように、各種ステートメントの最大許可ネストを指定します。



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。

ネスト制限の値は、1 および 50 (両端を含む) の間でなければなりません。

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

MAXSTMT

MAXSTMT オプションのもとで、MSG(390) オプションも有効になっている場合、コンパイラーは、指定された数を超えるステートメントの存在するブロックにフラグを立てます。ただし、Windows では、上記のようなブロックに対する最適化がオフになることはありません。

▶▶—MAXSTMT—(サイズ)—▶▶

デフォルト: MAXSTMT(4096)

MAXTEMP

MAXTEMP オプションは、コンパイラーが生成する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た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

このオプションを指定すると、マクロ機能出力ソースは、.DEK というファイル拡張子で書き込まれ、ファイルは、現行ディレクトリーに格納されます。

▶▶—NOMDECK
MDECK—▶▶

省略形: NMD、MD

このオプションは、ライブラリー・コールを通して行われる変換に対するメッセージを、コンパイラが発行する時点を制御します。

* ライブラリー・コールを通して行われる変換に対して、その変換が、コードをコンパイルしているプラットフォーム上で行われる場合に限り、コンパイラーが警告メッセージを発行するようにします。

ライブラリー・コールを通して行われる変換に対して、その変換が、390 上で行われる場合に限り、コンパイラーが警告メッセージを発行するようにします。

このオプションでは、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。

特別言語文字。

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。

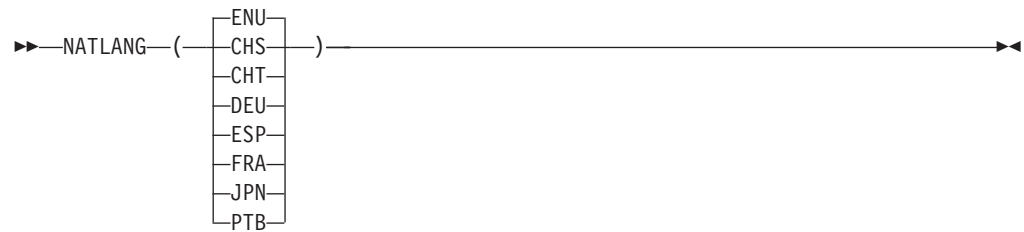
2 番目のサブオプションを省略すると、PL/I は最初のサブオプションで指定された文字を小文字と大文字の両方として使用します。2 番目のサブオプションを指定する場合は、最初のサブオプションで指定したのと同じ数の文字を指定しなければなりません。

例:

```
names('äöüß' 'ÄÖÜß')
```

NATLANG

このオプションは、コンパイラー・メッセージとリストに使用される各国語を設定します。



CHS

中国語 (簡体字)

CHT

中国語 (繁体字)

DEU

ドイツ語

ENU

米国英語、大/小文字混合。

ESP

スペイン語

FRA

フランス語

JPN

日本語

PTB

ブラジル・ポルトガル語

デフォルト: NATLANG(ENU)

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。



ソース・リストの例については、139 ページの『コンパイラー・リストの使用』を参照してください。

NOT

このオプションは、論理 NOT 記号として解釈される記号を最大 7 個まで指定します。

**char**

単一文字の記号です。26 個の英字、10 個の数字、および「PLI 言語解説書」に定義されている特殊文字（論理 NOT 記号 (^) を除く）は、一切指定しないでください。

デフォルト: NOT (^)

NOT 記号に対する PL/I デフォルト・コード・ポイントの値は、16 進数値の 5E となり、多くの端末上で、論理 NOT 記号 (^) として表示されます。

コマンド行からコンパイラーを呼び出し、NOT オプションの一部として脱字記号 (^) を指定する場合は、その脱字記号の前にもう一つ脱字記号を置く必要があります。

例:

```
not('¥')
not('^¥')
```

コマンド行上で、コンパイラーを呼び出し、垂直バー (|) または脱字記号 (^) を使用するコンパイル時オプションを指定する場合は、二重引用符を使用してその文字を囲ってください。

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、SOURCE および XREF オプションから作成されるコンパイラ・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。

NUMBER と STMT は相互に排他的です。NONNUMBER を指定すると、STMT を暗黙指定したことになります。

省略形: NUM、NNUM

OBJECT

このオプションでは、オブジェクト・コードを生成するかどうかを指定します。



省略形: OBJ、NOBJ

モジュールは、現行ディレクトリーに保管されます。

OFFSET

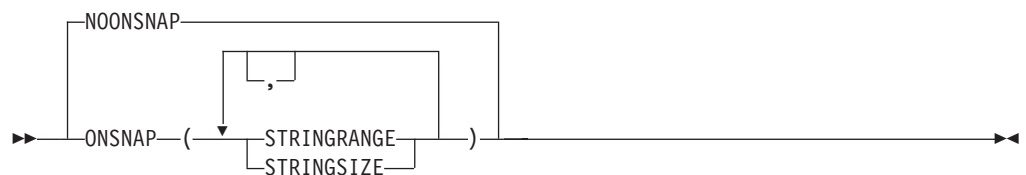
このオプションでは、コンパイラーが、拡張子 .cod を持つアセンブラー風のリスト・ファイルを生成するかどうかを指定します。



省略形: OF、NOF

.cod ファイルには、生成される各命令に対するオフセットとマシン・コードが格納されます。サンプル・プログラム cod2off を使用して、このファイルのサイズを縮小して、コンパイルのすべてのブロックの各ステートメントの開始位置に対するオフセット・リストだけにします。

ONSNAP

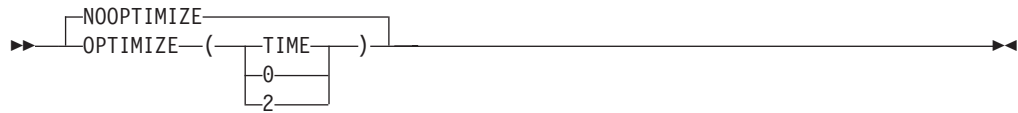


OPTIONS(MAIN) 属性または OPTIONS(FROMALIEN) 属性を持つ PROCEDURE に対して、ONSNAP オプションは、ON STRINGRANGE SNAP; ステートメントまたは ON STRINGSIZE SNAP; ステートメント、あるいはその両方が、コンパイラーによってその PROCEDURE のプロローグ・コードに挿入されることを指定します。これによって、そのような PROCEDURE から呼び出されるその他のルーチンで該当する条件が発生した場合に、呼び出しチェーンをより簡単に判別することができます。

ONSNAP オプションは、これらの属性のいずれも持っていない PROCEDURE には影響を及ぼしません。

OPTIMIZE

このオプションでは、必要な最適化のタイプを指定します。



省略形: NOPT、OPT

NOOPTIMIZE または OPTIMIZE(0)

上記のオプションのいずれかを使用して、コンパイルが可能な限り迅速に進行するように、オブジェクト・コードの標準的な最適化コードを生成します。

OPTIMIZE(TIME) または OPTIMIZE(2)

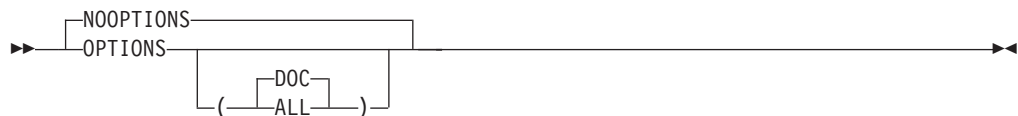
上記のオプションのいずれかを使用して、オブジェクト・コードの拡張最適化を進め、より高速に実行されるオブジェクト・コードを生成します。最適化には、コンパイル時間が余分に必要になりますが、通常、実行時間は短くなります。

インライン化は、最適化の指定時に限り行われます。

最適化の詳細な説明については、341 ページの『第 20 章 パフォーマンスの向上』を参照してください。

OPTIONS

このオプションにより、コンパイル作業に有効なすべてのコンパイル時オプションのリストが表示されます (例については、139 ページの『第 8 章 コンパイル出力』を参照してください)。



省略形: NOP、OP

OPTIONS(DOC) の場合、OPTIONS リストは、コンパイラーのリリース時点で本書に記載のオプション (およびサブオプション) のみを含みます。

OPTIONS(ALL) の場合、OPTIONS リストは、コンパイラーのリリース後に PTF によって追加されたオプションも含みます。

OR

このオプションは、論理 OR 記号 (|) として解釈される記号を最大 7 個まで指定します。指定した記号は、(対にすると) 連結記号としても使用されます。



char

単一文字の記号です。26 個の英字、10 個の数字、および「PL/I 言語解説書」に定義されている特殊文字 (論理 OR 記号 (|) を除く) は、一切指定しないでください。

コンパイラを呼び出し、コマンド行で OR オプションの一部として垂直バー (|) を指定する場合は、垂直バーの前に脱字記号 (^) を置く必要があります。

デフォルト: OR (|)

OR 記号 (|) に対する PL/I デフォルト・コード・ポイントは、16 進数値の 7C です。

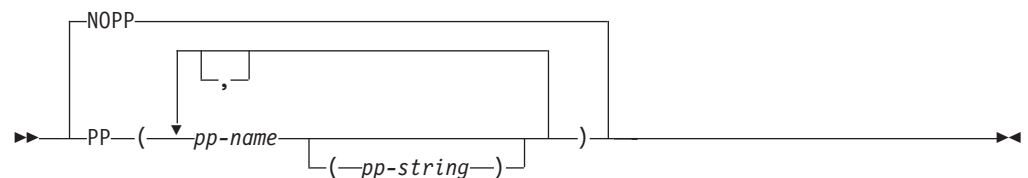
例:

```
or('¥')
or('^|¥')
```

コマンド行上で、コンパイラを呼び出し、垂直バー (|) または脱字記号 (^) を使用するコンパイル時オプションを指定する場合は、二重引用符を使用してその文字を囲んでください。

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。CICS、INCLUDE、MACRO、および SQL のみが、現在サポートされているプリプロセッサです。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 文字のストリングです (引用符で区切られています)。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。

コンパイル時オプション

デフォルト: NOPP

プリプロセッサ・オプションは、左から右に処理されます。2つのオプションが競合する場合、最後(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 プリプロセッサが呼び出された場合、オプション CASE(UPPER) が使用されます。

プリプロセッサは、同じものを複数回指定できます。さらに、最大で 31 個のプリプロセッサ・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する場合、コンパイラはその PP オプションを効率的に連結します。そのため、PP(SQL) PP(CICS) と指定しても PP(SQL CICS) と指定しても同じ結果になります。これにより、例えば PP(MACRO SQL('OPTIONS')) と PP(MACRO SQL('OPTIONS DATE(ISO)')) が指定された場合、PP オプションは結果として PP(MACRO SQL('OPTIONS') MACRO SQL('OPTIONS DATE(ISO)')) となり、MACRO と SQL の両方のプリプロセッサが 2 回呼び出されることにもなります。前の SQL オプションをオーバーライドしようとしてこの操作を行っている場合、PP オプションでプリプロセッサ・オプションを指定するのではなく、PPSQL オプションでプリプロセッサ・オプションを指定することをお勧めします。すなわち、PP(MACRO SQL) PPSQL('OPTIONS DATE(ISO)') のように指定します。

最高 31 つのプリプロセッサを指定できます。

例:

以下の例では、PL/I マクロ機能、SQL プリプロセッサを呼び出した後、再度、PL/I マクロ機能を呼び出します。

```
pp(macro('x') sql('dbname(sample)') macro)
```

PPCICS

PPCICS オプションは、CICS プリプロセッサが呼び出されたときにその CICS プリプロセッサに渡されるオプションを指定します。

▶ 

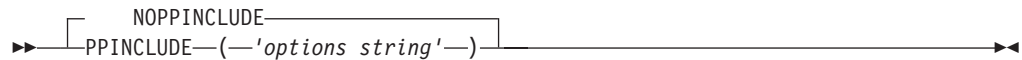
そのため、PPCICS('EDF') PP(CICS) と指定しても PP(CICS('EDF')) と指定しても同じ結果になります。

このオプションは、PP(CICS) オプションが指定されていない場合は無効です。ただし、CICS プリプロセッサの呼び出し時に使用される 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。PP(CICS) が指定されている場合は、PPCICS オプションに指定されているオプションのセットは必ず使用されます。

また、プリプロセッサの呼び出し時に指定されたオプションは、PPCICS オプションに指定されたオプションに優先します。そのため、PPCICS('EDF') PP(CICS('NOEDF')) と指定した場合は、PP(CICS('EDF NOEDF')) と指定した場合や、さらに簡単にした PP(CICS('NOEDF')) を指定した場合と同じ結果になります。

PPINCLUDE

PPINCLUDE オプションは、INCLUDE プリプロセッサが呼び出されたときにその INCLUDE プリプロセッサに渡されるオプションを指定します。



そのため、PPINCLUDE('ID(-inc)') PP(INCLUDE) と指定しても
PP(INCLUDE('ID(-inc)')) と指定しても同じ結果になります。

このオプションは、PP(INCLUDE) オプションが指定されていない場合は無効です。ただし、INCLUDE プリプロセッサの呼び出し時に使用される 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。PP(INCLUDE) が指定されている場合は、PPINCLUDE オプションに指定されているオプションのセットは必ず使用されます。

また、プリプロセッサの呼び出し時に指定されたオプションは、PPINCLUDE オプションに指定されたオプションに優先します。そのため、PPINCLUDE('ID(-inc)') PP(INCLUDE('ID(+include)')) と指定した場合は、PP(INCLUDE('ID(-inc) ID(+include)')) と指定した場合や、さらに簡単にした PP(INCLUDE('ID(+include)')) を指定した場合と同じ結果になります。

PPMACRO

PPMACRO オプションは、MACRO プリプロセッサが呼び出されたときにその MACRO プリプロセッサに渡されるオプションを指定します。



そのため、PPMACRO('CASE(ASIS)') PP(MACRO) と指定しても PP(MACRO('CASE(ASIS)')) と指定しても同じ結果になります。

このオプションは、PP(MACRO) オプションが指定されていない場合は無効です。ただし、MACRO プリプロセッサの呼び出し時に使用される 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。MACRO または PP(MACRO) オプションが指定されている場合は、PPMACRO オプションに指定されているオプションのセットは必ず使用されます。

また、プリプロセッサの呼び出し時に指定されたオプションは、PPMACRO オプションに指定されたオプションに優先します。そのため、PPMACRO('CASE(ASIS)')

PP(MACRO('CASE(UPPER)')) と指定した場合は、PP(MACRO('CASE(ASIS) CASE(UPPER)')) と指定した場合や、さらに簡単にした PP(MACRO('CASE(UPPER)')) を指定した場合と同じ結果になります。

PPSQL

PPSQL オプションは、SQL プリプロセッサが呼び出されたときにその SQL プリプロセッサに渡されるオプションを指定します。



そのため、PPSQL('ONEPASS') PP(SQL) と指定しても PP(SQL('ONEPASS')) と指定しても同じ結果になります。

このオプションは、PP(SQL) オプションが指定されていない場合は無効です。ただし、SQL プリプロセッサの呼び出し時に使用される 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。PP(SQL) が指定されている場合は、PPSQL オプションに指定されているオプションのセットは必ず使用されます。

また、プリプロセッサの呼び出し時に指定されたオプションは、PPSQL オプションに指定されたオプションに優先します。そのため、PPSQL('ONEPASS') PP(SQL('TWO PASS')) を指定した場合は、PP(SQL('ONEPASS TWO PASS')) と指定した場合や、さらに簡単にした PP(SQL('TWO PASS')) を指定した場合と同じ結果になります。

PPTRACE

このオプションを指定すると、プリプロセッサ用の DECK ファイルが書き込まれるとき、そのファイル内の各非ブランク行の前に、%LINE ディレクティブを含む行が追加されます。このディレクティブは、その非ブランク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。



PPTRACE は、PL/I コンパイラに組み込まれていないプリプロセッサのみと使用してください。

PRECTYPE

PRECTYPE オプションは、オペランドが FIXED BIN で、精度のみが指定されているときに、コンパイラーが MULTIPLY、DIVIDE、ADD、および SUBTRACT 組み込み関数の属性を導き出す方法を決定します。



PRECTYPE オプションには、以下の 3 つのサブオプションがあります。

ANS

PRECTYPE(ANS) では、BIF(*x,y,p*) の値 *p* は 2 進数の指定と解釈されます。演算は 2 項演算として実行され、その結果には属性 FIXED BIN(*p,0*) が付与されます。

DECDIGIT

PRECTYPE(DECDIGIT) では、BIF(*x,y,p*) の値 *p* は 10 進数の指定と解釈されます。演算は 2 項演算として実行され、その結果には属性 FIXED BIN(*s*) が付与されます (*s* は *p* (すなわち、 $s = \text{ceil}(3.32 * p)$) に等しい、対応する 2 進数です)。

DECRESULT

PRECTYPE(DECRESULT) では、BIF(*x,y,p*) の値 *p* は 10 進数の指定、および DECDIGIT では true と解釈されますが、演算は 10 進演算として実行され、その結果には属性 FIXED DEC(*p,0*) が付与されます。

PRECTYPE(ANS) がデフォルトです。

PREFIX

このオプションを指定すると、ソース・プログラムを変更することなく、コンパイル対象のコンパイル単位内に指定した PL/I 条件を有効または無効にできます。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。



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)

例:

コンパイル時オプション

以下のソースを仮定します。

```
(stringsize):  
name: proc options (reentrant reorder);  
end;
```

オプション `prefix (size nounderflow)` は、以下のようにプログラムを論理的に変更します。

```
(size nounderflow):  
(stringsize):  
name: proc options (reentrant reorder);  
end;
```

PROBE

このオプションは、スタックが 2K バイト以上拡張可能なときに必ずコンパイラーが生成する追加命令であるスタック・プローブの生成を制御します。この追加コードにより、スタック上に使用可能な十分な記憶域が存在しない場合は、記憶保護例外が発生します。



PROBE

スタック・プローブが生成されることを指定します。

NOPROBE

スタック・プローブを生成しません。

相当な自動ストレージを必要とし、不十分なスタック・サイズとリンクされているプログラムの場合は、スタック・プローブが生成されない限り、例外を起こし、無限ループに入る可能性があります。スタック・プローブの存在により、正常にリンクされた非マルチスレッド・プログラムのパフォーマンスが低下します。

PROCEED

このオプションにより、前の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、(プリプロセッサまたはコンパイラーによる) 処理が継続されるかどうかが決まります。



省略形: PRO、NPRO

PROCEED

プリプロセッサによって、現在の処理段階の前に発行されたメッセージに関係なく、プリプロセッサの呼び出しとコンパイラーは、動作を継続します。

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)

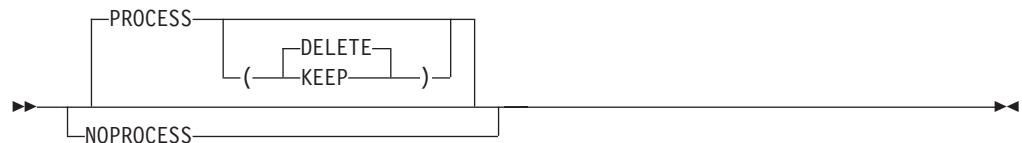
このプリプロセスの段階でエラー、重大エラー、または回復不能エラーが検出された場合は、プリプロセッサおよびコンパイラーの呼び出しは継続されません。

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は、それらが MDECK ファイルに書き込まれるかどうかを決定します。



NOPROCESS オプションの場合、コンパイラーは E レベルのメッセージですべての *PROCESS ステートメントにフラグを立てます。

PROCESS(KEEP) オプションの場合、コンパイラーは *PROCESS ステートメントにフラグを立てずに、すべての *PROCESS ステートメントを MDECK 出力に保持します。

PROCESS(DELETE) オプションの場合、コンパイラーは *PROCESS ステートメントにフラグを立てませんが、いずれの *PROCESS ステートメントも MDECK 出力に保持しません。

QUOTE

QUOTE オプションは、引用符文字として使用可能な代替記号を最大 7 つ指定します。



注: 引用符の間に空白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。

コンパイル時オプション

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

GRAPHIC オプションも指定した場合、QUOTE オプションは無視されます。

REDUCE

REDUCE オプションは、コンパイラーが、構造に対するヌル・ストリングの割り当てを、よりシンプルな操作 (埋め込みバイトが上書きされるだけの場合もあります) に縮小するのを許可されていることを指定します。



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、下の構造には、*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。

しかし、REDUCE を指定した場合、割り当ては 3 つの操作に縮小されますが、埋め込みバイトはゼロにリセットされます。

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

コンパイラーに対して、指定された DATE 属性を受け入れ、その DATE 属性を DATE 組み込み関数の結果に適用するように指示します。

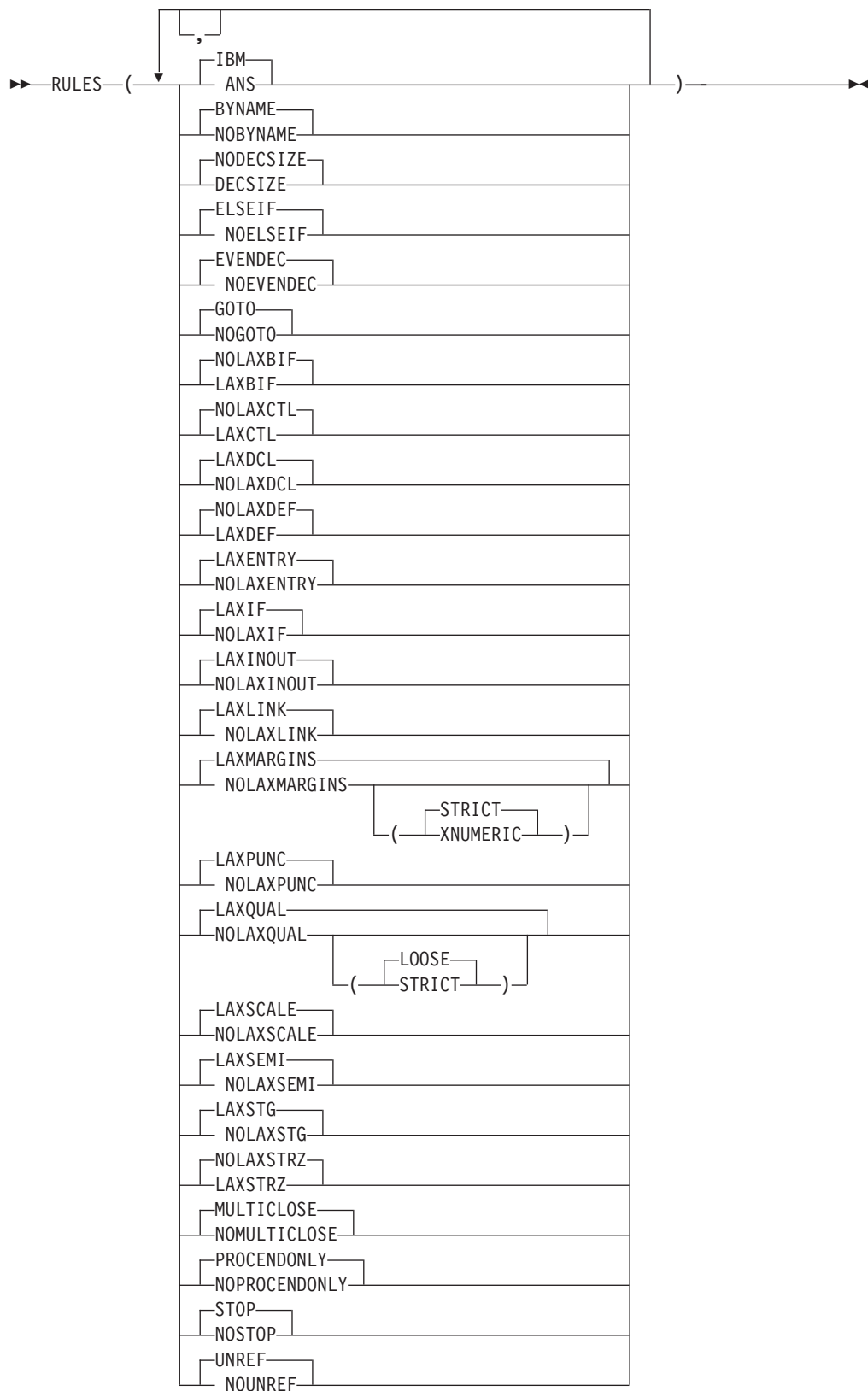
►►—RESPECT—(—DATE—)—►►

デフォルトを使用すると、コンパイラーは、指定されている DATE 属性を無視するため、DATE 属性は、DATE 組み込み関数の結果に適用されません。

RULES

このオプションは、特定の言語機能を使用可能または使用不可にし、代替の選択肢が提供されている場合はセマンティクスを選択するために使用できます。これは一般プログラミング・エラーの診断に役立ちます。

コンパイル時オプション



IBM または 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。
- 算術演算または比較での変換は、「*pre-Enterprise 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、「*pre-Enterprise 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` であっても、結果は必ず `SIGNED` となります。
- 2 つの `UNSIGNED FIXED BIN` オペランドを用いて加算、乗算、または除算を行なった場合でも、結果の属性は `SIGNED` となります。
- 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、結果の属性は `SIGNED` となります。

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` 組み込み関数または `MIN` 組み込み関数の引数がすべて、`UNSIGNED FIXED BIN` である場合は、結果も `UNSIGNED` となります。
- 2 つの `UNSIGNED FIXED BIN` オペランドを用いて `ADD`、`MULTIPLY`、または `DIVIDE` を行なった場合も、結果の属性は `UNSIGNED` 属性となります。
- 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。

BYNAME または NOBYNAME

`NOBYNAME` を指定すると、コンパイラーは 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立てます。

DECSIZE または NODECSIZE

`DECSIZE` を指定すると、コンパイラーは、`SIZE` 条件が割り当てによって引き起こされる場合、`SIZE` 条件が使用不可になったときに、`FIXED DECIMAL` 式の `FIXED DECIMAL` 変数へのすべての割り当てにフラグを立てます。

RULES(DECSIZE) を指定すると、コンパイラーは、大量のメッセージを出すことがあります。これは、SIZE が使用不可にされた場合、 $X = X + 1$ の形式のステートメントにはすべてフラグが立てられるためです (X が FIXED DECIMAL の場合)。

ELSEIF または NOELSEIF

NOELSEIF を指定すると、コンパイラーは IF ステートメントが直後に続くすべての ELSE ステートメントにフラグを立て、それを SELECT ステートメントとして書き換えるよう提案します。

このオプションは、ネストした一連の 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ときに役立ちます。

EVENDEC または NOEVENDEC

NOEVENDEC を指定すると、コンパイラーは、偶数の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 ブロックの外にあるものを除き、すべての GOTO ステートメントにフラグが立てられます。

LAXBIF または NOLAXBIF

LAXBIF を指定すると、コンパイラーは、空のパラメーター・リストなしで使った場合でも、NULL などの組み込み関数にコンテキスト宣言を作成します。

LAXCTL または 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 の場合、以下のコードは正しくありません。

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

しかし、次のコードは NOLAXCTL でも有効です。

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

LAXDCL または 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不可能になります。

LAXDEF または 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 または NOLAXENTRY

LAXENTRY を指定すると、プロトタイプ化されていないエントリー宣言が可能

になります。NOLAXENTRY を指定すると、コンパイラーは、プロトタイプ化されていないすべてのエン트리宣言 (つまり、パラメーター・リストを指定していないすべての ENTRY 宣言) にフラグを立てます。これは、ENTRY にパラメーターがない場合、単に 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ていることに注意してください。

LAXIF または NOLAXIF

RULES(NOLAXIF) を指定すると、コンパイラーは、BIT(1) NONVARYING 属性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 の場合、以下の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 (および場合により出力) パラメーターであると想定し、そのため、このようなパラメーターが初期化されていないと考えられる場合は警告を出します。

LAXLINK または NOLAXLINK

NOLAXLINK を指定すると、コンパイラーは、以下のいずれかが一致しない場合に、2 つの ENTRY 変数または定数のすべての代入または比較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)) として、および A2 が ENTRY(POINTER) VARIABLE として宣言されているときに RULES(NOLAXLINK) の場合、コンパイラーは、A1 を A2 に代入する試みにフラグを立てます。

- RETURNS 属性

例えば、A3 が ENTRY RETURNS(FIXED BIN(31)) として、および A4 が RETURNS 属性なしで ENTRY VARIABLE として宣言されているときに RULES(NOLAXLINK) の場合、コンパイラーは、A3 を A4 に代入する試みにフラグを立てます。

- LINKAGE およびその他の OPTIONS サブオプション

例えば、A5 が ENTRY OPTIONS(ASM) として、および A6 が OPTIONS 属性なしで ENTRY VARIABLE として宣言されているときに RULES(NOLAXLINK) の場合、コンパイラーは、A5 を A6 に代入する試みにフラグを立てます。これは、A5 の宣言中の OPTIONS(ASM) は、A5 が LINKAGE(SYSTEM) を持つことを意味し、その一方、A6 は OPTIONS 属性がないため、デフォルトで LINKAGE(OPTLINK) を持つからです。

LAXMARGINS または 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、誤って右マージンから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。

NOLAXMARGINS および **STMT** オプションがプリプロセッサのいずれかとともに使用される場合、**NOLAXMARGINS** オプションが原因でフラグが立てられたステートメントはステートメント・ゼロとして報告されます。(これは、すべてのプリプロセッサが完了してからステートメントの番号付けが行われるためです。ただし、ソースが読み取られると同時にマージンの外側にあるテキストが検出されます。)

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。ただし、右マージンが 72 列目であり、73 列目から 80 列目のすべてに数字が入っている場合、コンパイラーはフラグを立てません。

LAXPUNC または **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(LOOSE) を指定すると、コンパイラーは、レベル 1 ではなく、ドット修飾のない構造体メンバーへのすべての参照にフラグを立てます。次の例を見てください。

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

c  = 11; /* would be flagged */
b.c = 13; /* would not be flagged */
a.c = 17; /* would not be flagged */
```

NOLAXQUAL(STRICT) を指定すると、コンパイラーは、レベル 1 の名前を含まない構造体メンバーへのすべての参照にフラグを立てます。次の例を見てください。

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

c  = 11; /* would be flagged */
b.c = 13; /* would be flagged */
a.c = 17; /* would not be flagged */
```

LAXSCALE|NOLAXSCALE

NOLAXSCALE を指定すると、コンパイラーは、すべての FIXED BIN(p,q) または FIXED DEC(p,q) 宣言にフラグを立てます (q < p または p > q)。

LAXSEMI または NOLAXSEMI

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

LAXSTG または NOLAXSTG

NOLAXSTG を指定すると、コンパイラーは、B がたとえパラメーターであっても (ここが重要な部分である)、変数 A が ADDR(B) および STG(A) > STG(B) に BASED として宣言されている宣言にフラグを立てます。

コンパイラーは、B が AUTOMATIC または STATIC ストレージにあると、前からこの種の問題にフラグを立てますが、B がパラメーターであるときには、デフォルトでこれにフラグを立てません (お客様によっては、実引数を記述しないでプレースホルダー属性で B を宣言するからです)。パラメーターおよび引数の宣言が一致しない (または一致する必要のある) お客様にとって、RULES(NOLAXSTG) を指定することは、より多くのストレージ・オーバーレイの問題を検出するのに役立つ可能性があります。

LAXSTRZ または 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 または 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立て、E レベル・メッセージを出します。

PROCENDONLY|NOPROCENDONLY

NOPROCENDONLY を指定すると、PROCEDURE を閉じる END ステートメントでその PROCEDURE が指定されていない場合 (つまり、END キーワードのすぐ後にセミコロンが続く場合)、その END ステートメントにフラグが立てられます。

STOP|NOSTOP

NOSTOP を指定すると、すべての STOP および EXIT ステートメントにフラグが立てられます。

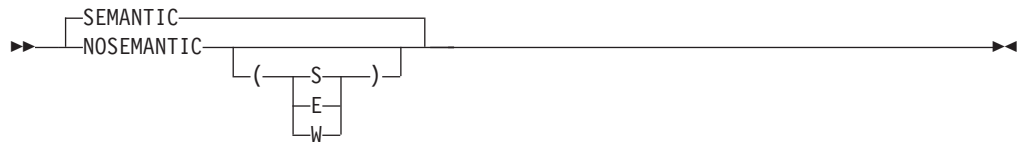
UNREF または NOUNREF

NOUNREF を指定すると、コンパイラーは、レベル 1 の AUTOMATIC 変数 (参照されておらず、かつそれが構造体または共用体であれば、参照されているサブエレメントを含まない変数) すべてにフラグを立てます。

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

SEMANTIC

この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。

SNAP

このオプションでは、例外が発生した場合の SNAP と PLIDUMP のトレースバック出力を完全なものにするかどうかを指定します。



SNAP

SNAP と PLIDUMP のトレースバック出力には、常に、コール・スタック上の PL/I ルーチンがすべて含まれます。SNAP は、プログラムのサイズを相当に増大し、パフォーマンスを低下させる場合があります。このオプションは、実動プログラムには推奨されません。

NOSNAP

SNAP と PLIDUMP のトレースバック出力は、完全な一揃いでない可能性があります。

SOSI

このオプションは、DBCS シフトアウト文字およびシフトイン文字を含むソースの処理方法を指定します。

```

>> [NOSOSI]
>> [SOSI]

```

NOSOSI

NOSOSI オプションを指定すると、これらの文字に対して特別な処理は行われません。

SOSI

SOSI オプションを指定すると、コンパイラーは、DBCS シフトアウト文字およびシフトイン文字を含むソースを受け入れ、正しく処理します。

SOURCE

SOURCE オプションを指定すると、コンパイラーへのソース入力のリストが作成されます。

```

>> [NOSOURCE]
>> [SOURCE]

```

省略形: S、NS

SOURCE

コンパイラーは、ソース・リストを作成します。

NOSOURCE

コンパイラーは、ソース・リストを作成しません。

ソース・リストは、構文検査が実行されない限り生成されません。

STATIC

STATIC オプションは、INTERNAL STATIC 変数を、オブジェクト・モジュールに(非参照としてであっても) 保存するかどうかを制御します。

```

>> [SHORT]
>> [FULL]
>> STATIC-( )

```

SHORT

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。

コンパイル時オプション

FULL

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

INTERNAL STATIC 変数が "eyecatchers" として使用される場合は、STATIC(FULL) オプションを指定して、生成されたオブジェクト・モジュールに入るようにします。

STMT

STMT オプションは、ソース・プログラム内のステートメントをカウントし、この「ステートメント番号」を使用して、AGGREGATE、ATTRIBUTES、SOURCE および XREF オプションを用いて作成されたコンパイラ・リスト内のステートメントを識別することを指定します。



NOSTMT を指定すると、NUMBER を暗黙指定したことになります。

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。

STORAGE

STORAGE オプションは、それぞれのプロシージャール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に作成させます。



省略形: STG、NSTG

SYNTAX

このオプションを指定すると、コンパイラの構文検査段階の実行は、この処理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: SYN、NSYN

SYNTAX

NOSYNTAX(S) と同等です。

NOSYNTAX

構文検査は実行されません。

NOSYNTAX(S)

重大エラーまたは回復不能エラーが検出された場合、構文検査は実行されません。

NOSYNTAX(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、構文検査は実行されません。

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、構文検査は実行されません。

NOSYNTAX オプションでコンパイルが強制終了されると、相互参照リスト、属性リスト、ソース・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。

SYSPARM

このオプションを指定すると、マクロ機能組み込み関数 **SYSPARM** が戻す字符串の値を指定できます。

▶▶—SYSPARM—(—'string' —)—————▶▶

string

この字符串の長さは最大 64 文字です。ヌル・字符串がデフォルトですが、字符串値を指定する場合には、『コンパイル時オプションの使用規則』のステップ 2 (39 ページ) の *strings* に関する注を参照してください。

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。

デフォルト: SYSPARM("")

SYSTEM

このオプションでは、**PL/I** プログラムがそのもとで実行されるオペレーティング・システムとハードウェア・プラットフォームを指定します。**MAIN** プロシージャが受け取れるパラメーターも強制します。

さらに、サブオプションを使用すると、オブジェクト・コードが実行されるハードウェア・プラットフォームを活用することが可能になります。

▶▶—SYSTEM—(—

WINDOWS
CICS
IMS

—)—————▶▶

WINDOWS

プログラムが、**WINDOWS** のもとで実行されることを指定します。

コンパイル時オプション

CICS

プログラムが、CICS のもとで実行されることを指定します。

IMS

プログラムが、IMS のもとで実行されることを指定します。

S486

オブジェクト・コードは、80486、または互換性のあるチップを搭載したマシン上で稼動します。コードは、Pentium チップを搭載したマシン上で稼動しますが、386 チップでは稼動しません。

Pentium

オブジェクト・コードは、Pentium チップを搭載したマシン上で稼動します。コードは、Pentium チップを搭載しないマシン上では稼動しません。

SYSTEM(CICS) でコンパイルされた MAIN プロシージャールの場合、OPTIONS (BYVALUE) が仮定され、指定された場合、PROCEDURE OPTIONS(BYADDR) が診断されます。

TERMINAL

このオプションは、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。



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 オプションでは、オブジェクト・コードの一部として生成されるテスト機能のレベルを指定します。この機能を使用すると、テスト・フックの位置を制御したり、記号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。



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。

NOTEST

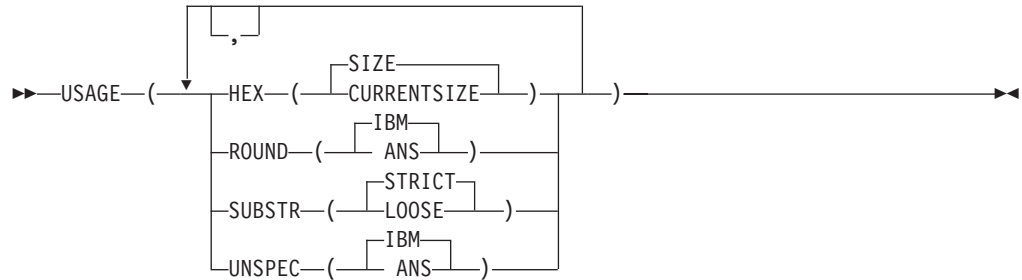
すべてのテスト情報の生成を抑止します。

TEST

テスト情報をオブジェクト・コードに組み込むことを指定します。

USAGE

USAGE オプションを使用すると、IBM または ANS 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。

**HEX(SIZE | CURRENTSIZE)**

HEX(SIZE) サブオプションを指定すると、HEX が VARYING または VARYINGZ スtringに適用される場合、Stringによって使用されるストレージの最大量を示す 16 進Stringが返されます。

HEX(CURRENTSIZE) サブオプションを指定すると、HEX が VARYING または VARYINGZ スtringに適用される場合、Stringによって使用されるストレージの現行量を示す 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) サブオプションを指定すると、x が CHARACTER タイプを持つ場合、SUBSTR(x,y,z) 組み込み関数参照によって MIN(z, MAXLENGTH(x)) と同じ長さのStringが返されます。

SUBSTR(LOOSE) サブオプションを指定すると、同じ参照によって、長さが z のStringが返されます。

SUBSTR(LOOSE) サブオプションは、SUBSTR(x,y,z) 参照 (x は CHAR(1) BASED 変数) を使用する場合に便利です。

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。

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。

►► WIDECHAR (LITTLEENDIAN
BIGENDIAN) ◀◀

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。

デフォルト: WIDECHAR(LITTLEENDIAN)

WINDOW

WINDOW オプションは、各種の日付関連の組み込み関数で使用される w ウィンドウ引数を設定します。

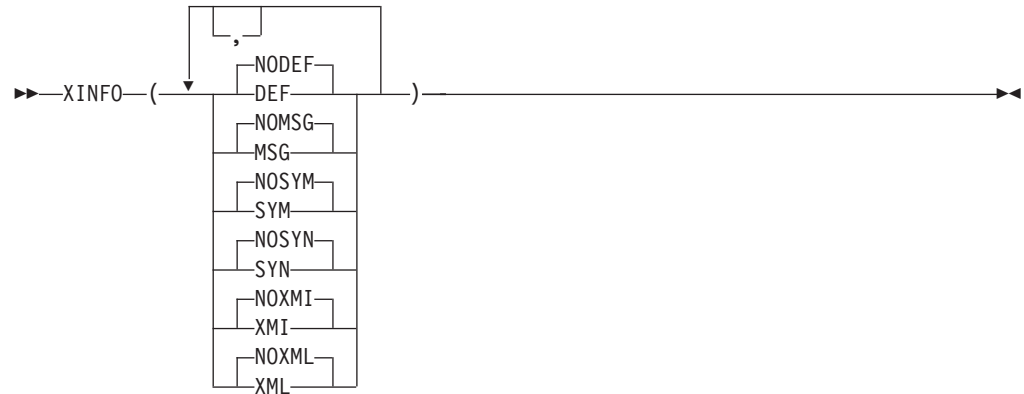
►► WINDOW (-w-) ◀◀

w の値は、固定ウィンドウの開始を示す符号なし整数、または「スライディング」ウィンドウを指定する負の整数のいずれかになります。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。

デフォルト: WINDOW(1950)

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
- 動的に呼び出されて取り出されるモジュール

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」が付きます。

例えば、次のプログラムがあるとして。

```
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** ファイルの形式の詳細については、付録を参照してください。

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、拡張子「adt」が付きます。

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

記号情報が ADATA ファイルに生成されます。

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、拡張子「adt」が付きます。

NOSYM

記号情報は 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。XINFO(SYN) オプションを指定すると、メモリーおよび生成されるファイルの両面で、コンパイラーに必要なストレージの量が大幅に増加する可能性があります。

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、拡張子「adt」が付きます。

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。この XMI は、他のツールによる場合を除いて、読み取りまたは解釈されるものではありません。

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」が付きます。

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

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」が付きます。

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 INCLUDEFROMFILE (#PCDATA)>
<!ELEMENT INCLUDEDONLINE (#PCDATA)>

```

NOXML

XML サイド・ファイルは作成されません。

XML

XML オプションを使用すると、XMLCHAR 組み込み関数によって生成される XML の名前の大/小文字を選択できます。

```

>> XML ( ( CASE ( ( UPPER
                ASIS
            ) ) ) )

```

CASE(UPPER | ASIS)

CASE(UPPER) サブオプションの場合、XMLCHAR 組み込み関数によって生成される XML で名前はすべて大文字になります。

CASE(ASIS) サブオプションの場合、XMLCHAR 組み込み関数によって生成される XML で名前は、その宣言で使用されている大/小文字になります。

MACRO プリプロセッサをマクロ・プリプロセッサ・オプション

CASE(ASIS) なしで使用すると、コンパイラに見えるソースはすべての名前が大文字になり、それにより XML(CASE(ASIS)) オプションの指定は役立たなくなります。

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。

```

>> NOXREF
    XREF ( ( FULL
          SHORT
        ) )

```

省略形: X、NX

NOXREF

コンパイラがリストの一部としてこの情報を生成しないことを指示します。

XREF

コンパイラが、相互参照リストを生成することを指定します。

相互参照リスト以外にも、コンパイラは、未参照 ID のリストも生成します。そのリストでは、名前付き定数または静的割り当て不可変数である場合、変数は現れません。共用体または構造体内のフィールドが参照される場合も、共用

コンパイル時オプション

体または構造体の名前は現れません。そのメンバーが一切参照されない場合に限り、共用体または構造体のレベル 1 の名前が現れます。

相互参照テーブルの例および内容の説明については、139 ページの『コンパイラー・リストの使用』を参照してください。XREF と ATTRIBUTES を両方とも指定すると、2 つのリストが結合されます。

第 7 章 PL/I プリプロセッサ

インクルード・プリプロセッサ	102
例:	102
インクルード・プリプロセッサ・オプション環 境変数	102
マクロ・プリプロセッサ	103
マクロ・プリプロセッサのオプション	103
マクロ機能オプション環境変数	105
SQL プリプロセッサ	106
プログラミングとコンパイルに関する考慮事項	106
SQL プリプロセッサのオプション	107
SQL プリプロセッサ・オプション環境変数	115
SQL プリプロセッサ BIND 環境変数	115
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	115
SQL 連絡域の定義	115
SQL 記述子域の定義	116
SQL ステートメントの組み込み	117
ホスト変数の使用	118
SQL および PL/I の同等なデータ型の判別	119
ラージ・オブジェクト (LOB) サポート	121
LOB に関する一般情報	121
LOB サポートのための PL/I 変数宣言	122
LOB サポートのためのサンプル・プログラム	123
ユーザー定義関数のサンプル・プログラム	124
SQL データ型と PL/I データ型の互換性の判 別	126

ホスト構造体の使用	126
標識変数の使用	127
ホスト構造体の例	128
CONNECT TO ステートメント	128
DECLARE TABLE ステートメント	128
DECLARE STATEMENT ステートメント	129
論理 NOT 記号 (¬)	129
SQL エラー戻りコードの処理	129
DFT(EBCDIC NONNATIVE) のもとでの可変 長ストリングの使用	130
DEFAULT(EBCDIC) コンパイル時オプション の使用	130
SQL 互換性と移行に関する考慮事項	130
CICS サポート	132
プログラミングとコンパイルに関する考慮事項	132
CICS プリプロセッサのオプション	134
CICS プリプロセッサ・オプション環境変数	135
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	135
CICS ステートメントの組み込み	135
PL/I を使用した CICS トランザクションの作成	136
PL/I プログラムに使用される CICS 異常終了	137
CICS ランタイム・ユーザー出口	137

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 つまたは複数選択できます。インクルード・プリプロセッサ、マクロ機能、SQL プリプロセッサまたは CICS プリプロセッサ、およびそれらの呼び出し順序を選択できます。

- ・ 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。
- ・ マクロ機能は、% ステートメントやマクロに基づいて、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。構成ファイルの中の対応する属性を使用して、プリプロセッサそれぞれのデフォルト・オプションを設定することができます。

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。最初の一続きの非ブランク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 つ以上のブランク、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)
```

例:

次の 1 つ目の例では、-INC (および場合によっては先行ブランク) 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ブランク) 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)'))
```

インクルード・プリプロセッサ・オプション環境変数

IBM.PPINCLUDE 環境変数を使用してインクルード・プリプロセッサのデフォルト・オプションを設定することができます。 31 ページの『IBM.PPINCLUDE』を参照してください。

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 つまたはそれ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。

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—(— ) —▶▶

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。

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

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。

INCONLY

このオプションは、プリプロセッサが %INCLUDE および %XINCLUDE 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な場合は、以下のマクロとして INCLUDE も XINCLUDE も使用できません。

- プロシージャー名
- ステートメント・ラベル
- 変数名

NOINCONLY

このオプションは、プリプロセッサが %INCLUDE および %XINCLUDE ステートメントのみではなく、すべてのプリプロセッサ・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 INCONLY オプションは相互に排他的で、互換性のために NOINCONLY がデフォルトです。

NAMEPREFIX

このオプションは、プリプロセッサ・プロシージャーおよびプリプロセッサ変数の名前を、指定された文字で開始する必要があることを指定します。

▶▶—NAMEPREFIX—(キャラクター型)————▶▶

文字は「そのまま」指定される必要があります。引用符で囲んではなりません。

NONAMEPREFIX

このオプションは、プリプロセッサ・プロシージャーおよびプリプロセッサ変数の名前を、特定の 1 文字で開始する必要がないことを指定します。

NONAMEPREFIX 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/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/小文字を区別しません。

ASIS

再スキャンは大/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/小文字の区別を無視し、RESCAN(ASIS) は、大/小文字を区別します。

set IBM.PPMACRO コマンドを使用して、マクロ・プリプロセッサのデフォルト・オプションを設定することができます。

マクロ機能オプション環境変数

マクロ機能のデフォルト・オプションは、IBM.PPMACRO 環境変数を使用して設定することができます。 31 ページの『IBM.PPMACRO』を参照してください。

SQL プリプロセッサ

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。EXEC SQL サポートを活用するには、IBM DB2 Universal Database (以降、DB2 と呼びます) for Windows をインストールしておく必要があります。

ワークステーション PL/I 製品は、DB2 のほとんどの機能に対応し、リリースごとに機能の追加が図られます。下位レベルの DB2 製品の使用中に新規の DB2 機能を指定すると、警告メッセージが生成され、指定された機能のオプションは無視されます。

プログラミングとコンパイルに関する考慮事項

PL/I SQL サポートの使用時には、特定のオプションを考慮する必要があります。以下のテーブルに、そうした考慮事項を説明します。

表 3. EXEC SQL サポートに対する考慮事項

ターゲット・システム	使用する コンパイル時オプション
DB2 for Windows をネイティブ・モードで使用する Windows	DEFAULT (ASCII NATIVE IEEE)
DB2 for Windows をネイティブ・モードで使用する CICS	DEFAULT (ASCII NATIVE IEEE)
DB2 for Windows を z/OS エミュレーション・モードで使用する CICS VS/86	DEFAULT (EBCDIC NONNATIVE HEXADEC)
DB2 for Windows を z/OS エミュレーション・モードで使用する IMS	DEFAULT (EBCDIC NONNATIVE HEXADEC)
DB2 for Windows を z/OS エミュレーション・モードで使用する ISPF ダイアログ管理機能	DEFAULT (EBCDIC NONNATIVE)

PL/I ソース・プログラム内に存在する EXEC SQL ステートメントは、PP(SQL) オプションを使用して処理します。

```
pp(sql('option-string'))
```

前の例の 'option-string' は、引用符で囲まれた文字ストリングとなります。例えば、pp(sql('dbname(Sample)')) は、プリプロセッサに対して、SAMPLE データベースを使用して稼働するように指示します。

プログラムで EXEC SQL ステートメントを使用する場合は、例えば、次のように、リンク・コマンドに他のリンク・ライブラリーに加えて、SQL ライブラリーを指定する必要があります。

```
ilink myprog.obj db2api.lib
```

SQL ユーザー

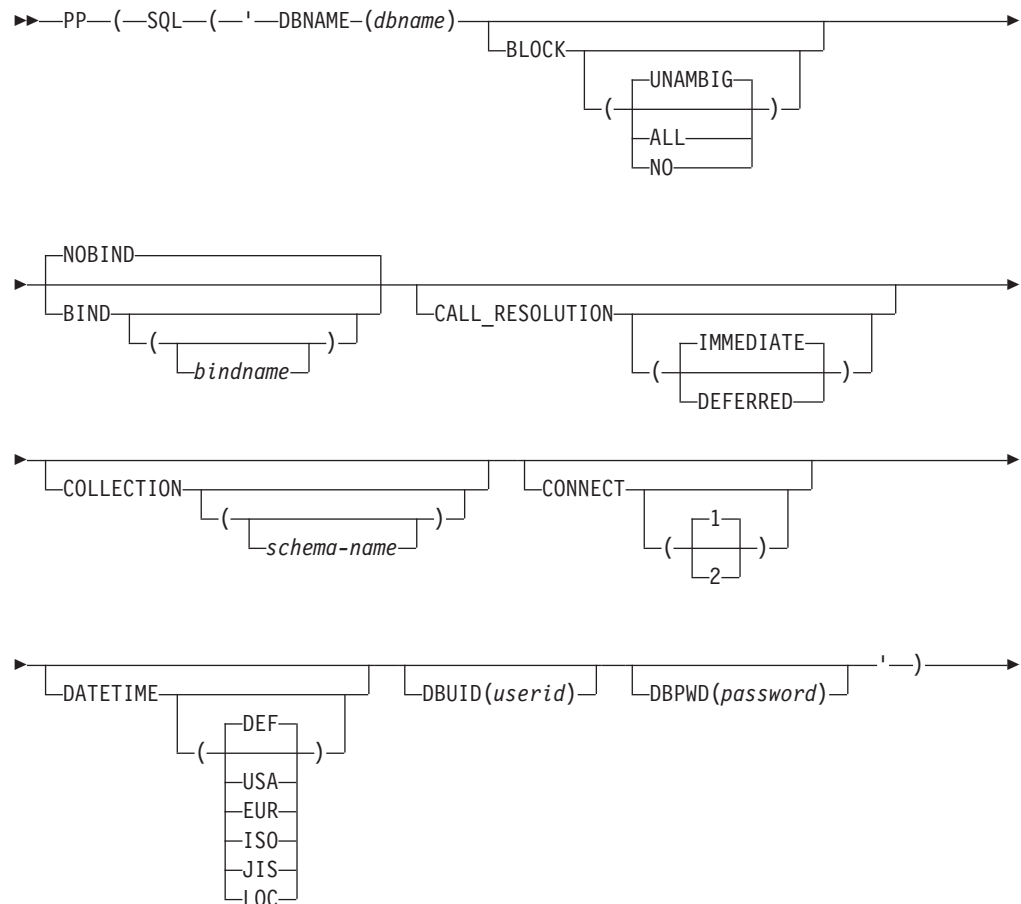
EXEC SQL ステートメントを含むプログラムをコンパイルするには、DB2 Universal Database for Windows をインストールし、始動しておく必要があります。DB2 のインストール方法については、使用しているプラットフォームのデータベース・インストール・ガイドを参照してください。

データベース・マネージャは、コマンド・プロンプトで以下のコマンドを実行して始動できます。

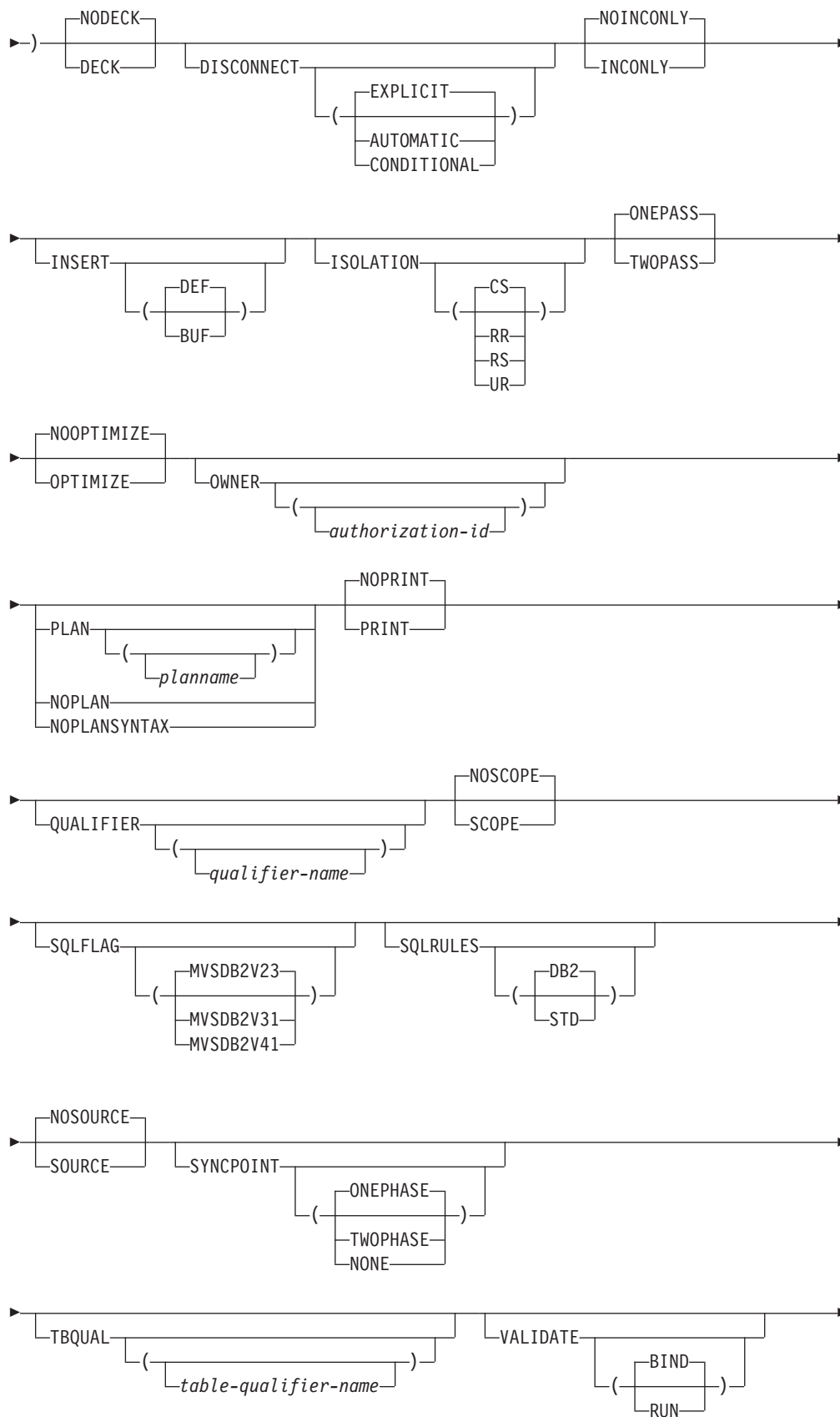
```
db2start
```

SQL プリプロセッサのオプション

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。



SQL プリプロセッサ





省略形:

DB、BLK、CRESO、DT、ISOL、ON、TW、S、NS、D、ND、OPT、NOPT、INS、COL、CON、DISC、SQLR、SYNC

DBNAME

データベースの元来の名前または別名を指定します。このオプションを使用して、指定したデータベースに対して、SQL ステートメントを処理するようにプリプロセッサに指示します。このオプションを省略するか、データベース名を指定しない場合、プリプロセッサは、暗黙的な接続が使用可能な場合には、デフォルト・データベースを使用します。デフォルト・データベースは、環境変数 DB2DBDFT によって指定されます。詳細な説明は、DB2 の資料を参照してください。

プリプロセッサには、連動するデータベースを用意する必要があります。用意しない場合は、エラーが発生します。

BLOCK

使用するレコード・ブロッキングのタイプおよび未確定カーソルの処理方法を指定します。このオプションに対する有効な値は、以下のとおりです。

UNAMBIG

ブロッキングは、読取専用カーソル、FOR UPDATE OF として指定されていないカーソル、静的な DELETE WHERE CURRENT OF ステートメントを持たないカーソル、および動的ステートメントを持たないカーソルに対して行われます。未確定カーソルは、更新可能です。

ALL

ブロッキングは、読取専用カーソル、FOR UPDATE OF として指定されていないカーソル、および静的な DELETE WHERE CURRENT OF ステートメントが実行されないカーソルに対して行われます。未確定かつ動的なカーソルは、読取専用として処理されます。

NO

ブロッキングは、パッケージ内のどのカーソルに対しても行われません。未確定カーソルは、更新可能です。

BIND または NOBIND

バインド・ファイル bindname を作成するかどうかを決定します。バインド・ファイルは、拡張子が .BND であり、現行ディレクトリーまたは IBM_BIND 環境変数によって指定されたディレクトリーのいずれかに保管されます。バンド名は、指定しない場合、入力ソース・ファイルの名前にデフォルト設定されます。

CALL_RESOLUTION

CALL ステートメントが推奨されない sqleproc() API の呼び出しとして、または通常の SQL ステートメントとして実行するかどうかを判別します。プリコンパイラーが CALL ステートメント上のプロシージャーを CALL_RESOLUTION IMMEDIATE で解決できない場合、SQL0204 が発行されることに注意してください。

IMMEDIATE

CALL ステートメントは、通常の SQL ステートメントとして実行されます。これはデフォルトです。

DEFERRED

CALL ステートメントは、推奨されない `sqleproc()` API の呼び出しとして実行されます。

COLLECTION

パッケージに対する 8 文字のコレクション ID を指定します。

schema-name

8 文字の ID です。

COLLECTION オプションには、デフォルト値は存在しません。COLLECTION を指定する場合は、`schema-name` も指定する必要があります。

CONNECT

データベースに対して行われる CONNECT タイプを指定します。

- 1 を指定すると、CONNECT コマンドが、タイプ 1 CONNECT として処理されます。これが、デフォルト設定です。
- 2 を指定すると、CONNECT コマンドが、タイプ 2 CONNECT として処理されます。

デフォルト・オプションの値は、CONNECT(1) です。

CON、CONNECT、CON()、および CONNECT() などのオプションのストリングは、CONNECT(1) と判断されます。

DATETIME

日付フィールドと時刻フィールドに、ホスト変数のストリング表現を割り当てるときに使用する日付形式と時刻形式を決定します。以下の 3 文字の省略形が、変数 *location* に対して有効です。

DEF データベースの国別コードに関連する日付/時刻形式を使用します。これは、DATETIME が指定されていない場合のデフォルトでもあります。

USA U.S. 形式に対する IBM 標準です。

日付形式: mm/dd/yyyy

時刻形式: hh:mm xM (AM or PM)

EUR ヨーロッパ形式に対する IBM 標準です。

日付形式: dd.mm.yyyy

時刻形式: hh.mm.ss

ISO 国際標準化機構

日付形式: yyyy-mm-dd

時刻形式: hh.mm.ss

JIS 日本工業規格

日付形式: yyyy-mm-dd

時刻形式: hh.mm.ss

LOC ローカル形式で、DEF と必ずしも等しくなるとは限りません。

DBUID または DBPWD

リモート接続の試行時に、ユーザー ID とパスワードの指定を要求するデータベース・マネージャーに対して、*userid* と *password* を指定することを可能にし

ます。例えば、ユーザー ID やパスワードは、Windows サーバーに常駐するリモート・データベースに対するコンパイル時に必要となる場合があります。

オプションの DBUID と DBPWD は大/小文字を区別しませんが、*userid* 値 (最大長 8 文字) および *password* の値 (最大長 18 文字) は、両方とも、大/小文字を区別します。

ユーザー ID とパスワードは、コンパイル処理時にデータベース・マネージャに接続するため、SQL プリプロセッサのみが使用します。アプリケーションが、実行時に接続する場合は、その接続に対するユーザー ID とパスワードをプログラム内の EXEC SQL CONNECT ステートメントで供給する必要があります。

DECK または NODECK

このオプションを指定すると、SQL プリプロセッサ出力ソースは、拡張子 .DEK を持つファイルに書き込まれ、そのファイルは現行ディレクトリに入れます。

DISCONNECT

データベースに対して行われる DISCONNECT タイプを指定します。

EXPLICIT

このオプションを指定すると、RELEASE ステートメントで明示的に解放用とマークされたデータベース接続のみが、コミット時に切断されます。これが、デフォルト設定です。

AUTOMATIC

このオプションを指定すると、すべてのデータベース接続が、コミット時に切断されます。

CONDITIONAL

このオプションを指定すると、RELEASE とマークされているか、開いている WITH HOLD カーソルを持たないデータベース接続が、コミット時に切断されます。

デフォルト・オプションの値は、DISCONNECT(EXPLICIT) です。DISC、DISCONNECT、DISC()、DISCONNECT() などのオプションのストリングは、DISCONNECT(EXPLICIT) と判断されます。

INONLY または NOINONLY

SQL プリプロセッサで EXEC SQL INCLUDE ステートメントのみを処理するかどうかを決定します。INONLY が指定されると、SQL プリプロセッサによってコードは生成されず、すべての EXEC SQL INCLUDE ステートメントが展開されます。NOINONLY が指定されると、SQL プリプロセッサによってすべてのステートメントが処理され、コードが生成されます。INONLY と NOINONLY は相互に排他的で、互換性のために NOINONLY がデフォルトになっています。

INSERT

DB2/6000 Parallel Edition サーバー上のパフォーマンスを向上させるため、挿入データをバッファに入れることを要求します。

DEF VALUES による標準 INSERT 実行を使用します。これが、デフォルト設定です。

BUF VALUES による INSERT 実行時にバッファリングを使用します。

注: このオプションは、DB2 Parallel Edition サーバーに対するプリコンパイル時に限り使用できます。INSERT を DB2 V1.x サーバーに対して使用すると、INSERT は無視され、警告メッセージが発行されます。INSERT を DB2 V2.x サーバーに対して使用すると、INSERT は無視され、警告メッセージが発行され、そのオプションがバインド・ファイルに追加されます。

ISOLATION

このパッケージにバインドされているプログラムが他の実行プログラムの影響からどこまで分離できるかを決定します。

CS 分離レベルとして、カーソル固定を指定します。

RR 分離レベルとして、反復可能読み取りを指定します。

RS 分離レベルとして、読み取り固定を指定します。読み取り固定では、パッケージ内の SQL ステートメントの実行が、アプリケーションによって読み取りおよび変更される行について、他のアプリケーション・プロセスから確実に分離されます。

UR 分離レベルとして、非コミット読み取りを指定します。

ONEPASS または TWOPASS

ONEPASS がデフォルトであり、ホスト変数は使用前に宣言する必要があることを示します。TWOPASS を使用すると、使用前にホスト変数を宣言する必要のないことが指示されます。

OPTIMIZE または NOOPTIMIZE

OPTIMIZE を指定すると、SQLDA 初期設定は、ホスト変数を使用する SQL ステートメント用に最適化されます。ホスト変数のアドレスがプログラム実行中に変化する可能性のある場合、または AUTOMATIC ホスト変数を使用する場合は、このオプションを指定しないでください。(NOOPTIMIZE) がデフォルトです。

OWNER

パッケージ所有者の 30 文字の **authorization-id** を指定します。所有者は、パッケージに含まれる SQL ステートメントの実行に必要な特権を持っている必要があります。SYSADM または DBADM 権限を持つユーザーのみが、ユーザー ID 以外の **authorization-id** を指定できます。デフォルト値は、プリコンパイル/バインド処理の基本許可 ID です。SYSIBM、SYSCAT および SYSSTAT は、このオプションには無効な値です。

PLAN、NOPLAN、または NOPLANSYNTAX

アクセス・プラン *planname* を作成するかどうかを決定します。プラン名は、指定しない場合、入力ソース・ファイルの名前にデフォルト設定されます。

NOPLANSYNTAX を指定すると、アクセス・プランは作成されず、構文検査は、DB2 バージョン 2.1 構文に基づいて実行されます。

PRINT または NOPRINT

SQL プリプロセッサによって生成されるソース・コードを、後続のプリプロセッサまたはコンパイラの生成するソース・リスト (1 つ以上) 内に印刷するかどうかを指定します。

QUALIFIER

パッケージに含まれる非修飾オブジェクトに 30 文字の暗黙修飾子名を提供します。所有者が明示的に指定されているかどうかに関係なく、デフォルトは所有者の許可 ID です。

SCOPE または NOSCOPE

宣言の有効範囲に対する PL/I 規則が、ホスト変数参照を解決するときに適用されるかどうかを決定します。

SCOPE が指定されると、ホスト変数参照を解決するときに、宣言の有効範囲に対する PL/I 規則が適用されます。つまり、SQL ステートメント内のホスト変数参照は常に、その変数が別の PL/I ステートメントで使用されている場合と同じように解決されます。

NOSCOPE が指定されると、すべてのホスト変数の名前は、各プログラム内で固有である必要があります。ホスト変数名が複数回宣言されると、SQL ステートメント内のその変数への参照は、最初の宣言を使用して解決されます。

前のリリースでの動作との互換性のために、NOSCOPE がデフォルトになっています。

SOURCE または NOSOURCE

SQL プリプロセッサへのソース入力を印刷するかどうかを指定します。

SQLFLAG

このオプションで指定された SQL 言語構文からの逸脱を特定し、報告します。このオプションを指定しない場合、flagger 機能は呼び出されません。詳細な説明は、DB2 の資料を参照してください。

MVSDB2V23

SQL ステートメントが、MVS DB2 V2.3 SQL 言語構文に基づいて検査されます。これが、デフォルト設定です。

MVSDB2V31

SQL ステートメントが、MVS DB2 V3.1 SQL 言語構文に基づいて検査されます。

MVSDB2V41

SQL ステートメントが、MVS DB2 V4.1 SQL 言語構文に基づいて検査されます。

SQLRULES

DB2 規則または ISO/ANS SQL92 に基づく標準 (STD) 規則のいずれかに従って、タイプ 2 CONNECT を処理するかを指定します。

DB2

SQL CONNECT ステートメントを使用して、現行接続から別の確立されている (休止状態) の接続への切り替えを可能にします。これが、デフォルト設定です。

STD

SQL CONNECT ステートメントを使用して新規の接続に限りその確立を行うことを可能にします。休止接続に切り替えるには、SQL SET CONNECTION を使用する必要があります。

デフォルト・オプションの値は、SQLRULES(DB2) です。SQLR、SQLRULES、SQLR()、SQLRULES() などのオプションのストリングは、SQLRULES(DB2) と判断されます。

SYNCPOINT

複数のデータベース接続の間でコミットまたはロールバックを調整する方法を指定します。

ONEPHASE

このオプションを指定すると、トランザクション・マネージャー (TM) を使用する 2 フェーズ・コミットを実行しません。1 フェーズ・コミットを使用して、複数のデータベース・トランザクションで、各データベースが行った処理をコミットします。これが、デフォルト設定です。

TWOPHASE

これを指定すると、このプロトコルをサポートするデータベース間の 2 フェーズ・コミットを調整するために TM が必要となります。

NONE

これを指定すると、TM を使用する 2 フェーズ・コミットを実行されず、TM は、単一の更新プログラム、複数の読み取りプログラムを実行しません。COMMIT が、各参加しているデータベースに送信されます。どのコミットが失敗しても、回復の責任は、アプリケーション側にあります。

デフォルト・オプションの値は、SYNCPOINT(ONEPHASE) です。SYNC、SYNCPOINT、SYNC()、SYNCPOINT() などのオプションのストリングは、SYNCPOINT(ONEPHASE) と判断されます。

TBQUAL

パッケージに含まれる非修飾オブジェクトに 8 文字の暗黙テーブル修飾子名を提供します。

VALIDATE

許可エラーおよび「オブジェクトが見つからないエラー」をデータベース・マネージャーがいつチェックするかを決定します。パッケージ所有者の許可 ID が、妥当性検査に使用されます。

BIND

妥当性検査はプリコンパイル/バインド時に実行されます。すべてのオブジェクトが存在しない場合、またはすべての権限が保持されていない場合は、エラー・メッセージが生成されます。 **sqlerror continue** を指定した場合は、エラー・メッセージに関係なく、パッケージ/バインド・ファイルが生成されますが、エラーのあるステートメントは実行できません。

RUN

妥当性検査はバインド時に実行されます。すべてのオブジェクトが存在し、すべての権限が保持されている場合、実行時にそれ以上の検査は行われません。すべてのオブジェクトが存在しない場合、またはすべての権限がプリコンパイル/バインド時に保持されていない場合は、警告メッセージが生成され、**sqlerror continue** オプション設定に関係なく、パッケージは正常にバインドされます。ただし、プリコンパイル/バインド処理中に権限検査および存在検査をパスしなかった SQL ステートメントに対しては、実行時にこれらの検査を再実行できます。

VERSION

パッケージのバージョン ID を定義します。このオプションを指定しない場合、パッケージ・バージョンは "" (空ストリング) となります。

version-id

任意の英数字値、\$、#、@、_、-、または . (文字長は最大 64 文字) を使用して、バージョン ID を指定します。

AUTO

バージョン ID が整合性トークンから生成されます。整合性トークンがタイム・スタンプである場合 (LEVEL オプションを指定しない場合)、タイム・スタンプは 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。

SQL プリプロセッサ・オプション環境変数

IBM.PPSQL 環境変数を使用して SQL プリプロセッサのデフォルト・オプションを設定することができます。 31 ページの『IBM.PPSQL』を参照してください。

SQL プリプロセッサ BIND 環境変数

BIND オプションを指定した場合は、SQL プリプロセッサによって、コンパイルするプログラムの現行ディレクトリーに、バインド・ファイルが作成されます。出力ファイルの格納先は、例えば、以下のように、IBM.BIND 環境変数を設定して変更できます。

```
set ibm.bind=C:%bindlib
```

SQL バインド出力ファイルは、別名を指定しない限り、1 次入力ファイルと同名で、拡張子は BND となります。

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

PL/I アプリケーションでの SQL ステートメントは、「*SQL Reference, Volume 1 and Volume 2*」(SBOF-8923) に定義されている言語を使用して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、連絡域 (SQLCA) を組み込む必要があります。 116 ページの図 1 に示すように、SQLCA の一部は、SQLCODE 変数と SQLSTATE 変数から構成されています。

- SQLCODE の値は、各 SQL ステートメントの実行後にデータベース・マネージャー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・マネージャーによって設定されます。

SQLCA をインクルードするには、次のように SQL INCLUDE ステートメントを使用する必要があります。

SQL プリプロセッサ

```
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 の宣言の有効範囲には、プログラム内の SQL ステートメントの有効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 sqlerrm      char(70) var,     /* Error message tokens */
2 sqlerrp      char(8),          /* Diagnostic information */
2 sqlerrrd(6)   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 sqlnext,
3 sqlwarn8     char(1),
3 sqlwarn9     char(1),
3 sqlwarna     char(1),
3 sqlstate     char(5);         /* State corresponding to SQLCODE */
```

図1. 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 SQL INCLUDE ステートメントを使用する必要があります。

```
exec sql include sqllda;
```

SQLDA は、SQL 宣言セクション内で定義してはなりません。

```

Dcl
1 Sqllda 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 sqlld 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 Sqlsize fixed binary(15); /* number of sqlvars (sqln) */
dcl Sqldaptr pointer;

```

図2. SQL 記述子域の PL/I 宣言

SQL ステートメントの組み込み

PL/I 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 Department
export 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 PL/I コメントを組み込むことができます。

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、 m と n が MARGINS(m,n) コンパイル時オプションで指定される場所で、列 m から n にコーディングする必要があります。

名前: ホスト変数に対しては任意の有効な PL/I 変数名を使用できますが、制限として、「SQL」、「DSN」、または「IBM」で始まるホスト変数名、外部入り口名、またはアクセス計画名は使用しないでください。これらの名前は、データベース・マネージャーおよび PL/I 用に予約済みです。ホスト変数名の長さは、100 文字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ず、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有効範囲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 が有効な場合は、SQL ステートメント内で使用するホスト変数は、SQL ステートメント内のホスト変数が最初に使用される前に宣言する必要があります。さらに、次の制限があります。

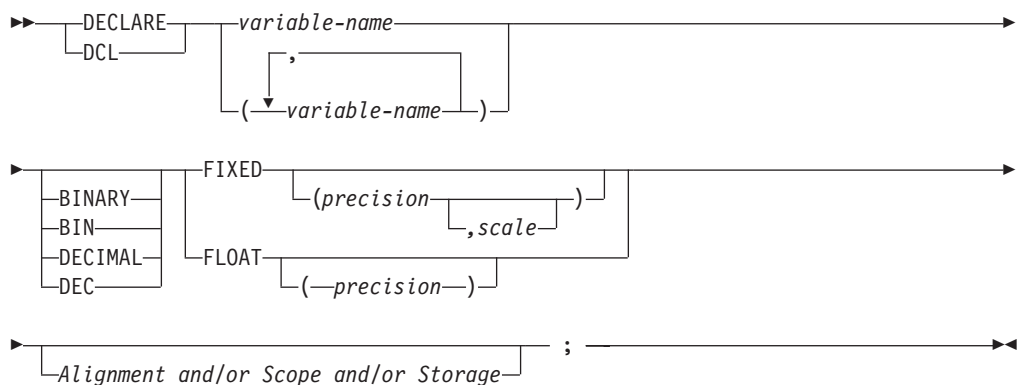
- SQL ステートメント内では、すべてのホスト変数の前にコロン (:) を付ける必要があります。
- ホスト変数を使用する SQL ステートメントは、変数の宣言を行ったステートメントの有効範囲内になければなりません。
- ホスト変数を配列として宣言することはできません。ただし、配列がホスト構造体に関連付けられている場合に、標識変数の配列を使用することは可能です。

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、「The host variable token ID is not valid」というメッセージが出されることがあります。

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、有効範囲、およびストレージの属性は無視されます。

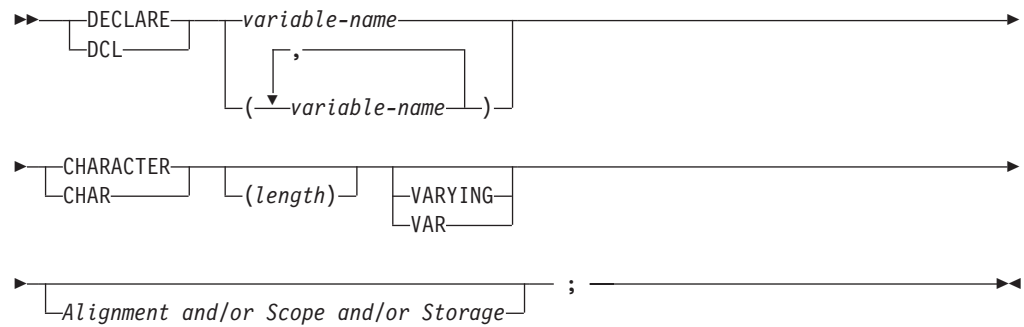
数値ホスト変数: 次の図は、有効な数値ホスト変数宣言の構文を示しています。



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ができます。
- *scale* の値は、DECIMAL FIXED の場合だけ指定できます。

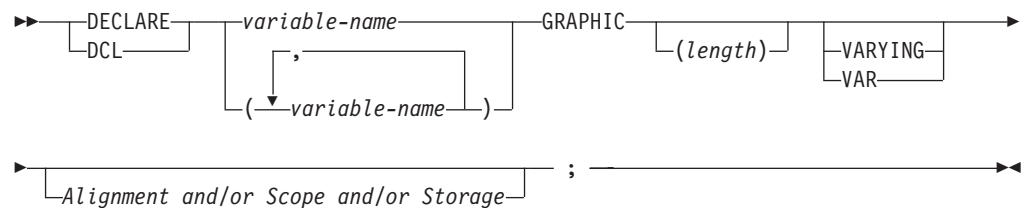
文字ホスト変数: 次の図は、有効な文字ホスト変数の構文を示しています。



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。



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。

SQL および PL/I の同等なデータ型の判別

ホスト変数の基本 SQLTYPE および SQLLEN は、次の表のとおり決定されます。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。

表 4. PL/I 宣言から生成される SQL データ型

PL/I データ型	ホスト変数の SQLTYPE	ホスト変数の SQLLEN	SQL データ型
BIN FIXED(n), n < 16	500	2	SMALLINT
BIN FIXED(n), n の範囲は 16 から 31	496	4	INTEGER
DEC FIXED(p,s)	484	p (バイト 1) s (バイト 2)	DECIMAL(p,s)
BIN FLOAT(p), 22 ≤ p ≤ 53	480	8	FLOAT

表 4. PL/I 宣言から生成される SQL データ型 (続き)

PL/I データ型	ホスト変数の SQLTYPE	ホスト変数の SQLLEN	SQL データ型
DEC FLOAT(m), $7 \leq m \leq 16$	480	8	FLOAT
CHAR(n), $1 \leq n \leq 254$	452	n	CHAR(n)
CHAR(n) VARYING, $1 \leq n \leq 4000$	448	n	VARCHAR(n)
CHAR(n) VARYING, $n > 4000$	456	n	LONG VARCHAR
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

SQL には、単一精度または拡張精度の浮動小数点データ型が存在しないため、データ挿入に、同データ型のホスト変数を使用すると、ホスト変数は、倍精度浮動小数点の一時値に変換され、その一時値がデータベースに挿入されます。単一精度または拡張精度の浮動小数点ホスト変数を使用してデータ検索を行うと、倍精度浮動小数点の一時値が、データベースに対するデータ検索に使用され、一時変数の結果がホスト変数に代入されます。

次の表を使用して、特定の SQL データ型と同等な PL/I データ型を判別できます。

表 5. SQL データ型と PL/I 宣言の対応

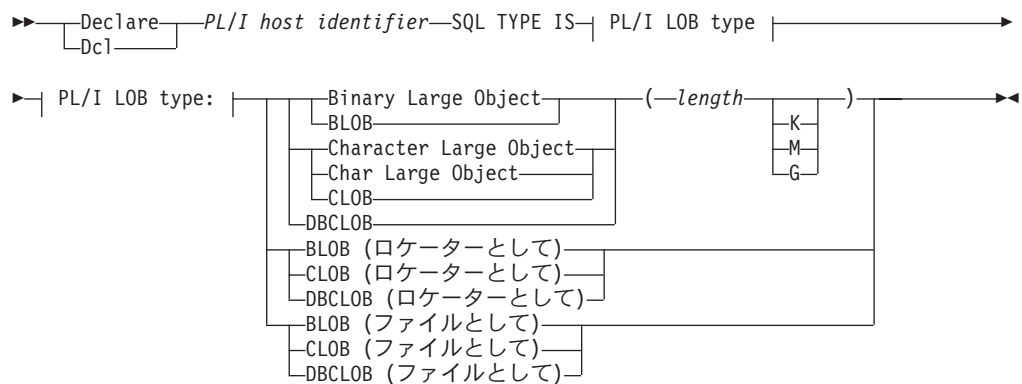
SQL データ型	同等な PL/I 宣言	注
SMALLINT	BIN FIXED(15)	
INTEGER	BIN FIXED(31)	
DECIMAL(p,s)	DEC FIXED(p) または DEC FIXED(p,s)	p = precision および s = scale; $1 \leq p \leq 31$ および $0 \leq s \leq p$
FLOAT	BIN FLOAT(p) または DEC FLOAT(m)	$22 \leq p \leq 53$ $7 \leq m \leq 16$
CHAR(n)	CHAR(n)	$1 \leq n \leq 254$
VARCHAR(n)	CHAR(n) VAR	$1 \leq n \leq 4000$
LONG VARCHAR	CHAR(n) VAR	$n > 4000$
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 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

ラージ・オブジェクト (LOB) サポート

バイナリー・ラージ・オブジェクト (BLOB)、文字ラージ・オブジェクト (CLOB)、および 2 バイト文字ラージ・オブジェクト (DBCLOB) が、LOB ロケータと LOB ファイルの概念とともに、現在では、プリプロセッサによって認識されます。これらの詳細については、DB2 マニュアルを参照してください。

LOB に関する一般情報

LOB、CLOB、および BLOB は、最大で 2,147,483,640 バイト (2 ギガバイトから PL/I オーバーヘッドの 8 バイトを引いた値) の長さにすることができます。2 バイト CLOB は、1,073,741,820 文字 (1 ギガバイトから PL/I オーバーヘッドの 4 文字を引いた値) の長さにすることができます。BLOB、CLOB、および DBCLOB は、以下の構文 (ラージ・オブジェクト列、ロケータ、およびファイルに対する PL/I 変数) を使用して、PL/I プログラム内で宣言することができます。



BLOB、CLOB、および DBCLOB データ型

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、この宣言を次の構造体に変換します。

```
Define structure
1 lob-type-name_length,
2 Length unsigned fixed bin(31),
2 Data(length) char(1);
Dcl my-identifier-name TYPE lob-type-name_length;
```

この構造体の中で、my-identifier-name は PL/I ホスト ID の名前、lob-type-name_length は LOB のタイプと長さからなる、プリプロセッサが生成した名前です。

DBCLOB データ型の場合、生成される構造体は次のように少し異なります。

```
Define structure
1 lob-type-name_length,
2 Length unsigned fixed bin(31),
2 Data(length) type wchar_t;
```

この場合、type `wchar_t` は、インクルード・メンバー `sqlsystem.cpy` 内で定義されます。このメンバーは、DBCLOB データ型を使用するためにはインクルードする必要があります。

length

長さフィールドは、固定長 2 進数にマッピングされる符号なし整数です。長さフィールドの値は、0 から $(2^{32})-8$ の範囲になります。長さフィールドに K、M、または G が追加されている場合は、長さは、プリプロセッサによって計算されます。

BLOB、CLOB、および DBCLOB LOCATOR データ型

BLOB、CLOB、および DBCLOB ロケーターの変数宣言も、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 AS LOCATOR;
```

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) unsigned;
```

```
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。

BLOB、CLOB、および DBCLOB FILE データ型

BLOB、CLOB、および DBCLOB ファイルの変数宣言も、PL/I SQL プリプロセッサによって変換されます。

例えば、以下の宣言文を考えてみます。

```
Dcl my-identifier-name SQL TYPE IS lob-type AS FILE;
```

SQL プリプロセッサは、以下のように宣言文を変換します。

```
Define structure
1 lob-type_FILE,
2 Name_Length unsigned fixed bin(31),
2 Data_Length unsigned fixed bin(31),
2 File_Options unsigned fixed bin(31),
2 Name char(255);
```

```
Dcl my-identifier-name TYPE lob-type_FILE;
```

この場合も、`my-identifier-name` は PL/I ホスト ID、`lob-type_FILE` は LOB のタイプとストリング FILE 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。

例 1:

```
Dcl my_blob SQL TYPE IS blob(2000);
```

変換後:

```

Define structure
1  blob_2000,
2  Length unsigned fixed bin(31),
2  Data(2000) char(1);
Dcl my_blob type blob_2000;

```

例 2:

```
Dcl my_dbclob SQL TYPE IS DBCLOB(1M);
```

変換後:

```

Define structure
1  dbclob_1m,
2  Length unsigned fixed bin(31),
2  Data(1048576) type wchar_t;
Dcl my_dbclob type dbclob_1m ;

```

例 3:

```
Dcl my_clob_locator SQL TYPE IS clob as locator;
```

変換後:

```

Define alias clob_locator fixed bin(31) unsigned;
Dcl my_clob_locator type clob_locator;

```

例 4:

```
Dcl my_blob_file SQL TYPE IS blob as file;
```

変換後:

```

Define structure
1  blob_FILE,
2  Name_Length unsigned fixed bin(31),
2  Data_Length unsigned fixed bin(31),
2  File_Options unsigned fixed bin(31),
2  Name char(255);

Dcl my_blob_file type blob_file;

```

例 5:

```
Dcl my_dbclob_file SQL TYPE IS dbclob as file;
```

変換後:

```

Define structure
1  dbclob_FILE,
2  Name_Length unsigned fixed bin(31),
2  Data_Length unsigned fixed bin(31),
2  File_Options unsigned fixed bin(31),
2  Name char(255);

Dcl my_dbclob_file type dbclob_file;

```

LOB サポートのためのサンプル・プログラム

LOB 型の PL/I プログラムにおける使用法を示すため、以下の 3 つのサンプル・プログラムが用意されています。

SQLLOB1.PLI

データベースからファイルへの BLOB の取り込み方法を示します。

SQLLOB2A.PLI

LOB 式の最終代入まで、バイトの移動を行わないで LOCATOR 変数を使用して、LOB を変更する方法を示します。

SQLLOB2B.PLI

SQLLOB2A.PLI に作成されている CLOB を表示用ファイルに取り出します。

ユーザー定義関数のサンプル・プログラム

ユーザー定義関数 (UDF) サンプル・プログラムにアクセスするには、以下の項目をインストールしておく必要があります。

- DB2 V2.1 以上
- サンプル・データベース

複数の PL/I プログラムが、UDF をコーディングし、使用方法を示すために用意されています。以下は、それらのプログラムの使用方法を簡単に説明するものです。

ファイル UDFDLL.PLI には、5 つのサンプル UDF が格納されています。サンプルは、本質的に単純なものですが、UDF の基本概念を示しています。

MyAdd

2 つの整数を加算して、第 3 の整数に結果を戻します。

MyDiv 2 つの整数を除算して、第 3 の整数に結果を戻します。

MyUpper

小文字の a、e、i、o、u をすべて大文字に変更します。

MyCount

スクラッチパッドを使用したカウンター機能を簡単に実現したものです。

ClobUpper

CLOB 内の小文字の a、e、i、o、u をすべて大文字に変更した後、ファイルに書き込みます。

コマンド・ファイル bldudfdll を使用して、コンパイルして、udfdll ライブラリーにリンクします。

udfdll ライブラリーのコンパイルとリンクが完了したら、使用中のデータベース・インスタンスのユーザー定義関数ディレクトリーにコピーします。例えば、PL/I for AIX を使用していて、/u/inst1/sqllib/function が、AIX マシン上のデータベース・インスタンスに対するユーザー定義関数ディレクトリーである場合は、udfdll をそこにコピーします。

ユーザー定義関数は、使用する前に、DB2 に対して定義しておく必要があります。これは、CREATE FUNCTION コマンドを使用して行います。サンプル・プログラム addudf.pli が、各 UDF に対する CREATE FUNCTION コールを実行するために用意されています。CREATE FUNCTION コールは、例えば、以下のような形式をしています。

```
CREATE FUNCTION MyAdd ( INT, INT ) RETURNS INT NO SQL
LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
STYLE DB2SQL EXTERNAL NAME 'udfdll!MyAdd'

CREATE FUNCTION MyDiv ( INT, INT ) RETURNS INT NO SQL
```

```

LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
STYLE DB2SQL EXTERNAL NAME 'udfd11!MyDiv'

CREATE FUNCTION MyUpper ( VARCHAR(61) ) RETURNS VARCHAR(61) NO SQL
LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
STYLE DB2SQL EXTERNAL NAME 'udfd11!MyUpper'

CREATE FUNCTION MyCount ( ) RETURNS INT NO SQL
LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
STYLE DB2SQL EXTERNAL NAME 'udfd11!MyCount'
SCRATCHPAD

CREATE FUNCTION ClobUpper ( CLOB(5K) ) RETURNS CLOB(5K) NO SQL
LANGUAGE C FENCED VARIANT NO EXTERNAL ACTION PARAMETER
STYLE DB2SQL EXTERNAL NAME 'udfd11!ClobUpper'

```

上記は、CREATE FUNCTION コマンドのサンプルにすぎません。詳細情報または改良については、DB2 マニュアルを参照してください。

コマンド・ファイル bldaddudf を使用して、addudf.pli プログラムをコンパイル、リンクします。コンパイル、リンクが完了した後、実行して、ユーザー定義関数をデータベースに対して定義します。

上記で作成し、データベースに追加の完了したユーザー定義関数を呼び出す場合に使用できるサンプル PL/I プログラムが複数用意されています。

UDFMYADD.PLI

STAFF テーブルから ID と Dept を取り出した後、MyAdd UDF を呼び出して、それらを一緒に追加します。コマンド・ファイル bldmyadd を使用して、このプログラムをコンパイル、リンクします。

UDFMYDIV.PLI

STAFF テーブルから ID と Dept を取り出した後、MyDiv UDF を呼び出して、それらを除算します。コマンド・ファイル bldmydiv を使用して、このプログラムをコンパイル、リンクします。

UDFMYUP.PLI

STAFF テーブルから Name を取り出した後、MyUpper を呼び出し、母音字を大文字に変更します。コマンド・ファイル bldmyup を使用して、このプログラムをコンパイル、リンクします。

UDFMYCNT.PLI

STAFF テーブルから ID を取り出し、コール数を出力した後、ID をコール数で除算します。コマンド・ファイル bldmycnt を使用して、このプログラムをコンパイル、リンクします。

UDFCLOB.PLI

従業員 '000150' の経歴を取り出した後、ClobUpper を呼び出し、母音文字を大文字に変更します。コマンド・ファイル bldclobu を使用して、このプログラムをコンパイルおよびリンクします。プログラムの実行後、ファイル udfclob.txt を調べ、結果をチェックします。

上記の PL/I サンプル・プログラムは、そのコンパイル、リンクが終了し、DB2 に対する UDF の定義も終了すると、コマンド行から実行することが可能になります。

それらの UDF は、他の DB2 組み込み関数とまったく同様に、DB2 コマンド行からも呼び出すことができます。UDF のカスタマイズと活用方法の詳細については、DB2 マニュアルを参照してください。

SQL データ型と PL/I データ型の互換性の判別

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。

- 数値データ型は、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)、または DEC FLOAT (*m*) (ただし *m* は 7 から 16) の PL/I ホスト変数と互換性があります。
- 文字データ型は、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。

グラフィック・データ型は、相互に互換性があります。GRAPHIC または VARGRAPHIC の列は、固定長または可変長の PL/I グラフィック文字ホスト変数と互換性があります。

- Datetime データ型は、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もつ構造体の名前を、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。

```
dc1 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。

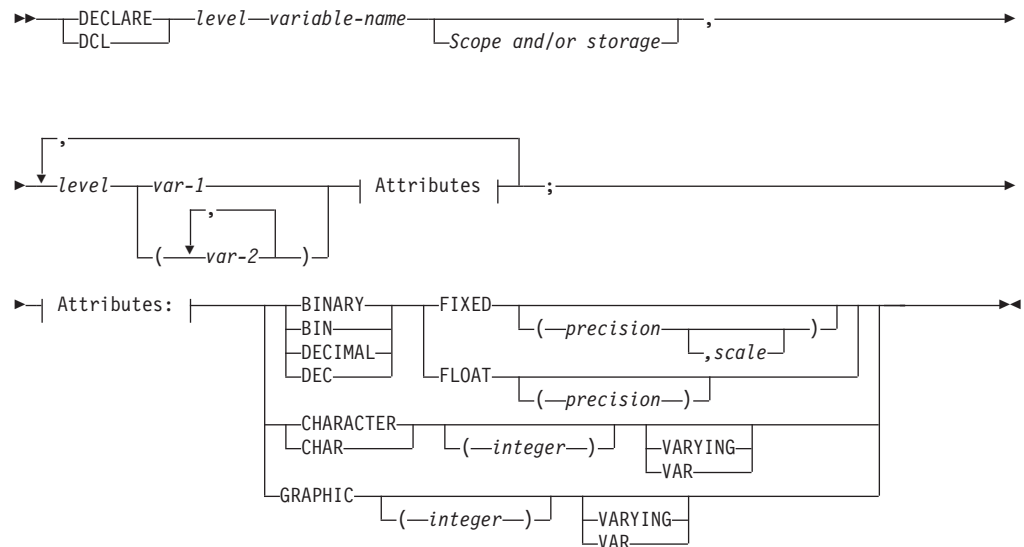
ホスト構造体は 2 レベルに制限されます。ホスト構造体は、ホスト変数の名前付き集合と考えることができます。

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。

```
dc1 1 A,  
    2 B char,  
    2 (C, D) char;  
dc1 (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。

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;
```

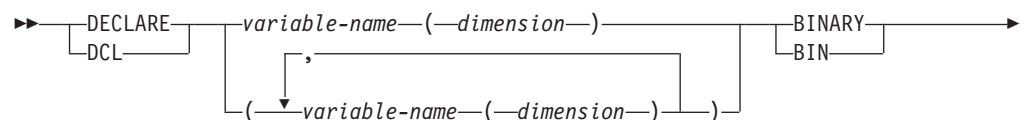
変数は次のように宣言できます。

```
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;
```

CONNECT TO ステートメント

アプリケーションの接続先のデータベースは、例えば、以下のように、ホスト変数を使用して表すことができます。

```
exec sql connect to :dbase;
```

ホスト変数を指定する場合は、次のことに注意します。

- 1 つの文字または 1 つの可変長文字変数にする必要があります。
- その前にコロンを置きますが、後に、標識変数が来ないようにする必要があります。
- ホスト変数内に含めるサーバー名は、左寄せにする必要があります。
- サーバー名の長さが、固定長ホスト文字変数の長さより短い場合、右側にブランクを埋め込む必要があります。

```
dcl dbase char (10);
dbase = 'SAMPLE';          /* blanks are padded automatically */
exec sql connect to :dbase;
```

- 可変長ホスト文字変数を使用すると、コンパイラから以下の警告を受け取る場合があります。このメッセージは無視して構いません。

```
IBM1214I W   xxx.x   A dummy argument is created for argument
                  number 6 in entry reference SQLESTRD_API
```

DECLARE TABLE ステートメント

プリプロセッサは、DECLARE TABLE ステートメントをすべて無視します。

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。

論理 NOT 記号 (¬)

プリプロセッサは、SQL ステートメント内で以下の変換を行います。

- ¬= は <> に変換されます
- ¬< は >= に変換されます
- ¬> は <= に変換されます

SQL エラー戻りコードの処理

PL/I には、表示用の複数行メッセージに SQLCODE を変換する場合に使用できるサンプル・プログラム DSNTIAR.PLI が用意されています。この PL/I プログラムには、メインフレーム DB2* の DSNTIAR プログラムと同じ機能が備わっています。

DSNTIAR をコンパイルする場合は、DSNTIAR を使用するプログラムをコンパイルするために用いる DEFAULT と SYSTEM コンパイル時オプションと同じものを使用する必要があります。

- Windows PL/I プログラム内で、DSNTIAR を使用する場合は、DSNTIAR を以下のコンパイル時オプションを使用してコンパイルする必要があります。
 - DEFAULT(ASCII NATIVE LINKAGE(OPTLINK))
 - SYSTEM(WINDOWS) (Windows をしている場合)
- ホスト・エミュレーション PL/I プログラム内で、DSNTIAR を使用する場合は、DSNTIAR を DEFAULT(EBCDIC NONNATIVE LINKAGE(SYSTEM)) と SYSTEM(MVS) の両コンパイル時オプションを使用してコンパイルする必要があります。

呼び出し側が、メインフレーム DB2 資料の説明通りに、エントリーを宣言し、インターフェースに従う必要があります。ご参考までに、宣言文は、以下のような形式をとります。

```
dcl dsntiar entry options(asm inter retcode);
```

3 つの引数を常に引き渡します。

arg 1

この入力引数は、必ず SQLCA とします。

arg 2

この入出力引数は、以下の形式の構造体とします。

```
dcl 1 Message,
      2 Buffer_length fixed bin(15) init(n), /* input */
      2 User_buffer char(n);                /* output */
```

n には、該当する値を入力します。

arg 3

この入力引数は、論理レコード長を指定する FIXED BIN(31) 値とします。

DFT(EBCDIC NONNATIVE) のもとでの可変長ストリングの使用

コンパイル時オプション DFT(EBCDIC NONNATIVE) を指定し、データベースへの入力として、可変長ストリング・ホスト変数を使用する場合は、ホスト変数を初期設定する必要がある、その初期設定を怠ると、プログラムの実行時に、記憶保護例外が発生する可能性があります。

メインフレーム DB2 で、未初期設定の可変長ストリングを使用すると、プログラムは、エラー状態になり、記憶保護例外も引き起こす場合があります。

DEFAULT(EBCDIC) コンパイル時オプションの使用

入力/出力文字ホスト変数を含む SQL ステートメントでコンパイル時オプション DEFAULT(EBCDIC) を使用すると、SQL プリプロセッサは、文字データが、FOR BIT DATA 列属性を持っていない場合に限り、追加のコードを SQL ステートメントの拡張部に挿入し、ASCII と EBCDIC の間で文字データを変換します。

特定の文字データの自動変換の回避: データを変換しない場合は、プリプロセッサに対して明示的にコマンドを指定する必要があります。例えば、CHARACTER 変数と FOR BIT DATA 列間の変換を行う必要のない場合は、以下の例に示すように PL/I コメントを挿入することもできます。

```
dc1 SL1 /* %ATTR FOR BIT DATA */ char(9);
```

コメント内の最初の非ブランク文字は、パーセント (%) 記号にし、その後に、キーワード ATTR FOR BIT DATA を続ける必要があります。

このコメントは、変数の宣言文の最後までに存在する限り、変数名の後の任意の箇所に置くことができます。以下の例では、SL2 と SL4 は、いずれも変換されません。

```
Dc1 SL2 /* %ATTR FOR BIT DATA */ char(9),  
    SL3 char (20); /* %ATTR FOR BIT DATA */  
Dc1 (SL4 /* %ATTR FOR BIT DATA */,  
    SL5) char (9);
```

DCLGEN を使用する自動変換の回避: DEFAULT(EBCDIC) を使用したことで行われる変換を回避する別の方法は、DCLGEN ユーティリティを使用することです。DCLGEN ユーティリティは、データベース・テーブルに対する宣言文を作成するため、PL/I for Windows に添付されています。

DCLGEN は、列が FOR BIT DATA 属性で定義されていると認識すると、必要なコメント・ディレクティブを出力内に自動的に生成します。

DEFAULT(NONNATIVE) コンパイル時オプションの使用: 10 進数フィールドを記述する SQLDA とともにコンパイル時オプション DEFAULT(NONNATIVE) を使用する場合は、SQL プリプロセッサによる変換の終了後に、SQLLEN フィールドを再反転する必要があります。

SQL 互換性と移行に関する考慮事項

ワークステーション・コンパイラは、以下のステートメントを容認します。

```
' EXEC SQL CONNECT :userid IDENTIFIED BY :passwd'
```

上記のステートメントは、PL/I SQL プリプロセッサによって変換された後、以下のステートメントとしてデータベース・プリコンパイラ・サービスに送信されます。

```
' EXEC SQL CONNECT'
```

これにより、VM SQL/DS ユーザーは、大幅な変更を行わずに、プログラムをコンパイルすることが可能になります。

CICS サポート

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 プリプロセッサを呼び出さない場合、実際のコードは生成できません。

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。

特定の開発プラットフォーム上の CICS のもとでの最終実行、または S/390 上の CICS/ESA、CICS/MVS、CICS/VSE システムのもとでの最終実行を目的として、これらのアプリケーションは、Windows 上の CICS のもとで開発することができます。

インストールされている CICS が、Windows サポート用のシステム環境変数に、すべての ¥OPT¥... 設定を追加していること確認します。プログラムのコンパイル時に、CICS システムが作動可能である必要はありません。

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、以下のオプションを使用する必要があります。

- SYSTEM(CICS) コンパイル時オプション。
- PP(CICS(*options*) MACRO) コンパイル時オプション。MACRO オプションは、PP の CICS オプションに続きます。

CICS プログラムが、EXEC CICS ステートメントを格納するファイルをインクルードするか、EXEC CICS ステートメントを含むマクロを使用する場合は、PP オプションの CICS オプションの前に、以下の例に示すように、MACRO コンパイル時オプションまたは PP の MACRO オプションのいずれかを使用することも必要になります。

```
pp (macro(...) cics(...) macro(...))
```

cicsdb2.pli という名前の CICS および DB2 PL/I プログラムを編集する場合は、以下のコマンドを使用します。

```
pli -l/usr/lpp/cics/include
-qsystem=CICS
-qpp=CICS=noedf:nodebug:nosource:noprint:MACRO
-o cicsdb2.ibmpli
-bl:/usr/lpp/cics/lib/cicsprIBMPI.exp
-eplicics
-L/usr/lib/dce
-ldcelibc_r
-ldcephthreads
-ldb2
-lplishr_r
-lc_r
cicsdb2.pli
```

INCLUDE(EXT) コンパイル時オプションに対する拡張子として、INC が指定されていることを確認します。 58 ページの『INCLUDE』を参照してください。

IBM.SYSLIB 環境変数または INCLUDE 環境変数には、例えば以下のように、CICS インクルード・ファイル・ディレクトリーを指定する必要があります。

```
set include=d:¥cicsnnn¥plihdr;
```

基本マッピング・サポート (BMS) ユーティリティーである CICS MAP によって生成された PL/I 宣言文は、INCLUDE 環境変数で指定されている最初のディレクトリーに置かれます。詳しくは、30 ページの『コンパイル時環境変数の設定』を参照してください。

以下のいずれかの方法で生成される出力は、CPLI 一時データ・キュー (TDQ) に書き込まれます。

- SYSPRINT に対する PUT ステートメント
- MSGFILE に書き込まれるメッセージ
- DISPLAY ステートメント

PLIDUMP によって生成される出力は、常時、CPLD 一時データ・キューに書き込まれます。

フル・ワークステーション CICS API が、PL/I プログラムに対してサポートされています。以下を使用する PL/I プログラムに対するサポートも提供されています。

- 外部表示インターフェース (EPI)
- 外部呼び出しインターフェース (ECI)
- 外部トランザクション起動 (ETI)

S/390 CICS に適用される PL/I 考慮事項は、ワークステーション上の CICS にも適用されます。プログラムは、STAE オプションが常時有効であるかのように動作します。NOSTAE オプションはサポートされません。

S/390 CICS サブシステム上での最終実行を目的として、アプリケーションの開発を行っている場合は、DEFAULT(NONASSIGNABLE) コンパイル時オプションを使用して、PL/I プログラムの再入可能違反を検査することができます。

CICS/ESA、CICS/MVS、および CICS/VSE との互換性のため、EXEC CICS コマンドは、必ず、大文字にしてください。

CICS のもとでは、PL/I FETCH と RELEASE を使用できます。

1 つの CICS プログラムが、OPTIONS(MAIN) を持つプロシージャーを複数持つことはできません。

EXEC CICS ADDRESS コマンド、および CICS 制御ブロックに対するポインターを戻す他の類似コマンド (TWA COMMAREA や ACEE など) は、制御ブロックが存在しない場合に SYSNULL() ポインターを戻します。(例えば、'FF000000'x ではなく、'00000000'x)。プログラムでは、そのようなポインターをテストするため、SYSNULL 組み込み関数を使用する必要があります。

CICS プリプロセッサによって処理される各 PL/I コンパイル単位は、以下を生成します。

```
dc1 IBM CICS_ID char(n) static init('cics-id-and-version');
```

プログラムがコンパイルされた対象の CICS システムの名前、バージョン、およびリリース・レベルが示されます。

プログラムの性質、およびプログラムの実行に使用される CICS システムに従って、オプションを考慮する必要もあります。

表 6. EXEC CICS サポートに対する考慮事項

使用する対象	使用するコンパイル時オプション
CICS for Windows	PP(CICS MACRO)
ネイティブ・データを格納する CICS ファイル	場合に応じて DEFAULT (ASCII NATIVE IEEE)
ネイティブ・モードの DB2/2	場合に応じて DEFAULT (ASCII NATIVE IEEE)
ホスト S/390 データを格納する CICS ファイル	場合に応じて DEFAULT (EBCDIC NONNATIVE HEXADEC)
ホスト S/390 モードの DB2/2	場合に応じて DEFAULT (EBCDIC NONNATIVE HEXADEC)

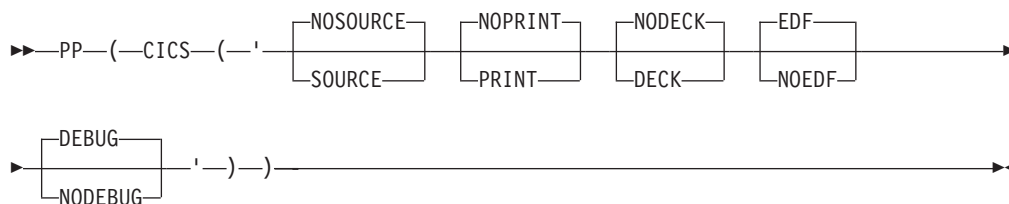
表 7. EXEC CICS サポートに対する考慮事項

使用する対象	使用するコンパイル時オプション
CICS for Windows	PP(CICS MACRO)
ネイティブ・データを格納する CICS ファイル	場合に応じて DEFAULT (ASCII NATIVE IEEE)
ネイティブ・モードの UDB	場合に応じて DEFAULT (ASCII NATIVE IEEE)

EXEC CICS ステートメントを含むプログラムをコンパイルするには、CICS をインストールしておく必要があります。ワークステーション上に CICS をインストールする方法については、各製品のインストール指示を参照してください。

CICS プリプロセッサのオプション

以下の構文図は、CICS プリプロセッサのサポートするオプションを示しています。



省略形: S、NS、D、ND

SOURCE または NOSOURCE

CICS プリプロセッサへのソース入力を印刷するかどうかを指定します。

PRINT または NOPRINT

CICS プリプロセッサによって生成されるソース・コードを、後続のプリプロセッサまたはコンパイラの生成するソース・リスト (1 つ以上) 内に印刷するかどうかを指定します。

DECK または NODECK

このオプションを指定すると、CICS プリプロセッサ出力ソースは、拡張子 .DEK のファイルに書き込まれます。ファイルは、現行ディレクトリーに存在します。

EDF または NOEDF

PL/I プログラムに対して、CICS 実行診断機能 (EDF) を使用可能にするかどうかを指定します。NOEDF の指定には、パフォーマンス上の利点はありませんが、このオプションは、十分にテストされたプログラムで、CICS コマンドが EDF 表示に現れないようにする上で有用になる場合があります。

DEBUG または NODEBUG

CICS 実行診断機能 (EDF) による使用を目的に、CICS にソース・プログラムの行番号を、CICS プリプロセッサが引き渡すかどうかを指定します。

CICS プリプロセッサ・オプション環境変数

IBM.PPCICS 環境変数を使用して CICS プリプロセッサのデフォルト・オプションを設定することができます。 32 ページの『IBM.PPCICS』を参照してください。

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

「*TXseries for Multiplatforms, CICS アプリケーション・プログラミング・ガイド*」(SD88-7389) 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、以下に説明があります。

CICS ステートメントの組み込み

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

このセクションでは、ワークステーション上の CICS の PL/I サポートに適用される規則と指針について説明します。

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。

S/390 PL/I CICS サポートとの互換性を確保するには、以下の規則を守ることを推奨します。

- マクロ・レベル・サポートを使用しないでください。コマンド・レベル・サポートのみが提供されています。
- 以下を除いて、PL/I 入力または出力を使用しないでください。

SYSPRINT 用のストリーム出力
PLIDUMP

上記はデバッグ専用意図されているため、パフォーマンスの観点から、実動プログラムにインクルードしない方が賢明です。

- 以下のステートメントを使用しないでください。

DELAY
WAIT

- PL/I 言語間機能を使用して、FORTRAN、COBOL、または C と通信しないように推奨します。ただし、異なる言語で書かれた CICS プログラムは、EXEC CICS LINK コマンドまたは XCTL コマンドを使用して相互に通信することができます。

PL/I 以外の言語で書かれたサブルーチンは、サブルーチンに EXEC CICS コードが一切に含まれていない限り、PL/I 言語間機能を使用して呼び出すことができます。EXEC CICS コードを含む非 PL/I プログラムと通信する場合は、説明したように、EXEC CICS LINK または EXEC CICS XCTL を使用する必要があります。

COBOL と C は、以下の IBM PL/I 製品によって、CICS のもとでサポートされています。

IBM Enterprise PL/I for z/OS
IBM PL/I for AIX
IBM VisualAge PL/I for Windows
IBM PL/I MVS および VM

- PLISRTx 組み込みサブルーチンを使用しないでください。
- PLITDLI、ASMTDLI、または EXEC DLI を使用して、IMS に対する呼び出しを行わないでください。

PL/I プログラムに使用される CICS 異常終了

APLS

この異常終了は、強制終了が ERROR 条件によって引き起こされ、ERROR 条件が異常終了 (ASRA 異常終了を除く) が原因でない場合に、強制終了時に発行されます。

これは、以下のいずれかの場合に、PL/I によって発行される異常終了コードです。

1. トランザクションは、PL/I ソフトウェア割り込み (例えば、CONVERSION) を原因とし、エラー状態で強制終了し、ERROR ON ユニットは存在しません。
2. プログラムは、ERROR ON ユニットから正常に戻ります。

プログラムが失敗している関係上、DTB などが必要に応じて発生できるように、障害は、異常終了としてワークステーション上の CICS に反映される必要があります。

APLT

エラーが、ユーザー出口で検出されました。

CICS ランタイム・ユーザー出口

IBM 提供 CICS ユーザー出口 CEEFXITA の検討と変更を (必要に応じて) 行うことを強く推奨します。 364 ページの『CICS ランタイム・ユーザー出口の使用』を参照してください。

第 8 章 コンパイル出力

コンパイラー・リストの使用 139 コンパイラー出力ファイル 147

コンパイル結果は、ソース・プログラムのエラー・フリーの程度、およびユーザーが指定したコンパイル時オプションによって異なります。コンパイル結果には、診断メッセージ、戻りコード、ディスクに保管されているその他の出力（例えば、オブジェクト・モジュールとリストなど）を含めることができます。以下のセクションでは、コンパイラー・リストのサンプルについて説明します。147 ページの『コンパイラー出力ファイル』では、ユーザーがコンパイラーから要求できる他の種類の出力ファイルについて説明しています。

コンパイラー・リストの使用

コンパイル時、コンパイラーは、ソース・プログラム、コンパイル処理、およびオブジェクト・モジュールに関する情報を記載するリストを生成します。TERMINAL オプションは、端末に、診断情報と統計情報を送信します。IBM.PRINT 環境変数は、印刷可能なファイル用の出力ディレクトリーを指定します (IBM.PRINT 環境変数の詳細については、32 ページの『IBM.PRINT』を参照してください)。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。

CHIMES プログラムのこのリストは、コンパイラー・リストの有効性の高いセクションの一部を強調して示します。図 3 は、CHIMES プログラムのコンパイラー・リストとほぼ同じものです。

```
5724-B67  IBM(R) PL/I for Windows(R) V7.6                (Built:20090606)  2009.06.07 11:41:55      Page    1
          Options Specified  1
Environment:

Command:  number options a(s) x nest gonumber lc(55)

Line.File Process Statements
1.0      *PROCESS MACRO S A(F) X AG;
2.0      *PROCESS LANGLVL(SAA2);
3.0      *PROCESS NOT('^\') OR('|');
```

図 3. CHIMES プログラムのコンパイラー・リスト (1/5)

```

Options Used 2
+ AGGREGATE(DECIMAL)
+ ATTRIBUTES(FULL)
  BIFPREC(31)
  BLANK('09'x)
  CHECK( NOCONFORMANCE NOSTORAGE )
  CMPAT(LE)
  CODEPAGE(00819)
  NOCOMPILE(S)
  NOCOPYRIGHT
  CURRENCY('$')
  NODBCS
  DEFAULT(IBM ASSIGNABLE NOINITFILL NONCONNECTED LOWERINC
    DESCRIPTOR DESCLIST DUMMY(ALIGNED) ORDINAL(MIN)
    BYADDR RETURNS(BYVALUE) LINKAGE(OPTLINK) NORETCODE
    NOINLINE REORDER NOOVERLAP NONRECURSIVE ALIGNED
    NULLSYS BINIARG NULLSTRADDR EVENDEC SHORT(HEXADEC)
    ASCII IEEE NATIVE NATIVEADDR E(IEEE))
  NODLLINIT
  NOEXIT
  EXTRN(SHORT)
  FLAG(W)
  FLOATINMATH(ASIS)
+ GONUMBER
  NOGRAPHIC
  IMPRECISE
  INCAFTER(PROCESS(""))
  INCLUDE(EXT('inc' 'cpy' 'mac'))
  NOINITAUTO
  NOINITBASED
  NOINITCTL
  NOINITSTATIC
  NOINSOURCE
  LANGLVL(SAA2 NOEXT)
  LIBS( SINGLE DYNAMIC )
  LIMITS( EXTNAME(100) FIXEDBIN(31,31) FIXEDDEC(15) NAME(100) )
  LINECOUNT(60)
  NOLINEDIR
  NOLIST
  LISTVIEW(SOURCE)
+ MACRO
  MARGINI(' ')
  MARGINS(2,72)
  MAXGEN(100000)
  MAXMSG(W 250)
  MAXNEST( BLOCK(17) DO(17) IF(17) )
  MAXSTMT(4096)
  MAXTEMP(50000)
  NOMDECK
  MSG(*)
  NAMES('@#$' '@#$')
  NATLANG(ENU)
  NONEST
+ NOT('^')
  NUMBER
  OBJECT
  NOOFFSET
  NOONSNAP
  OPTIMIZE(0)
+ OPTIONS(DOC)
  OR('|')
+ PP( MACRO )
  NOPPCICS
  NOPPMACRO
  NOPPINCLUDE
  NOPPSQL
  NOPPTRACE
  PRECTYPE(ANS)
  PREFIX(CONVERSION FIXEDOVERFLOW INVALIDOP OVERFLOW
    NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE
    UNDERFLOW ZERODIVIDE)
  PROBE
  NOPROCEED(S)

```

図 3. CHIMES プログラムのコンパイラー・リスト (2/5)

5724-B67 IBM(R) PL/I for Windows(R) V7.6

(Built:20090606)

2009.06.07 11:41:55

Page 3

```

PROCESS(DELETE)
QUOTE('"')
REDUCE
RESEXP
RESPECT()
RULES(IBM BYNAME NODECSIZE ELSEIF EVEDEC GOTO NOLAXBIF NOLAXCTL
      LAXDCL NOLAXDEF LAXENTRY LAXIF LAXINOUT LAXLINK LAXMARGINS
      LAXPUNC LAXQUAL LAXSCALE LAXSEMI LAXSTG NOLAXSTRZ MULTICLOSE UNREF)
NOSEMANTIC(S)
NOSNAP
NOSOSI
+ SOURCE
  STATIC(SHORT)
NOSTMT
NOSTORAGE
NOSYNTAX(S)
SYSPARM('')
SYSTEM(WINDOWS PENTIUM)
TERMINAL
NOTEST
  USAGE( HEX(SIZE) ROUND(IBM) SUBSTR(STRICT) UNSPEC(IBM) )
  WIDECHAR(LITTLEENDIAN)
  WINDOW(1950)
  XINFO(NODEF NOXML)
  XML( CASE(UPPER) )
+ XREF(FULL)

```

5724-B67 IBM(R) PL/I for Windows(R) V7.6

(Built:20090606)

2009.06.07 11:41:55

Page 4

Compiler Source
Line.File LV NT

```

5.0
6.0 /*****/
7.0 /*                                     */
8.0 /* NAME - CHIMES.PLI                 */
9.0 /*                                     */
10.0 /* DESCRIPTION                      */
11.0 /*   Plays a tune using system API services */
12.0 /*                                     */
13.0 /*   5639-A83, 5639-A24 (C) Copyright IBM Corp. 1992,1996. */
14.0 /*   All Rights Reserved.             */
15.0 /*   US Government Users Restricted Rights-- Use, duplication or */
16.0 /*   disclosure restricted by GSA ADP Schedule Contract with */
17.0 /*   IBM Corp.                         */
18.0 /*                                     */
19.0 /* DISCLAIMER OF WARRANTIES          */
20.0 /*   The following 'enclosed' code is sample code created by IBM */
21.0 /*   Corporation. This sample code is not part of any standard */
22.0 /*   IBM product and is provided to you solely for the purpose of */
23.0 /*   assisting you in the development of your applications. The */
24.0 /*   code is provided "AS IS", without warranty of any kind. */
25.0 /*   IBM shall not be liable for any damages arising out of your */
26.0 /*   use of the sample code, even if IBM has been advised of the */
27.0 /*   possibility of such damages.      */
28.0 /*                                     */
29.0 /*****/
30.0 CHIMES: PROC OPTIONS(MAIN);          /* Play a tune using DOSBEEP tones */
31.0
32.0     DCL ( REST VALUE( 0 ),            /* Declare Named Constants */
33.0           G4 VALUE( 392 ),             /* for note and rest tone */
34.0           C5 VALUE( 523 ),             /* values and timings. */
35.0           D5 VALUE( 587 ),
36.0           E5 VALUE( 657 ),
37.0           WHOLE VALUE( 800 ) ) FIXED BIN(31);
38.0

```

図 3. CHIMES プログラムのコンパイラー・リスト (3/5)

コンパイラー・リストの使用

```
39.0      DCL NOTES(19,2) STATIC NONASGN FIXED BIN(31)
40.0      3      INIT( E5, (WHOLE/2),          /* Declare tone and timing */
41.0              C5, (WHOLE/2),              /* for each note of tune. */
42.0              D5, (WHOLE/2),
43.0              G4, (WHOLE),                /* Initial values may be */
44.0              REST, (WHOLE/2),            /* restricted expressions */
45.0              G4, (WHOLE/2),              /* using Named Constants */
46.0              D5, (WHOLE/2),              /* previously defined in */
47.0              E5, (WHOLE/2),              /* this program. */
48.0              C5, (WHOLE),
49.0              REST, (WHOLE/2),
50.0              E5, (WHOLE/2),
51.0              C5, (WHOLE/2),
52.0              D5, (WHOLE/2),
53.0              G4, (WHOLE),
54.0              REST, (WHOLE/2),
55.0              G4, (WHOLE/2),
56.0              D5, (WHOLE/2),
57.0              E5, (WHOLE/2),
58.0              C5, (WHOLE) );
59.0
60.0      DCL I FIXED BIN(31);
61.0
62.0      /* Declare external APIs called by chimes. */
63.0
64.0      DCL BEEP      ENTRY( FIXED BIN(31), FIXED BIN(31) /* tone, time */
65.0                      EXT( 'Beep' ) /* External name of function*/
66.0                      OPTIONS( BYVALUE /* Pass parameters by value */
67.0                              LINKAGE(STDCLL));
68.0
69.0
70.0
71.0      DCL SLEEP      ENTRY( FIXED BIN(31) ) /* Time duration only*/
72.0                      EXT( 'Sleep' )
73.0                      OPTIONS( BYVALUE
74.0                              LINKAGE(STDCLL) );
75.0
76.0
77.0      /* Play all of the notes and rests of the tune using a do loop. */
78.0
79.0      DO I = LBOUND(NOTES,1) TO HBOUND(NOTES,1);
80.0          IF NOTES(I,1) ^= 0 /* Note the use of ^ for logical NOT*/
81.0              THEN CALL BEEP( NOTES(I,1), NOTES(I,2) );
82.0          ELSE CALL SLEEP( NOTES(I,2) );
83.0      END;
84.0
85.0      END;
```

図 3. CHIMES プログラムのコンパイラー・リスト (4/5)

Attribute/Xref Table 4			
Line	File Identifier	Attributes	
65.0	BEEP	CONSTANT EXTERNAL('Beep') ENTRY(BYVALUE FIXED BIN(31,0), BYVALUE FIXED BIN(31,0)) Refs: 94.0	
34.0	C5	CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0 39.0	
30.0	CHIMES	CONSTANT EXTERNAL ENTRY() CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0 39.0	
35.0	D5	CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0 39.0	
36.0	E5	CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0 39.0	
33.0	G4	CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0 39.0	
++++++	HBOUND	BUILTIN Refs: 92.0	
60.0	I	AUTOMATIC FIXED BIN(31,0) Refs: 93.0 94.0 94.0 95.0 Sets: 92.0	
++++++	LBOUND	BUILTIN Refs: 92.0	
39.0	NOTES	STATIC NONASSIGNABLE DIM(1:19,1:2) FIXED BIN(31,0) INITIAL Refs: 92.0 92.0 93.0 94.0 94.0 95.0	
32.0	REST	CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0	
71.0	SLEEP	CONSTANT EXTERNAL('Sleep') ENTRY(BYVALUE FIXED BIN(31,0)) Refs: 95.0	
37.0	WHOLE	CONSTANT FIXED BIN(31,0) Refs: 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0 39.0	

Aggregate Length Table 5				
Line	File Dims	Offset	Size	Size Identifier
39.0	2	0	152	4 NOTES

File Reference Table 6		
File	Included From	Name
1		C:\ibmpli\%samples%\chimes.pli

Component	Return Code	Messages (Total/Suppressed)	Time 7
MACRO	0	0 / 0	0 secs
Compiler	0	0 / 0	1 secs

End of compilation of CHIMES

図 3. CHIMES プログラムのコンパイラー・リスト (5/5)

1 指定するオプション

コンパイラー・リストのこのセクションは、ユーザーが指定したコンパイル時オプションをすべて示します。Install: の下に示されたオプションは、IBM.OPTIONS 環境変数で指定されています。Command: の下に示されたオプションは、ユーザーがコンパイラーを呼び出したときに、それらのオプションをコマンド行で指定したことを示します (この例では、コマンド・オプションは存在しません)。*PROCESS ステートメントまたは %PROCESS ステートメントで指定されたオプションが、コマンド・オプションの下に示されています。

2 使用するオプション

コンパイラー・リストには、デフォルト・オプションを始めとし、使用されているコンパイル時オプションすべてのリストが含まれます。正符号 (+) のマークが付いたオプションの場合、そのデフォルトが変更されています。相互に矛盾するコンパイル時オプションが存在する場合、コンパイラーは、優先順位が最高のオプションを使用します。以下のリストは、優先順位の高いものから始まり、コンパイラーが使用するオプションを示します。

- ***PROCESS** ステートメントまたは **%PROCESS** ステートメントを使用して指定したオプション。
- **PLI** コマンドを使用して、コンパイラーを呼び出したときに指定したオプション。
- インストール時または **IBM.OPTIONS** 環境変数によってインストールされたインストール・オプション (**IBM.OPTIONS** 環境変数 の詳細については、31 ページの『**IBM.OPTIONS**』を参照してください)。

3 NUMBER オプションの使用

記載されるステートメント番号は、**NUMBER** オプションによって生成されます。この場合には、ステートメントは、ファイル 1 の 14 番目の行から始まっています。リストの一番下にあるファイル参照テーブルも、ファイル 1 が **D:\ibmpli\samples\chimes.pli** を参照していることを示しています。

コンパイル時にステートメント番号を生成すると、リストを参照することなく、(例えば、メッセージで示される) 編集の必要な行を突き止めることができます。

4 属性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、コンパイラー・リストの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。**FULL** 属性は、すべての **ID** と属性をリストに含めます。**ATTRIBUTES** の **SHORT** サブオプションを指定すると、未参照 **ID** はリストに含まれなくなります。

XREF オプションを指定すると、コンパイラーは、**ID** が含まれている **Line.File** 番号 (それぞれ、ファイル内のステートメント番号とファイル番号) とともに、ソース・プログラム内の **ID** のリストを含む相互参照テーブルをコンパイラー・リストに印刷します。

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

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 (この例では、示されていません)。

ATTRIBUTES と **XREF** を (この例のように) 指定すると、2 つのテーブルが結合されます。

明示的宣言変数は、その中に変数の存在する **DECLARE** ステートメントの番号とともにリストに含まれます。暗黙的宣言変数は、アスタリスクによって示さ

れ、文脈的宣言変数（この例では、HBOUND と LBOUND）は正符号 (+) で示されます。（未宣言変数も、診断メッセージに記載されます。）

属性 INTERNAL と REAL が記載されることはありません。それぞれ矛盾する属性の EXTERNAL と COMPLEX がリストに含まれていない限り、それらが有効であることが仮定されます。

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、は明示的に宣言された属性のみがリストに含まれます。

配列の場合は、次元属性が最初に印刷されます。配列の境界が制限付きの式である場合、その式の値が境界に対して示されますが、そうでない場合はアスタリスクが示されます。

ビット・ストリングまたは文字ストリングの長さが制限付きの式である場合、その値が表示されますが、そうでない場合はアスタリスクが示されます。

5 集合長さテーブル

AGGREGATE オプションを指定した場合は、コンパイラーが、コンパイラー・リストに集合長さテーブルを含めます。集合長さテーブルには、プログラム内の各集合のマッピング状態が示されます。表 8 は、集合長さテーブルの列のヘッディングと各列の説明を示しています。

表 8. 集合長さテーブル・ヘッディングと説明

ヘッディング	説明
Line.File	集合が宣言されているステートメント番号とファイル番号
Offset	集合の先頭からの各エレメントのバイト・オフセット
Total Size	集合のバイト単位での全体サイズ
Base Size	データ型のバイト単位でのサイズ
Identifier	集合名と集合内のエレメント

6 ファイル参照テーブル

ファイル参照テーブルの **Included From** 列には、**Name** 列の対応するファイルがインクルードされた箇所が示されます。最初にリストされるファイルはソース・ファイルなので、この列にある最初の項目はブランクです。**Included From** 列のエントリーは、インクルード・ステートメントの行番号、その後続く、ピリオドとインクルード・ステートメントを格納するソース・ファイルのファイル番号を示します。

7 コンポーネント、戻りコード、診断メッセージ、時間

コンパイラー・リストの最終部分は、以下のヘッディングから構成されます。

コンポーネント

情報を提供するコンポーネントまたはプロセッサーを示します。呼び出されている場合、マクロ機能、またはコンパイラー自体が、通知用メッセージを提供できます。

戻りコード

コンパイルの完了時点で発行された、コンポーネントの生成した最高順位の戻りコードを示します。可能な戻りコードは以下のとおりです。

0 (通知)

(この例のように) 警告メッセージは検出されませんでした。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。

4 (警告)

コンパイラーが小さなエラーを発見したことを示しますが、コンパイラーによって、エラーは修正された可能性があります。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。

8 (エラー)

コンパイラーが大きなエラーを発見したことを示しますが、コンパイラーによって、エラーは修正された可能性があります。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る場合があります。

12 (重大エラー)

修正できなかったエラーをコンパイラーが検出したことを示します。プログラムがコンパイルされてオブジェクト・モジュールが生成されても、そのオブジェクト・プログラムは使用できません。

16 (回復不能エラー)

コンパイルのエラー強制終了を示します。オブジェクト・モジュールは正常には作成されませんでした。

注: PL/I 用の CMD ファイルをコーディングするとき、コンパイル後手順を実行するかどうかを決定するため、戻りコードを使用することができます。

メッセージ

以下を示します。

- 存在する場合、発行されたメッセージ数。
- 存在する場合、FLAG コンパイル時オプションによって設定されている重大度レベル以下である理由から、抑制されたメッセージ数。

コンパイラー、マクロ機能、SQL プリプロセッサ、およびランタイム環境用のメッセージは、「メッセージおよびコード」に記載、説明されています。

FLAG オプションで指定された重大度を超えるメッセージのみが発行されます。NOTERMINAL コンパイル時オプションを指定しない限り、メッセージ、ステートメント、および戻りコードが、画面上に表示されます。

時間

コンポーネントのプログラム処理に要する合計時間を示します。

コンパイラー出力ファイル

デフォルト・オプションを使用してプログラムをコンパイルすると、オブジェクト・モジュールが現行ディレクトリー内に作成されます。コンパイル時オプションを変更すると、オブジェクト・モジュールに加えて、他の出力の作成も要求できます。表 9 に、デフォルトでは現行ディレクトリーに格納されるその他の可能なコンパイル出力を示します。

コンパイラー出力ファイルに対しては、すべて、メインプログラム・ファイルと同じ名前が使用されます。ファイル拡張子は、以下のテーブルに指定されています。

表 9. 可能なコンパイル・ディスク出力

出力	ファイル 拡張子	要求の方法 (コンパイル 時オプション)	再配置の方法 (環境変数)
プリプロセスされたソース・テキスト	DEK	該当するプリプロセッサの DECK オプション	IBM.DECK
オブジェクト・モジュール	OBJ	OBJECT	IBM.OBJECT
オブジェクト・リスト	ASM	LIST	IBM.PRINT
テンプレート .DEF ファイル	DEF	XINFO(DEF)	IBM.OBJECT
メッセージ・リスト	XML	XINFO(XML)	IBM.OBJECT

注: プログラム・リストを格納する .LST ファイルは、常時、作成されます。

第 9 章 プログラムのリンク

リンカーの開始	149	オブジェクト・ファイルの指定	153
静的リンク	149	応答ファイルの使用	153
コマンド行によるリンク	149	実行可能ファイルの出力タイプの指定	154
MAKE ファイルでリンク	151	.EXE ファイルの作成	154
入出力	152	ダイナミック・リンク・ライブラリーの作成	155
検索規則	152	実行可能ファイルのパッキング	156
ディレクトリーの指定	153	マップ・ファイルの生成	156
ファイル名のデフォルト	153	リンカーの戻りコード	156

以降のセクションでは、コンパイラーの生成したオブジェクト・ファイルを実行可能プログラム・ファイル (.EXE) またはダイナミック・リンク・ライブラリー (.DLL) にリンクする方法について説明します。

作成するすべての .EXE には、メインルーチン (OPTIONS(MAIN) を含むプロシージャ) がちょうど 1 つ含まれる必要があります。メインルーチンが存在しない場合、リンカーは、プログラムに開始アドレスが存在しないというエラーを報告します。メインルーチンが複数存在する場合、リンカーは、名前 main に対して重複する参照があるというエラーを報告します。

作成するすべての .DLL には、DLLINIT コンパイル時オプションでコンパイルされたモジュールが少なくとも 1 つなければなりません (54 ページの『DLLINIT』を参照)。

リンカーの開始

コンパイラーを使用して、ソース・ファイルからオブジェクト・モジュールの作成を完了したら、リンカーを使用して、オブジェクト・モジュールを PL/I ランタイム・ライブラリーとリンクし、EXE ファイルまたは DLL ファイルを作成します。

静的リンク

ライブラリーを .EXE に静的にリンクするには、LIBS(SINGLE STATIC) または LIBS(MULTI STATIC) のいずれかのコンパイル時オプションを指定します (61 ページの『LIBS』を参照してください)。また、/NOE リンカー・オプションを使用してリンクする必要もあります。

コマンド行によるリンク

ILINK コマンド、続いて、一連のオプション、ファイル名、またはディレクトリーをスペース文字またはタブ文字で区切って指定します。

options

1 つ以上の ILINK オプション。ILINK オプションは、1 つの / または - 文字から始まります。

filename

次に挙げるファイルの種類 の 1 つ以上の名称です。

- オブジェクト・ファイル (ファイル名拡張子 .OBJ)

リンカーの開始

- ライブラリー・ファイル (ファイル名拡張子 .LIB)
- 定義ファイル (ファイル名拡張子 .DEF)
- エクスポート・ファイル (ファイル名拡張子 .EXP)
- リソース・ファイル (ファイル名拡張子 .RES)

ILINK を正しく使用するには、少なくとも、オブジェクト・ファイルを 1 つ指定する必要があります。

directories

1 つの / または ¥ 文字で終了する 1 つ以上のディレクトリーの位置。

responsefile

応答ファイルの名前。ファイル名は、@ 文字の直後に続けます。

リンク時の考慮事項

出力ファイルの名前は、/OUT オプションを使用して指定できます。マップ・ファイルの名前は、/MAP オプションを使用して指定できます。

ユーザーが指定したライブラリーに加えて、リンカーは、デフォルトで、コンパイル時にオブジェクト・ファイル内に定義した PL/I ランタイム・ライブラリーを検索します (61 ページの『LIBS』を参照してください)。

ユーザーが指定したディレクトリーは、LIB 環境変数に設定されているディレクトリーよりも先に、リンカーの検索パスの一部となります。詳しくは、152 ページの『検索規則』および 153 ページの『ディレクトリーの指定』を参照してください。

オブジェクト・ファイルは、ワイルドカード文字を使用して複数指定できます。例えば、あるディレクトリー内のすべてのオブジェクト・ファイルを指定するには、*.OBJ を使用します。

ファイル名拡張子は想定されない

リンカーは、ファイルの拡張子を想定しません。拡張子なしにファイル名を指定すると、リンカーは、拡張子を付けずに、その名前のみでファイルの検索を行います。リンカーは、ファイルを発見できなかった場合、リンクを停止します。

例

以下のコマンドは、FUN.OBJ、TEXT.OBJ、TABLE.OBJ、および CARE.OBJ というオブジェクト・ファイルをリンクします。リンカーは、ライブラリー・ファイル XLIB.LIB とデフォルト・ライブラリー内で、未解決の外部参照を検索します。実行可能ファイルには、その名前が指定されていないため、最初のオブジェクト・ファイルのファイル名とデフォルトの拡張子 .EXE を採用して、FUN.EXE という名前が付けられます。リンカーは、FUNLIST.MAP というマップ・ファイルも生成します。

```
ilink /MAP:funlist fun.obj text.obj table.obj care.obj xlib.lib
```

以下のコマンドは、ファイルの MAIN.OBJ、GETDATA.OBJ、および PRINTIT.OBJ を MAIN.EXE という名前の実行可能ファイルにリンクし、MAIN.MAP という名前のマップ・ファイルを生成します。

```
ilink /MAP main.obj getdata.obj printit.obj
```

Windows では、同じコマンドが、GETDATA.DLL からエクスポートされる機能を指定するエクスポート・ファイル GETDATA.EXP 追加して、少し違った形式を取ります。

```
ilink getdata.obj printit.obj /OUT:getdata.dll /DLL getdata.exp
```

MAKE ファイルでリンク

MAKE ファイルを使用して、プロジェクトの作成に必要な一連の操作（コンパイルとリンクなど）を整理してまとめます。これにより、すべての操作を 1 回で呼び出すことができます。NMAKE ユーティリティを使用すると、変更を受けたファイル、および変更を受けたファイルに依存するファイルまたはその中に組み込まれたファイルのみに操作が実行されるため、変更時間の節約になります。

以下の図に基本的な MAKE ファイルの例を示します。

```
#-----
#
# fun.mak - sample makefile
#
# Usage: nmake fun.mak
#
# The following commands are done only when needed:
#
# - Compiles fun, text, table, care,
#   xlib1, and xlib2
# - Adds xlib1 and xlib2 to library xlib
# - Links fun, text, table, care, and xlib
#   to build fun.exe
#
# Each block is as follows:
# <target>: <list of dependencies for target>
#          <action(s) required to build target>
#-----

OBJS = fun.obj text.obj table.obj care.obj
LIBS = xlib.lib

fun.exe: $(OBJS) $(LIBS)
    ilink /MAP:funlist $(OBJS) $(LIBS)

xlib.lib: xlib1.obj xlib2.obj
    ilib /OUT:xlib.lib xlib1.obj xlib2.obj
fun.obj: fun.pli
    pli fun.pli

text.obj: text.pli
    pli text.pli

table.obj: table.pli
    pli table.pli

care.obj: care.pli
    pli care.pli

xlib1.obj: xlib1.pli
    pli xlib1.pli

xlib2.obj: xlib2.pli
    pli xlib2.pli
```

図 4. MAKE ファイルの例

入出力

リンカーは、オブジェクト・ファイルをユーザーが指定した他のライブラリー・ファイルとリンクして、実行可能プログラム・ファイル (.EXE) またはダイナミック・リンク・ライブラリー (.DLL) を作成するために設計されています。

リンカーは、実行可能ファイルの出力内容の情報を提供するマップ・ファイルをオプションとして生成します。

入力 出力

オプション

実行可能ファイル (.EXE または .DLL)

オブジェクト・ファイル (*.OBJ)

マップ・ファイル (.MAP)

ライブラリー・ファイル (*.LIB)

戻りコード

モジュール定義ファイル (.DEF)

(Windows)エクスポート・ファイル (*.EXP)

(Windows)リソース・ファイル (*.RES)

検索規則

オブジェクト・ファイル(.OBJ)、ライブラリー・ファイル(.LIB)、またはモジュール定義ファイル(.DEF) の検索時、リンカーは、次の列挙順にディレクトリーを検索します。

1. ファイルに対してユーザーが指定したディレクトリー、またはユーザーがパスを指定しなかった場合は現行ディレクトリー。デフォルト・ライブラリーには、パス指定を行いません。

注: ファイルへのパスを指定すると、リンカーは、そのパスのみを検索します。

2. コマンド行で個別に入力されたディレクトリー (ディレクトリーは、スラッシュ (/) または円記号 (¥) 文字で終了する必要があります)。詳しくは、153 ページの『ディレクトリーの指定』に関するセクションを参照してください。
3. LIB 環境変数に設定されているディレクトリー。

リンカーは、ファイルを発見できなかった場合、エラー・メッセージを生成し、リンクを停止します。

例

応答ファイルは、以下のような情報を格納できます。

```
FUN.OBJ TEXT.OBJ TABLE.OBJ CARE.OBJ
NEWLIBV3.LIB
C:¥TESTLIB¥
```

リンカーは、4 つのオブジェクト・ファイルをリンクし、FUN.EXE という名前の実行可能ファイルを作成します。リンカーは、NEWLIBV3.LIB 検索してから、デフォルト・ライブラリーを検索し、参照を解決します。

NEWLIBV3.LIB とデフォルト・ライブラリーを見付けるため、リンカーは、次の列挙順にディレクトリーを検索します。

1. 現行ディレクトリー (NEWLIBV3.LIB がパスなしで入力されているため)

2. C:\¥TESTLIB¥ ディレクトリー
3. LIB 環境変数に設定されているディレクトリー

ディレクトリーの指定

入力ファイルに対する追加ディレクトリーの検索をリンカーに指示するには、コマンド行で個別にドライブまたはディレクトリーを指定します。最後にスラッシュ (/) または円記号 (¥) 文字を付けて、ドライブまたはディレクトリーを指定し、リンカーがその指定をパスとして認識するようにします。

リンカーは、ユーザーが指定したパスを検索してから、LIB 環境変数内のパスを検索します。詳しくは、152 ページの『検索規則』に関するセクションを参照してください。

ファイル名のデフォルト

ファイル名を入力しない場合、リンカーは、以下のデフォルトを仮定します。

表 10. リンカーのファイル名デフォルト

ファイル	デフォルトのファイル名
オブジェクト・ファイル	なし。オブジェクト・ファイル名を少なくとも 1 つ入力する必要があります。
出力ファイル	最初のオブジェクト・ファイルのベース名。
マップ・ファイル	出力ファイルのベース名。
ライブラリー・ファイル	オブジェクト・ファイルに定義されているデフォルト・ライブラリー。LIBS コンパイル時オプションを使用して、デフォルト・ライブラリーを定義します。ユーザーが指定した追加ライブラリーは、デフォルト・ライブラリーの前に検索されます。
モジュール定義ファイル	なし。リンカーは、ユーザーが、すべてのモジュール・ステートメントに対するデフォルトを受け入れるものと仮定します。

オブジェクト・ファイルの指定

コマンド行からリンカーを呼び出すと、リンカーは、他のファイル、オプション、またはディレクトリーとして認識できない入力は、オブジェクト・ファイルであると仮定します。ファイルを区切るには、スペース文字またはタブ文字を使用します。リンカーが入力を解釈する方法の詳細については、149 ページの『コマンド行によるリンク』を参照してください。

オブジェクト・ファイルは、ワイルドカード文字を使用して複数指定することもできます。例えば、あるディレクトリー内のすべてのオブジェクト・ファイルを指定するには、*.OBJ を使用します。

応答ファイルの使用

コマンド行で、リンカー入力を指定する以外にも、応答ファイルにオプションやファイル名パラメーターを格納することができます。応答ファイルは、コマンド行のオプションやパラメーターと組み合わせることができます。

リンカーを呼び出す場合、以下の構文を使用します。

```
ilink @responsefile
```

応答ファイルの使用

responsefile に対する値は、応答ファイルの名前です。@ 記号は、ファイルが応答ファイルであることを示します。応答ファイルが、作業ディレクトリに存在しない場合は、ファイルへのパスとともにファイル名も指定します。

応答ファイルは、リンカー・コマンド行の任意の点に置いて使用を開始できます。応答ファイルは、コマンド行に複数指定できますが、ネストさせることはできません。

オプションは、応答ファイル内の任意の箇所に指定できます。リンカーは、オプションが有効でなかった場合、エラー・メッセージを生成し、リンクを停止します。

コマンド行とまったく同様にして、応答ファイルの内容を指定します。デフォルト構文は、コマンド行上の位置ではなく、ファイル拡張子によって入力を識別するため、ファイル内に行がいくつあるか、ブランク行が存在するかどうかは問題になりません。

例

FUN.LNK という名前の応答ファイルには、以下が格納されます。

```
/DEBUG /MAP
fun.obj text.obj table.obj care.obj
/exec
/map:funlist
graf.lib
```

ilink @fun.lnk を入力すると、リンカーは以下を実行します。

- 4 つのオブジェクト・モジュール fun.obj、text.obj、table.obj、および care.obj を fun.exe という名前の .EXE ファイルにリンクします。出力型が指定されていないため、リンカーは .exe にデフォルト設定します。
- マップ・ファイル funlist.map (拡張子 .map を仮定) を生成します。
- デバッグ情報を保存します (/DEBUG オプションによる)。
- ライブラリー・ファイル graf.lib、およびオブジェクト・ファイルに指定されているデフォルトの PL/I ライブラリーから必要なルーチンをすべてリンクします。

実行可能ファイルの出力タイプの指定

リンカーを使用して、実行可能モジュール (.EXE) やダイナミック・リンク・ライブラリー (.DLL) を作成できます。リンカーは、デフォルトでは、.EXE ファイルを作成します。

オプションを使用して、必要とする出力タイプを指定します。

- .DLL を生成するには、/DLL オプションを指定します。または、モジュール・ステートメント LIBRARY を含めます。

.EXE ファイルの作成

リンカーは、デフォルトでは、.EXE ファイルを作成します。/EXEC オプションを使用して、.EXE と出力ファイルを明示的に指定します。

.EXE ファイルは、直接実行可能なファイルです。プログラムは、ファイルの名前を入力して実行できます。対照的に、DLL と装置ドライバー・プログラムは、他のプロセスから呼び出されて実行されるもので、単独では動作できません。.EXE ファイルのサイズを小さくし、そのパフォーマンスを改善するには、以下のオプションを使用します。

- 出力ファイルのセクションにファイル・アライメントを設定する /ALIGNFILE:*n*。実行可能ファイルのサイズを縮小するには、*n* に係数を小さく設定し、実行可能ファイルのロード時間を縮小するには、係数を大きく設定します。デフォルトでは、アライメントは、512 に設定されています。
- 実行可能ファイルのロード・アドレスを指定する /BASE:*n*。例えば、複数の DLL をベース・アドレスにロードし、DLL が互いにオーバーラップしないようにする場合、リンカーは、再配置レコードを再適用する必要がなくなります。*n* (ロード・アドレス) は、0x10000 の倍数でなければならず、0 にすることはできません。

出力ファイル名の拡張子を指定しない場合、リンカーは、ユーザーが指定した名前に拡張子 .EXE を自動的に追加します。出力ファイル名をまったく指定しない場合、リンカーは、最初にリンクした .OBJ ファイルと同じファイル名の .EXE ファイルを生成します。

ダイナミック・リンク・ライブラリーの作成

ダイナミック・リンク・ライブラリー (.DLL) ファイルは、ライブラリー (.LIB) ファイルとまったく同様に、共通関数の実行可能コードを格納します。(インポート・ライブラリーを使用して) DLL とリンクすると、DLL 内のコードは、実行可能ファイル内にコピーされません。その代わり、DLL 関数のインポート定義のみがコピーされるため、実行可能ファイルは小さくなります。ランタイムに、ダイナミック・リンク・ライブラリーが、.EXE ファイルとともに、メモリーにロードされます。

出力として DLL を作成するには、DLLINIT コンパイラー・オプションを使用して少なくとも 1 つのオブジェクト・ファイルをコンパイルし、/DLL リンカー・オプションでリンクします。DLL に組み込む関数を指定するエクスポート定義 (.EXP) ファイルをインクルードする必要があります。

詳しくは、367 ページの『第 22 章 ダイナミック・リンク・ライブラリーの構築』を参照してください。

DLL のサイズを小さくし、そのパフォーマンスを向上させるには、以下のオプションを使用します。

- 出力ファイルにアライメント係数を設定する /ALIGNFILE: 値。DLL のサイズを縮小するには、*value* に係数を小さく設定し、DLL のロード時間を縮小するには、係数を大きく設定します。デフォルトでは、アライメントは、512 に設定されています。

DLL の場合、/BASE 値を設定すると、指定したロード・アドレスが使用可能である場合に、ロード時間が節約できます。ロード・アドレスが使用可能でない場合は、/BASE 値は無視され、ロード時間上の利点はありません。

DLL の作成が終了すると、その DLL にリンクして、実行可能ファイルを作成することができます。

実行可能ファイルの出力タイプの指定

リンカーは、リンク処理時にオブジェクト・ファイルが必要とする関数を判別します。ILIB ユーティリティを使用して、インポート・ライブラリーを作成した後、.LIB ファイルをリンカーへの入力として使用します。

実行可能ファイルのパッキング

デバッグ時に、/DBGPACK を指定して、実行可能ファイルのサイズを縮小して、デバッガーのパフォーマンスを潜在的に向上させます。

マップ・ファイルの生成

/MAP を指定して、記号情報、および出力ファイル内のオブジェクト・モジュールのセクション名、アドレス、サイズをリストするマップ・ファイルを生成します。マップ・ファイルの名前を指定しない場合、マップ・ファイルは、その名前として、実行可能出力ファイルの名前と拡張子 .MAP を取ります。マップ・ファイルの生成を抑えるには、デフォルトの /NOMAP を使用します。

/LINENUMBERS を指定して、ソース・ファイル行番号と関連アドレスをマップ・ファイルに含めます。

リンカーの戻りコード

リンカーは、以下の戻りコードを返します。

コード 意味

- | | |
|----|--|
| 0 | リンクは正常に完了しました。リンカーは、エラーを検出せず、警告も発行しませんでした。 |
| 4 | 警告が発行されました。出力ファイルに問題のある可能性があります。 |
| 8 | エラーが検出されました。リンクは完了した可能性があります、出力ファイルを正常に実行することはできません。 |
| 12 | 警告が発行され、エラーも検出されました (戻りコード 4 と 8 を参照してください)。 |
| 16 | 重大エラーが検出されました。リンクは異常終了し、出力ファイルを正常に実行することはできません。 |
| 20 | 警告が発行され、重大エラーも検出されました (戻りコード 4 と 16 を参照してください)。 |
| 24 | エラーと重大エラーの両方が発行されました (戻りコード 8 と 16 を参照してください)。 |
| 28 | リンカーが警告を発行し、エラーを検出し、重大エラーを検出しました (戻りコード 4、8、および 16 を参照してください)。 |

MAKE ファイルを使用してリンカーを呼び出す場合は、NMAKE を強制して、ILINK コマンドの前に -7 置き、警告を無視することができます。

第 10 章 リンカー・オプションの設定

コマンド行でのオプションの設定	157	/FIXED、/NOFIXED	165
ILINK 環境変数でのオプションの設定	158	/FORCE	165
リンカーの使用	158	/HEAP	165
数値引数の指定	158	/HELP	165
Windows リンカー・オプションの要約	160	/INCLUDE	166
Windows リンカー・オプション	161	/INFORMATION、/NOINFORMATION	166
/?	161	/LINENUMBERS、/NOLINENUMBERS	166
/ALIGNADDR	161	/LOGO、/NOLOGO	166
/ALIGNFILE	161	/MAP、/NOMAP	167
/BASE	161	/OUT	167
/CODE	162	/PMTYPE	167
/DATA	162	/SECTION	168
/DBGPACK、/NODBGPACK	162	/SEGMENTS	168
/DEBUG、/NODEBUG	163	/STACK	169
/DEFAULTLIBRARYSEARCH	163	/STUB	169
/DLL	164	/SUBSYSTEM	169
/ENTRY	164	/VERBOSE	169
/EXECUTABLE	164	/VERSION	170
/EXTDICTIONARY、/NOEXTDICTIONARY	164		

リンカー・オプションでは、大/小文字を区別しないため、大文字のみ、小文字のみ、または大/小文字を混在させて指定することができます。オプションの前に、スラッシュ (/) の代わりに、ダッシュ (-) を使用することもできます。例えば、`-DEBUG` は、`/DEBUG` と等価になります。オプションは、短い形式でも長い形式でも指定できます。例えば、`/DE`、`/DEB`、および `/DEBU` はすべて、`/DEBUG` と等価になります。各オプションの許容できる最も短い形式については、160 ページの

『Windows リンカー・オプションの要約』を参照してください。小文字、大文字、長短の形式、ダッシュ、およびスラッシュは、以下の例のように、1 行のコマンド行で同時に使用することができます。

```
ilink /de -DBGPACK -Map /NOI prog.obj
```

スペース文字またはタブ文字でオプションを区切ります。次のようにして、リンカー・オプションを指定できます。

- コマンド行
- ILINK 環境変数

コマンド行上で指定されたオプションは、ILINK 環境変数のオプションをオーバーライドします。

一部のリンカー・オプションは、数値の引数を取ります。数値の入力は、10 進数、8 進数、または 16 進数の形式で行うことができます。詳しくは、158 ページの『数値引数の指定』を参照してください。

コマンド行でのオプションの設定

コマンド行で指定されたリンカー・オプションは、(158 ページの『ILINK 環境変数でのオプションの設定』に説明されているように) ILINK 環境変数で以前指定されたオプションをすべてオーバーライドします。

コマンド行でのオプション

オプションは、コマンド行の任意の位置に指定できます。スペース文字またはタブ文字でオプションを区切ります。

例えば、/MAP オプションで、オブジェクト・ファイルをリンクするには、次のように入力します。

```
ilink /M myprog.obj
```

ILINK 環境変数でのオプションの設定

頻繁に使用するオプションは、ILINK 環境変数に格納します。この方法は、リンクのたびに同じコマンド行オプションを繰り返し使用する場合に有効になります。環境変数ではファイル名は指定できず、リンカー・オプションのみ指定できます。

ILINK 環境変数は、コマンド行、コマンド (.CMD) ファイル、またはシステム・プロパティを使用して設定できます。コマンド・ファイルを実行して設定するか、またはコマンド行から設定する場合、オプションは、現行セッションでのみ (コンピューターをリブートするまで) 有効となります。システム・プロパティで設定されている場合、オプションは、ユーザーがコンピューターをブートしたときに設定され、.CMD ファイルを使用するか、コマンド行でオプションを指定してオーバーライドしない限り、リンカーを使用するたびに有効になります。

リンカーの使用

以下の例では、コマンド行上のオプションが、環境変数内のオプションをオーバーライドしています。以下のコマンドを入力した場合

```
SET ILINK=/NOI /AL:256 /DE
ILINK test
ILINK /NODEF /NODEB prog
```

最初のコマンドによって、環境変数は、オプション /NOIGNORECASE、/ALIGNMENT:256、および /DEBUG に設定されます。

2 番目のコマンドによって、環境変数に指定したオプションを使用して、ファイル test.obj がリンクされ、test.exe が作成されます。

最後のコマンドによって、オプションの /NOIGNORECASE と /ALIGNMENT:256 に加えて、オプション /NODEFAULTLIBRARYSEARCH を使用して、prog.obj がリンクされ、prog.exe が生成されます。コマンド行上の /NODEBUG オプションが、環境変数の /DEBUG オプションをオーバーライドしているため、リンカーは、/DEBUG オプションなしでリンクします。

数値引数の指定

一部のリンカー・オプションとモジュール・ステートメントは、数値の引数を取ります。数値は、以下の形式のいずれかで指定できます。

10 進数

0 または 0x を接頭部に持たない数値は、10 進数です。例えば、1234 は 10 進数です。

8 進数 0 (0x でない) を接頭部に持つ数値は 8 進数です。例えば、01234 は 8 進数です。

16 進数

0x を接頭部に持つ数値は 16 進数です。例えば、0x1234 は 16 進数です。

Windows リンカー・オプションの要約

表 11. Windows リンカー・オプションの要約

オプション	説明	デフォルト値
/?	ヘルプの表示	なし
/ALIGNADDR	アドレス・アライメントの設定	/A:0x00010000
/ALIGNFILE	ファイル・アライメントの設定	/A:512
/BASE	優先ロード・アドレスの設定	/BAS:0x00400000
/CODE	実行可能ファイルのセクション属性の設定	/CODE:RX
/DATA	データのセクション属性の設定	/DATA:RW
/DBGPACK、/NODBGPACK	デバッグ情報のパック	/NODB
/DEBUG、/NODEBUG	デバッグ情報のインクルード	/NODEB
/DEFAULTLIBRARYSEARCH	デフォルト・ライブラリーの検索	/DEF
/DLL	DLL の生成	/EXEC
/DLL	実行可能ファイルでのエントリー・ポイントの指定	なし
/EXECUTABLE	.EXE ファイルの生成	/EXEC
/EXTDICTIONARY、/NOEXTDICTIONARY	拡張ディクショナリーを使用するライブラリーの検索	/EXT
/EXTDICTIONARY、/NOEXTDICTIONARY	メモリー内でファイルを再配置しない	/NOFI
/FORCE	エラーを検出した場合でも実行可能出力ファイルを作成	/NOFO
/HEAP	プログラム・ヒープのサイズを設定	/HEAP:0x100000,0x1000
/HELP	ヘルプの表示	なし
/INCLUDE	記号への参照の強制	なし
/INFORMATION、/NOINFORMATION	リンク処理の状況の表示	/NOIN
/LINENUMBERS、/NOLINENUMBERS	マップ・ファイルへの行番号の取り込み	/NOLI
/LOGO、/NOLOGO	ロゴの表示、応答ファイルのエコー出力	/LO
/MAP、/NOMAP	マップ・ファイルの生成	/NOM
/OUT	出力ファイルの命名	最初の .obj ファイルの名前
/PMTYPE	アプリケーション・タイプの指定	/PMTYPE:VIO
/SECTION	セクションに対する属性の設定	/CODE と /DATA による設定
/SEGMENTS	セグメントの最大数の設定	/SE:256
/STACK	アプリケーションのスタック・サイズの設定	/STACK: 0x100000,0x1000
/STUB	DOS スタブ・ファイルの名前の指定	なし
/SUBSYSTEM	必須のサブシステムとバージョンの指定	/SUBSYSTEM: WINDOWS,4.0
/VERBOSE	リンク処理の状況の表示	/NOV
/VERSION	実行ファイルでのバージョン番号の書き込み	/VERSION:0.0

Windows リンカー・オプション

このセクションでは、アルファベット順にリンカー・オプションを説明します。

各オプションに対して、以下の項目を説明します。

- オプションを指定する構文。
- デフォルト設定。
- 許容される省略形。
- オプションとそのパラメーター、および他のオプションとの相互作用の説明。

/?

/? を使用して、有効なリンカー・オプションのリストを表示します。このオプションは /HELP と同等です。

/ALIGNADDR

/ALIGNADDR を使用して、セグメントのアドレス・アライメントを設定します。

アライメント係数は、.EXE ファイルまたは .DLL ファイルのセグメントの開始位置を決定します。ファイルの先頭を基準にして、各セグメントの開始点は、アライメント係数の整数倍 (バイト単位で) の位置に整列させられます。アライメント係数は、2 の累乗で、512 から 256M の間にする必要があります。

デフォルト: /ALIGNADDR:0x00010000

省略形: /ALIGN

/ALIGNFILE

/ALIGNFILE を使用して、セグメントのファイル・アライメントを設定します。

アライメント係数は、.EXE ファイルまたは .DLL ファイルのセグメントの開始位置を決定します。ファイルの先頭を基準にして、各セグメントの開始点は、アライメント係数の整数倍 (バイト単位で) の位置に整列させられます。アライメント係数は、2 の累乗で、512 から 64K の間にする必要があります。

デフォルト: /ALIGNFILE:512

省略形: /A

/BASE

/BASE を使用して、.DLL ファイルの最初のロード・セグメントに対する優先ロード・アドレスを指定します。

address の代わりに、@*filename*, *key* を指定すると、メモリー内でオーバーラップしないように、プログラムのセット (通常は、DLL のセット) が配置されます。

filename は、ファイルのセットに対するメモリー・マップを定義するテキスト・ファイルの名前です。*key* は、指定されたキーで始まる *filename* の行に対する参照です。メモリー・マップ・ファイル内の各行の構文は、*key address maxsize* となります。

各エレメントは、1 つ以上のスペースまたはタブで区切ります。*key* は、ファイル内の一意の名前です。*address* は、仮想アドレス・スペースのメモリー・イメージのロケーションです。*maxsize* は、メモリー・イメージがその中に納まる必要のあるメモリーの容量です。リンカーは、プログラムのメモリー・イメージが、指定されたサイズを超えると警告を発行します。メモリー・マップ・ファイル内のコメントは、セミコロン (;) で始まり、行の最後まで続きます。

デフォルト: /BASE:0x00400000

省略形: /BAS

/CODE

/CODE を使用して、すべてのコード・セクションに対するデフォルト属性を指定します。文字は、任意の順序で指定できます。

文字 属性

E または X

EXECUTE

R READ

S SHARED

W WRITE

デフォルト: /CODE:RX

CODE 記述省略形: None

/DATA

/DATA を使用して、すべてのデータ・セクションに対するデフォルト属性を指定します。文字は、任意の順序で指定できます。

文字 属性

E または X

EXECUTE

R READ

S SHARED

W WRITE

デフォルト: /DATA:RW

省略形: None

/DBGPACK、/NODBGPACK

/DBGPACK を使用して、冗長なデバッグ・タイプ情報を除去します。リンカーは、すべてのオブジェクト・ファイルと必要なライブラリー・コンポーネントからデバッグ・タイプ情報を取り、タイプごとに 1 つのエントリーに情報を縮小します。この結果、実行可能出力ファイルが小規模になり、デバッガーの性能が向上します。

パフォーマンスの考慮: 一般的に、/DBGPACK を使用してリンクを行うと、情報をパックするのに時間を要するため、リンク処理は遅くなります。ただし、冗長なデバッグ・タイプ情報が相当存在する場合は、ファイルに書き込む情報が少なくなるため、/DBGPACK により、実際のリンク処理が高速化されることもあります。

/DBGPACK を指定すると、デフォルトでは、/DEBUG もオンになります。

デフォルト: /NODBGPACK

省略形: /DBI/NODB

/DEBUG、/NODEBUG

/DEBUG を使用して、出力ファイルにデバッグ情報を含めると、デバッガーを使用して出力ファイルをデバッグしたり、パフォーマンス・アナライザーを使用してそのパフォーマンスを分析したりすることができます。リンカーは、出力ファイルに記号データと行番号情報を組み込みます。

デバッグの場合は、オブジェクト・ファイルを TEST を使用してコンパイルします。

パフォーマンス・アナライザーの場合は、PROFILE と GONUMBER を使用してオブジェクト・ファイルをコンパイルします。/DEBUG を使用してリンクすると、実行可能出力ファイルのサイズが大きくなります。

デフォルト: /NODEBUG

省略形: /DI/NODEB

/DEFAULTLIBRARYSEARCH

/DEFAULTLIBRARYSEARCH を使用して、参照の解決時にオブジェクト・ファイルのデフォルト・ライブラリーを検索するように、リンカーに指示します。

このオプションとともに *library* を指定すると、リンカーは、デフォルト・ライブラリーのリストに、そのライブラリー名を追加します。オブジェクト・ファイルに対するデフォルト・ライブラリーは、コンパイル時に定義され、そのオブジェクト・ファイル内に組み込まれます。リンカーは、デフォルトで、デフォルト・ライブラリーを検索します。

/NODEFAULTLIBRARYSEARCH を使用して、外部参照の解決時にデフォルト・ライブラリーを無視するように、リンカーに対して指示を出します。このオプションとともに *library* を指定すると、リンカーはそのデフォルト・ライブラリーを無視しますが、残りのデフォルト・ライブラリー (とオブジェクト・ファイルに定義されているその他のライブラリー) の検索は行います。

library を指定しないで、/NODEFAULTLIBRARYSEARCH を指定する場合は、VA PL/I ランタイム・ライブラリーを含め、使用するライブラリーをすべて明示的に指定することが必要になります。

デフォルト: /DEFAULTLIBRARYSEARCH

省略形: /DEF/NOD

/DLL

/DLL を使用して、ダイナミック・リンク・ライブラリー (.DLL ファイル) として出力ファイルを指定します。オブジェクト・ファイルは、PL/I オプションの DLLINIT を使用してコンパイルしておきます。

/EXEC とともに、/DLL を指定すると、最後に指定したオプションのみが有効になります。

/DLL も、その他のオプションも一切指定しない場合、リンカーは、デフォルトで .EXE ファイル (/EXEC) を作成します。

デフォルト: /EXECUTABLE

省略形: /EXEC

/ENTRY

/ENTRY を使用して、実行可能ファイルにエン트리・ポイント (ルーチンまたは関数の名前) を指定します。

デフォルト: None

省略形: /EN

/EXECUTABLE

/EXEC を使用して、実行可能プログラム (.EXE ファイル) として出力ファイルを指定します。リンカーは、デフォルトで、.EXE ファイルを生成します。

/DLL とともに、/EXEC を指定すると、最後に指定したオプションのみが有効になります。

/EXEC も /DLL も指定しない場合、リンカーは、デフォルトで、.EXE ファイルを作成します。

デフォルト: /EXECUTABLE

省略形: /EXEC

/EXTDICTIONARY、/NOEXTDICTIONARY

/EXTDICTIONARY を使用して、外部参照の解決時に、ライブラリーの拡張ディクショナリーを検索するように、リンカーに指示します。拡張ディクショナリーとは、ライブラリー内のモジュール関係のリストのことです。リンカーが、ライブラリーからモジュールを取り込むとき、拡張ディクショナリーを検査して、そのモジュールがライブラリー内の他のモジュールを必要としていないかを確認した後、自動的に追加モジュールを取り込みます。

リンカーは、デフォルトでは、拡張ディクショナリーを検索し、リンク処理の高速化を図ります。

オブジェクト・コード内に記号を定義する場合、その記号が、リンクするライブラリーの 1 つでも定義されている場合には、/NOEXTDICTIONARY を使用します。それを怠ると、リンカーは、同じ記号を 2 つの別の箇所で定義しているという理由で、エラーを発行します。/NOEXTDICTIONARY を使用してリンクすると、リンカーは、拡張ディクショナリーを検索することなく、直接、ディクショナリーを検索します。この結果、参照を個別に解決する必要が生じるため、リンク処理が遅くなります。

デフォルト: /EXTDICTIONARY

省略形: /EXT/NOE

/FIXED、/NOFIXED

/FIXED を使用して、指定されているベース・アドレスが使用可能でない場合に、ローダーに対して、ファイルをメモリ内で再配置しないように指示します。

ベース・アドレスの詳細については、/BASE リンカー・オプションを参照してください。

デフォルト: /NOFIXED

省略形: /FI/NOFI

/FORCE

/FORCE を使用して、リンク処理時にエラーが発生した場合でも、実行可能出力ファイルを作成します。

デフォルト: /NOFORCE

省略形: /FO/NOFO

/HEAP

/HEAP を使用して、プログラム・ヒープのサイズをバイト単位で設定します。*reserve* 引数は、予約する仮想アドレス・スペースの合計を設定します。*commit* は、初期に割り当てる物理メモリーの量を設定します。*commit* が *reserve* より低い場合は、メモリー要求回数は減少しますが、実行時間が遅くなることがあります。

デフォルト: /HEAP:0x100000,0x1000

省略形: /HEA

/HELP

/HELP を使用して、有効なリンカー・オプションのリストを表示します。このオプションは、/? と等価です。

デフォルト: None

省略形: /H

/INCLUDE

/INCLUDE を使用して、記号への参照を強制します。リンカーは、記号を定義しているオブジェクト・モジュールを検索します。

デフォルト: None

省略形: /INC

/INFORMATION、/NOINFORMATION

/VERBOSE リンカー・オプションの説明を参照してください。

デフォルト: /NOINFORMATION

省略形: /I/NOIN

/LINENUMBERS、/NOLINENUMBERS

/LINENUMBERS を使用して、ソース・ファイル行番号と関連アドレスをマップ・ファイルに含めます。このオプションを有効にするには、リンクするオブジェクト・ファイル内に行番号情報が存在している必要があります。

コンパイル時、GONUMBER オプションを使用して、オブジェクト・ファイルに行番号を含めます (または、TEST オプションを使用して、デバッグ情報をすべて含めます)。

行番号情報を持たないオブジェクト・ファイルをリンカーに処理させると、/LINENUMBERS オプションは、何の効果もありません。

/LINENUMBERS オプションにより、ユーザーが /NOMAP を指定した場合でも、リンカーは、強制的にマップ・ファイルを作成します。

デフォルトでは、マップ・ファイルの名前は出力ファイルと同名で、拡張子は .map となります。このデフォルト名は、マップ・ファイル名を指定してオーバーライドできます。

デフォルト: /NOLINENUMBERS

省略形: /L/NOLI

/LOGO、/NOLOGO

/NOLOGO を指定して、リンカーの開始時に表示される製品情報を抑制します。

コマンド行で応答ファイルの前、または ILINK 環境変数において、/NOLOGO を指定します。応答ファイルの中または後に存在するオプションは、無視されます。

デフォルト: /LOGO

省略形: /LO/NOL

/MAP、/NOMAP

/MAP を使用して、*name* という名前のマップ・ファイルを作成します。マップ・ファイルには、各セグメントの構成、およびオブジェクト・ファイルに定義されているパブリック (グローバル) 記号がリストにまとめられています。記号は、名前順とアドレス順に 2 度リストに記載されます。

ディレクトリーを指定しない場合、マップ・ファイルは、現行作業ディレクトリーに生成されます。*name* を指定しない場合、マップ・ファイルは、実行可能出力ファイルと同じ名前で、拡張子は `.map` となります。

デフォルト: /NOMAP

省略形: /MI/NOM

/OUT

/OUT を使用して、実行可能出力ファイルの名前を指定します。

name で拡張子を指定しない場合、リンカーは、作成しているファイルのタイプに従って拡張子を用意します。

作成するファイル

デフォルトの拡張子

実行可能プログラム

.EXE

ダイナミック・リンク・ライブラリー

.DLL

/OUT オプションを使用しないと、リンカーは、該当する拡張子とともに、ユーザーが指定した最初のオブジェクト・ファイルのファイル名を使用します。

デフォルト: 該当する拡張子の付いた、最初の `.OBJ` ファイルの名前。

省略形: /O

/PMTYPE

/PMTYPE を使用して、リンカーの生成する `.EXE` ファイルのタイプを指定します。ダイナミック・リンク・ライブラリー (DLL) の生成時には、このオプションを使用しないでください。

以下のタイプのいずれかを指定する必要があります。

PM 実行可能ファイルは、ウィンドウで実行される必要があります。

VIO 実行可能ファイルは、ウィンドウまたはフルスクリーンで実行できます。

NOVIO

実行可能ファイルはウィンドウで実行してはいけません。フルスクリーンを使用する必要があります。

デフォルト: /PMTYPE:VIO

省略形: /PM

/SECTION

/SECTION を使用して、*name* セクションのメモリー保護属性を指定します。*name* は大/小文字の区別があります。指定できる属性は、次のとおりです。

文字 設定される属性

E または X

EXECUTE

R READ

S SHARED

W WRITE

以下の例は、.EXE ファイル内のセクション dseg1 に対して、EXECUTE 属性でも WRITE 属性でもなく、READ 属性と SHARED 属性を設定します。

```
/SEC:dseg1,RS
```

デフォルト

セクションには、デフォルトで、以下のように属性が割り当てられています。

セグメント

デフォルトの属性

コード・セクション

EXECUTE、READ (ER)

データ・セクション (.EXE ファイル内)

READ、WRITE (RW)、共有なし

データ・セクション (.DLL ファイル内)

READ、WRITE、共有なし

CONST32_RO セクション

READ、SHARED (RS)

デフォルト: セグメントのタイプに依存します。

省略形: /SEC

/SEGMENTS

/SEGMENTS を使用して、1 つのプログラムに存在可能な論理セグメント数を設定します。*number* は、1 から 16375 の範囲の任意の値に設定できます。158 ページの『数値引数の指定』を参照。

各論理セグメントに対して、リンカーは、セグメント情報を追跡するためのスペースを割り振る必要があります。デフォルトとして、比較的小さいセグメント限界値 (256) を使用して、リンカーは、リンクを高速化し、ストレージ・スペースを低く割り振ることができます。

256 より大きいセグメント制限値を設定すると、リンカーは、セグメント情報用のスペースをより大きく割り振ります。この結果、リンク処理は低速化しますが、多数のセグメントを持つプログラムのリンクが可能になります。

256 個未満のセグメントを持つプログラムの場合、*number* をプログラム内のセグメントの実際数に設定して、リンク時間を短縮し、リンカーのストレージ要件を縮小することができます。

デフォルト: /SEGMENTS:256

省略形: /SE

/STACK

/STACK を使用して、プログラムのスタック・サイズを (バイト単位で) 設定します。スタック・サイズは、0 から 0xFFFFFfe の範囲の偶数にする必要があります。奇数を指定すると、次に大きい偶数に切り上げられます。

reserve は、予約する仮想アドレス・スペースの合計を設定します。*commit* は、初期に割り当てる物理メモリーの量を設定します。*commit* が *reserve* より低い場合は、メモリー要求回数は減少しますが、実行時間が遅くなることがあります。

デフォルト: /STACK:0x100000,0x1000

省略形: /ST

/STUB

/STUB を使用して、作成する出力ファイルの先頭に、DOS 実行可能ファイルの名前を指定します。

デフォルト: None

省略形: /STU

/SUBSYSTEM

/SUBSYSTEM を使用して、プログラムの実行に要求されるサブシステムとバージョンを指定します。*major* 引数と *minor* 引数は、オプションであり、サブシステムの必要最小限のバージョンを指定します。*major* 引数と *minor* 引数は、0 から 65535 の範囲の整数とします。

サブシステム	Major.Minor	説明
WINDOWS	3.10	グラフィック装置接続機構 (GDI) API を使用するグラフィック・アプリケーション。
CONSOLE	3.10	コンソール API を使用する文字モード・アプリケーション。

デフォルト: /SUBSYSTEM:WINDOWS,4.0

省略形: /SU

/VERBOSE

/VERBOSE を使用して、リンク処理が行われるのに応じて、リンクのフェーズ、リンクされているオブジェクト・ファイルの名前とパスを含め、リンク処理情報を表示するように、リンカーに指示します。

Windows リンカー・オプション

リンカーが別のファイルを検出したり、ファイルを想定しない順序で検索したりするため、リンク処理に問題が発生する場合は、/VERBOSE を使用して、リンクしているオブジェクト・ファイルの位置、およびリンク順序を判別します。

このオプションからの出力は、**stdout** に送信されます。Windows 転送記号を使用して、ファイルに出力先を変更できます。

/VERBOSE は、/INFORMATION と同じです。

デフォルト: /NOVERBOSE

省略形: /VERB/NOV

/VERSION

/VERSION を使用して、実行ファイルのヘッダーにバージョン番号を書き込みます。*major* 引数と *minor* 引数は、0 から 65535 の範囲の整数とします。

デフォルト: /VERSION:0.0

省略形: /VER

第 3 部 プログラムの実行およびデバッグ

第 11 章 ランタイム・オプションの使用

ランタイム環境変数の設定	173	複数のランタイム・オプションまたはサブオプションの指定	175
PATH	173	ランタイム・オプション	175
DPATH	173	NATLANG	175
ランタイム・オプションの指定	173	ランタイム DLL の出荷	175
ランタイム・オプションの指定場所	173		

実行可能な形の PL/I プログラムが準備できたら、その実行動作をテストする必要があります。最初のステップとして、プログラムを実行し、何が起こるかを調べます。アプリケーションの性質によっては、プログラムを呼び出す前に入出力のセットアップ (SET ステートメント) を行う必要があります。

ランタイム環境変数の設定

プログラムのランタイム環境を設定するには、環境変数を使用します。

PATH

PATH 環境変数を使用して、現行ディレクトリーにない EXE ファイルおよび CMD ファイルの検索パスを指定します。

```
set path=c:¥ibm;d:¥project
```

この変数では 1 つ以上のディレクトリーを指定できます。上記の例の場合は、最初に現行ディレクトリーが検索され、次に c:¥ibm、続いて d:¥project が検索されます。

DPATH

DPATH を使用して、ランタイム・メッセージの検索パスを指定します。この場合、プログラムは最初に現行ディレクトリーを検索し、次に DPATH 変数に指定されたディレクトリー (1 つまたは複数) を検索します。下記の例では、プログラムは現行ディレクトリー、続いて c:¥set1、d:¥set2 の順に検索します。

```
set dpath=c:¥set1;d:¥set2
```

ランタイム・オプションの指定

アプリケーションを実行するたびに、ランタイム・オプション一式が設定されます。これらのオプションによって、ストレージの割り振りやレポートの生成など、一部のアプリケーション実行プロパティーが決定します。IBM では各ランタイム・オプションにデフォルトを用意していますが、これらのデフォルトは必要に応じてアプリケーション実行前に変更できます。

ランタイム・オプションの指定場所

ランタイム・オプションのデフォルト設定は、環境変数やアプリケーション・ソース・コードの中で変更できます。選択可能な手段を、低優先度から高優先度の順で、以下に示します。

- IBM デフォルトを使用する。

- **CEE.OPTIONS** 環境変数でランタイム・オプションを設定する。

CEE.OPTIONS 環境変数を使ってランタイム・オプションを指定するには、コマンド行で SET コマンドを使用するか、あるいはこれらのオプションをシステム・プロパティに定義します。次に例を示します。

```
set cee.options=natlang(enu)
```

上記のとおり、CEE.OPTIONS 環境変数でオプションを設定する方法は 2 つあります。第 1 の方法、つまりシステム・プロパティに CEE.OPTIONS を設定する方法は、第 2 の、SET コマンドを使用する方法よりも優先度は低くなります。

1. システム・プロパティに CEE.OPTIONS を設定する。

システム・プロパティに指定されたランタイム・オプションは、開始されるすべてのセッションに対して有効になるオプションです。この方法は、実行するすべてのアプリケーションで有効にしたいランタイム・オプションを指定する場合に便利です。

システム・プロパティにすでに CEE.OPTIONS が存在する場合は、既存の変数を変更または追加してください。

2. コマンド行で SET コマンドを使って指定されたランタイム・オプションは、そのセッションまたはウィンドウに対してのみ有効で、システム・プロパティにランタイム・オプションが指定されていれば、それをオーバーライドします。この方法をお勧めします。

ランタイム・オプション設定を変更するには、SET コマンドを使って、希望する設定を指定します。SET コマンドが実行されるたびに、以前の SET コマンドは、システム・プロパティ内の定義も含めて完全に置き換えられます。したがって、以前の SET コマンドで指定されたすべてのランタイム・オプションの設定を後続の SET コマンドでも有効にしたい場合は、それらの設定も含める必要があります。

例えば、システム・プロパティに次のように設定されているとします。

```
cee.options=natlang(jpn)
```

その後、コマンド行から次のコマンドを入力します。

```
set cee.options=natlang(enu)
```

これにより、NATLANG はデフォルト値 NATLANG(ENU) に戻ります。

すべてのランタイム・オプションを IBM 提供のデフォルトに戻すには、CEE.OPTIONS をヌル引数に設定します。

```
set cee.options=
```

Windows では、CEE.OPTIONS の SET コマンドを含むいくつかのコマンドをコマンド・ファイルにまとめることができます。このコマンド・ファイルを実行することは、これらの各コマンドをコマンド行で一つずつ発行することに相当します。

複数のランタイム・オプションまたはサブオプションの指定

一連のランタイム・オプションを指定するときは、各オプションをコンマで区切るようにし、その際にスペースが入らないようにします。

ランタイム・オプションのサブオプションを区切るには、コンマを使用します。サブオプションを指定しない場合でも、省略したことを示すためにコンマを指定する必要があります。末尾のコンマは不要です。サブオプションを何も指定しない場合は、デフォルトが使用されます。例えば NATLANG() と指定しても、構文上は有効です。

各オプションのデフォルト設定については、オプションの構文図や、サブオプションがある場合はそのサブオプションの説明の中で示します。

ランタイム・オプション

このセクションでは、ランタイム・オプション NATLANG について説明します。

NATLANG

NATLANG オプションは、ランタイム・メッセージに使用する各国語を指定します。メッセージ翻訳は、日本語、および大/小文字混合の米国英語に対して提供されます。NATLANG は、メッセージ機能によるメッセージのフォーマット設定方法も決定します。

▶▶ NATLANG ((ENU
JPN)) ▶▶

JPN

これは、日本語を指定する 3 文字の ID です。メッセージ・テキストでは、SBCS (1 バイト文字セット) と DBCS (2 バイト文字セット) の文字を組み合わせることができます。

ENU

これは、大/小文字混合の英語を指定する 3 文字の ID です。メッセージ・テキストは SBCS 文字で構成され、大文字と小文字の両方から成っています。

使用法: NATLANG(ENU)

ダンプ出力だけでなく、ランタイム・オプションやストレージ報告書も、大/小文字混合の英語でのみ書き込まれます。

使用システムで無効な各国語を指定した場合は、デフォルトが使用されます。

ランタイム DLL の出荷

アプリケーションとともに DLL を出荷する場合は、以下のリストを参考にして、必要な DLL をアプリケーションに基づいて決定してください。

Windows では、非マルチスレッド化アプリケーションに対して、以下のファイルが必要です。

- BIN¥HEPWS20.DLL

ランタイム DLL の出荷

- BIN¥IBMWS20.DLL
- BIN¥IBMWSTB.DLL
- BIN¥IBMWS20F.DLL
- BIN¥IBMWS20G.DLL
- BIN¥IBMRTENU.DLL

Windows 上のマルチスレッド化アプリケーションに対しては、以下のファイルが必要です。

- BIN¥HEPWM20.DLL
- BIN¥IBMWM20.DLL
- BIN¥IBMWMTB.DLL
- BIN¥IBMWM20F.DLL
- BIN¥IBMWM20G.DLL
- BIN¥IBMRTENU.DLL

上記の DLL に加えて、Windows アプリケーションが BTREIVE を使用する場合は、BIN¥IBMPBTRV.DLL も出荷する必要があります。

上記のいずれかのファイルまたはモジュールのコピーが含まれているアプリケーションには、以下のラベルを記載する必要があります。

CONTAINS

IBM VisualAge PL/I for Windows Version 2.1.12

Runtime Modules

(c) Copyright IBM Corporation 2004

All Rights Reserved

第 12 章 プログラムのテストとデバッグ

プログラムのテスト	177	テストおよびデバッグで使用する条件 . . .	189
一般的なデバッグのヒント	179	一般的なプログラミング・エラー	190
PL/I デバッグ技法	179	ソース・プログラムの論理エラー	190
デバッグでのコンパイル時オプションの使用 . . .	179	PL/I の使用方法が無効	190
デバッグでのフットプリントの使用	181	未初期化の入力変数の呼び出し	190
デバッグでのダンプの使用	182	ループおよびその他の予期しないエラー . . .	190
定様式 PL/I ダンプ - PLIDUMP	182	ループ処理のヒント	191
トレース情報の SNAP ダンプ	186	予期しない入出力データ	191
デバッグでのエラーおよび条件処理の使用 . . .	186	予期しないプログラム終了	192
エラーおよび条件処理の用語	186	その他の予期しないプログラム結果	193
エラー処理の概念	188	コンパイラーまたはライブラリー・サブルーチン	
システム機能	188	の障害	193
言語機能	188	システム障害	193
修飾条件および非修飾条件の ON ユニット . .	189	ローパフォーマンス	194

効果的な設計やコーディングを行うことは、高品質のプログラムを作成する上で役に立ちますが、続いて、それらのプログラムを徹底的にテストする必要があります。開発のテスト・フェーズに十分な注意を向けることにより、次のような結果が得られます。

- 最小限のテスト実行だけで、プログラムは完全に作動可能になるので、プログラム開発の時間とコストが最小化されます。
- プログラムは、実動作業用にリリースされる前に、設計目標をすべて達成したことが証明されます。
- プログラムには十分なコメントが組み込まれているので、このプログラムのユーザーや保守担当者は、他の支援を必要とせずに作業できます。

通常は、テストのプロセスでバグ (作成者が意図しなかったプログラムのすべての処理を含む一般用語) が発見されます。プログラムからこれらのバグを除去するプロセスをデバッグと呼びます。

この章では、テストやデバッグのあらゆる範囲を対象とするのではなく、最高品質でエラー・フリーの PL/I プログラムの作成に役立つヒントや技法について説明します。さらに、テストおよびデバッグに関する一般的な情報と、PL/I 固有の情報が続きます。

プログラムのテスト

PL/I プログラムのテストは、特にプログラムが論理的に複雑であったり、多数のモジュールを必要とする場合に、難しくなることがあります。それでも、このステップは省略しないでください。と言うのも、実稼働環境に移す前にプログラムのバグを検出し除去することが重要であるためです。

以下に示す 3 つのテスト方法は、すべての PL/I プログラムに適用できます。

コード検査

コード検査 (机上検査とも呼ばれる) では、コードの一部を選択し、それをコンピューターの視点で読みます。ソース・プログラムを印刷したコピー、

またはソース・ファイルのオンライン表示を見て、プログラムのフローをたどります。入力データがある場合は、実際にありそうなデータを推測し、それを可変値に代入してください。計算があるときは、手で計算するか、電卓を使うなどして計算してください。多くの場合、コード検査によって、論理的な問題や構文エラー、コンパイラーで検出されないバグ (例: 「 $n*2$ 」でなく「 $n + 2$ 」になっている) が明らかになります。

データ・テスト

プログラムが設計どおりに動作することを検査するには、そのプログラムにテスト・データを供給します。データ・テストの目的は、プログラムが、実稼働環境で扱わなければならないすべての可能なデータに対して例外 (ランタイム・エラーなど) を処理するかどうかを確認することです。したがって、プログラムをテストするには、多様なデータを使用する必要があります。

例えば、エラー (OVERFLOW 条件など) になることがわかっている両極端のデータをプログラムに処理させて、プログラムの反応を確認します。プログラムには、考えられるすべてのデータに対応できるようにエラー検査 (ERROR ON ユニットなど) を組み込む必要があります。

重要: 置換不可能なデータをテストに使用しないでください。また、テスト中のプログラムがアクセスできる範囲内に置換不可能なデータを保管しないでください。

パス・テスト

プログラムのテストに使用するデータを選択するときは、そのプログラムのすべてのパーツをテストするようなデータを選んでください。つまり、プログラムがいくつかのモジュールで構成されている場合、プログラムのテストに使用するデータは、それらすべてのモジュールを使用する必要があるデータでなければなりません。ある時点でプログラムがとると考えられるパスが 5 つある場合は、プログラムが 5 つのパスそれぞれを通るようなデータ集合を用意してください。

プログラムが複雑になるにつれて、考えられるすべてのパスの組み合わせに対応するデータをプログラムに供給することは事実上不可能になります。ただし、ここで重要なのは、代表的な範囲のパスを検査するテスト・ケースを選択することです。例えば、起こりうるすべての DO ループの反復を検査するのではなく、最初のケースと最後のケース、および中間の 1 ケースについてテストします。

バグは、プログラムをテストするときに発見されますが、これらのバグを除去するには、そのバグの再現が必要となる場合があります。したがって、プログラムをテストするときは、常に既知の状態から開始してください。例えば、バグが検出された場合、プログラム作成者は変数の値、使用されたコンパイル時オプション、メモリー内容などを知る必要があります。その手助けとして、PL/I には SNAP や PLIDUMP などの機能があります。

一般に、昨日は完璧に動作したプログラムに今日バグが現れるのは、マシンの状態に 1 つ以上の変化があったためです。したがって、PL/I プログラムをテストするときは、コンパイル時および実行時のマシンの状態を詳細に把握するようにしてください。

一般的なデバッグのヒント

デバッグは、プログラムが作成者の意図しなかった処理を行うまでそのプログラムを実行するプロセスです。バグの発見後は、最初にバグが生成されたときとまったく同じマシン状態であってもバグが出現しないようにプログラムを変更します。この作業を行うには、後戻り、直観、および試行錯誤を組み合わせることが必要です。有効なデバッグを行う上で主に障害となるのは、1 つのバグを除去することでプログラムに新たなバグを招いてしまう可能性があることです。PL/I 固有のデバッグ技法だけでなく、一般的なデバッグのヒントも考慮する必要があります。

プログラムをデバッグするときは、以下のヒントを参考にしてください。

変更は 1 度に 1 箇所

バグを修正しようとするとき、プログラムのソース・コードへの変更は 1 度に 1 箇所のみに行ってください。1 箇所のみを変更することにより、変更前と変更後のプログラム動作を比較して、変更の効果を正確に測ることができます。

プログラム・ロジック・シーケンスに従う

プログラムのバグは、プログラム実行時に出現した順序どおりに修正してください。

予期しない結果を監視する

プログラム実行状態で予期しない変更が発生した場合、プログラム・ソース・コードにおいてその変更に対応する場所にあるバグを突き止めます。

例えば、プログラム実行状態で望ましくない変更として、10 進値「100」が文字変数「z」に意図せずに割り当てられる、などがあります。この場合、ソース・コードでは、割り当てステートメントに誤った変数を割り当てるエラーがあることがわかります。

PL/I デバッグ技法

PL/I には、プログラム・デバッグの方法がいくつかあります。これらの方法について、以降のセクションで説明します。

コンパイル時オプション

フットプリント

ダンプ

エラーおよび条件処理

デバッグでのコンパイル時オプションの使用

PL/I ワークステーション製品は、コンパイル時にプログラムのバグの多くを診断するように設計されており、どのような誤りがどこにあるかを示したコンパイラー・リストを提供します。さらに、コンパイル時オプションを使用すれば、コンパイラー・リストをより便利なものにすることができます。

以下に示すコンパイル時オプションは、PL/I プログラムのデバッグに役立ちます。

FLAG

一定の重大度を下回る診断メッセージのリストを抑制します。さらに、メッセージ数が指定の数に到達したらコンパイルを終了します。プログラムが期待どおりの動作をせず、コンパイラー・メッセージにはその問題についての説明がない場合は、FLAG を使用すれば、コンパイラー・リストに情報メッセージが含まれ

るようにすることができます。このメッセージ (デフォルトで抑制されていたメッセージ) は、プログラムの問題を解釈するのに役立ちます。 FLAG 使用の詳細については、55 ページの『FLAG』を参照してください。

GONUMBER

デバッグに必要なステートメント番号テーブルを作成します。

PREFIX

指定された PL/I 条件を使用可能または使用不可にします。条件にコンパイル時オプションを付けて指定できるので、ソース・プログラムを変更する必要はありません。 PREFIX(SUBRG STRZ STRG) と指定してコンパイルすると、デバッグで非常に役立ちます。 PREFIX 使用の詳細については、79 ページの『PREFIX』を参照してください。

RULES

コンパイラーが、さまざまな言語規則をどの程度の厳密性をもって施行するかを指定します。このオプションは、一般的なプログラミング・エラーのフラグを立てるのに使用できます。

デバッグでは特に、以下の RULES サブオプションが役に立ちます。

NOLAXIF

IF、WHILE、UNTIL、および WHEN 文節を BIT(1) NONVARYING 以外の値に評価することを禁止します。

NOLAXDCL

組み込み関数の場合と SYSIN ファイルおよび SYSPRINT ファイルの場合を除いて、すべての暗黙宣言およびコンテキスト宣言を禁止します。

NOLAXQUAL

コンパイラーは、レベル 1 以外で、ドット修飾のない構造体メンバーへの参照にフラグを立てます。

例えば、次のようなプログラムを考えます。

```
program: proc( ax1xcb, ak2xcb );
          dcl (ax1xcb, ax2xcb ) pointer;
          dcl
            1 xcb based,
            2 xcba13 fixed bin,...
          ak1xcb->xcba13 = ax2xcb->xcba13;
```

RULES(NOLAXDCL) を有効にすると、上記の 2 つのタイプミスは、コンパイラーによって暗黙宣言とみなされ、エラーとしてフラグが立てられます。

RULES 使用の詳細については、83 ページの『RULES』を参照してください。

SNAP

プログラム内のエラーを突き止めるのに有効なトレース情報のリストをコンパイラーが生成するように指定します。

デバッグでの SNAP 使用の詳細については、186 ページの『トレース情報の SNAP ダンプ』を参照してください。

SNAP 構文の詳細については、90 ページの『SNAP』を参照してください。

XREF

プログラム内で使用する名前と、その名前が参照または設定されるステートメントの番号をまとめたテーブルをコンパイラー・リストに入れるように指定しま

す。これにより、名前がソース・プログラム内のどこで使われるかを容易に追跡できます。XREF 使用の詳細については、99 ページの『XREF』を参照してください。

デバッグでのフットプリントの使用

デバッグするときは、以下の点について定期的に検査することが有効です。

- プログラムが実行フローのどこにいるか (例えば、どのモジュールが実行されているか)。
- ID の値。これにより、ID がいつ変更されたか、および何の値が割り当てられたかを確認できます。

これらの作業を行うには、組み込み関数や、PUT DATA および PUT LIST ステートメント、表示ステートメントを使用します。これらの方法について、以降のセクションで詳細に説明します。

組み込み関数

組み込み関数 PROCNAME、PACKAGENAME、および SOURCELIN は、問題の発生した場所やその問題の原因となったイベントのシーケンスを追跡する目的でプログラムの実行をたどるときに役立ちます。次のステートメントは、プロシージャ名や現在実行中のステートメントの行番号を表示したい場所に挿入できます。

```
display (procname() || sourceline());
```

PUT LIST

ストリングおよびデータ項目をデータ・ストリーム (例: プリンター向け出力ファイル) に送信できます。例えば、次のプロシージャは、FIXEDOVERFLOW 条件が発生したかどうかを示し、その条件の原因となった変数 (この場合は z) の値を印刷出力します。

```
Debug: Proc(x);
      dcl x fixed bin(31);
      on fixedoverflow
        begin;
          put skip list('Fixedoverflow raised because z = '||z);
        end;
      end;
      get list(z);
      x = 8 * z;
```

z が大きすぎる場合は、その値に 8 を掛けると、FIXED BIN(31) 変数には大きすぎる値になるため、FIXEDOVERFLOW 条件が発生します。PUT SKIP LIST は、データ (この場合はストリング「Fixedoverflow raised because z = ...」) をデフォルト・ファイル SYSPRINT に送信します。SYSPRINT を定義するには、export DD= ステートメントを使用します。SYSPRINT 使用の詳細については、239 ページの『SYSIN ファイルおよび SYSPRINT ファイルの使用方法』を参照してください。

PUT DATA

データ項目の値を出力ストリームに送信できます。例えば、プログラムに次の行を指定すると、string1 および string2 の値が出力ストリーム (例: SYSPRINT) に送信されます。

```
put data (string1, string2);
```

DISPLAY

DISPLAY を使用すれば、情報をモニターに送信できます。この方法は、プログラムがどこまで進行したか、あるいはプログラムがどのプロシージャーを実行しているか、などを知りたいときに有効です。次に例を示します。

```
Display ('End of job!');
Display ('Reached the MATH procedure');
Display ('Hurrah! Got past the string manipulation stuff...');
```

DISPLAY と PUT ステートメントを同時に使用した場合は、予測不能な順序で出力されます。DISPLAY ステートメント使用の詳細については、224 ページの『DISPLAY ステートメントの入出力』を参照してください。

デバッグでのダンプの使用

プログラムのデバッグでは、多くの場合、プログラムで使用されるストレージの全部または一部の印刷出力 (ダンプ) を取得すると便利です。ダンプを使って、トレース情報を提供することもできます。トレース情報は、プログラム内のエラーの原因を突き止めるのに役立ちます。

次の 2 種類のダンプが有効です。

PLIDUMP
SNAP

IMPRECISE コンパイル時オプションを使用すると、トレース情報が不完全になる可能性があります。IMPRECISE オプションの詳細については、57 ページの『IMPRECISE』を参照してください。

定様式 PL/I ダンプ - PLIDUMP

PLIDUMP を使用すると、以下の情報を取得できます。

- トレース情報。これにより、条件の発生箇所をソース・プログラムで探し出すことができます。
- ファイル情報。この情報には、ダンプ時にオープンしているファイルの属性、一定のファイル処理組み込み関数の値、および I/O ストレージ・バッファの内容が含まれます。

定様式 PL/I ダンプを取得するには、PLIDUMP への呼び出しをプログラムに組み込む必要があります。CALL ステートメントが現れる場所であればどこでも、ステートメント CALL PLIDUMP を置くことができます。書式は以下のとおりです。

```
call plidump('dump options string', 'dump title string');
```

ダンプ・オプション・ストリング (dump options string)

以下の任意のダンプ・オプション文字で構成されるストリングを示す式。

T - トレース

PL/I は呼び出しトレースを生成します。

NT - トレースなし

ダンプには、呼び出しトレースは出力されません。

F - ファイル情報

ダンプには、オープンしているすべてのファイルの属性の完全セットと、アクセス可能なすべての入出力バッファの内容が出力されます。

NF - ファイル情報なし

ダンプには、ファイル情報は出力されません。

S - 停止

ダンプ後にプログラムは終了します。

E - 終了

現行スレッドまたはプログラム (それがメイン・スレッドの場合) は、ダンプ後に終了します。

K 無視されます。**NK**

無視されます。

C - 継続

プログラムはダンプ後も継続します。

PL/I は、オプションを左から右に読み取ります。無効なオプションは無視されます。矛盾するオプションが存在する場合は、右端にあるオプションが採用されます。

dump title string (ダンプ・タイトル・ストリング)

必要の場合は文字に変換され、ダンプのヘッダーとして印刷される式。このストリングは、実質的には長さの制限はありません。PL/I は、このストリングをヘッダーとしてダンプに印刷します。文字ストリングが省略されると、PL/I はヘッダーを印刷しません。

プログラムが PLIDUMP を何度も呼び出す場合は、そのたびに異なるユーザー ID 文字ストリングを使うようにしてください。そうすれば、各ダンプが発生した場所を容易に識別できます。このヘッダーのほかに、新たに PLIDUMP が呼び出されるたびに、日付、時刻、およびページ番号 1 を示す見出しがユーザー ID の上に印刷されます。

PLIDUMP デフォルト: デフォルトのダンプ・オプションは T、F、および C で、ダンプ・タイトル・ストリングはヌルです。

```
plidump('TFC', ' ');
```

推奨される PLIDUMP コーディング: PLIDUMP はプログラム内のどこからでも呼び出せますが、通常のデバッグ方法では ON ユニットから PLIDUMP を呼び出します。ダンプ後のプログラム継続はオプションなので、プログラムは PLIDUMP を使用すれば、プログラム実行中に一連のダンプをとることができます。

DD:plidump 環境変数を使用すれば、PLIDUMP 出力を配置する場所を指定できます。例えば、次のように指定します。

```
set dd:plidump = d:%mydump;
```

PLIDUMP 指定では、他のオプション (RECSIZE など) をオーバーライドすることはできません。このファイルに関連付けられているデフォルト装置は stderr: です。

PLIDUMP の例: 184 ページの図 5 に示すプログラムを実行すると、185 ページの図 6 で示すような定様式ダンプが生成されます。

```
TestDump: proc options(main);
  declare
    Sysin input file,
    Sysprint stream print file;
  open file(Sysprint);
  open file(Sysin);
  put skip list('AbCdEfGhIjKlMnOpQrStUvWxYz');
  call IssueDump;

  IssueDump: proc;
    call plidump( ' ', 'Testing PLIDUMP');
  end IssueDump;
end TestDump;
```

図5. 定様式ダンプを生成する PL/I コード

IssueDump プロシーチャー内にある PLIDUMP の呼び出しでは、PLIDUMP オプション (このオプションは 2 つの文字ストリングのうちの最初のストリング) を何も指定していないので、デフォルトが使用されます。また、PL/I のデフォルト・ファイル SYSIN および SYSPRINT が明示的にオープンされているので、定様式ダンプでは、入出力バッファの該当部分の内容も表示されます。

```

1      * * * PLIDUMP * * *   Date = 910623   Time = 142249090                      Page 0001

2      User identifier: Testing PLIDUMP

3      * * * Calling trace * * *
      IBM0092I The PL/I PLIDUMP Service was called with Traceback (T) option
      At offset +00000024 in procedure with entry ISSUEDUMP
      From offset +0000010B in procedure with entry TESTDUMP
      * * * End of calling trace * * *

      * * * File Information * * *
      Attributes of file SYSIN
4      STREAM INPUT EXTERNAL
5      ENVIRONMENT( CONSECUTIVE RECSIZE(80) LINESIZE(0) )
6      I/O Built-in functions: COUNT(0) ENDFILE(0)
7      I/O Buffer:      000D9008  00000000 00000000 00000000 00000000 00000000  '.....'
                       000D9018  00000000 00000000 00000000 00000000 00000000  '.....'
                       000D9028  00000000 00000000 00000000 00000000 00000000  '.....'
                       000D9038  00000000 00000000 00000000 00000000 00000000  '.....'
                       000D9048  00000000 00000000 00000000 00000000 00000000  '.....'
                       000D9058  0000      '.....'

      Attributes of file SYSPRINT
      STREAM OUTPUT PRINT EXTERNAL
      ENVIRONMENT( CONSECUTIVE RECSIZE(124) LINESIZE(120) PAGESIZE(60) )
      I/O Built-in functions: PAGENO(1) COUNT(1) LINENO(1)
8      I/O Buffer:      000D8008  20416243 64456647 68496A4B 6C4D6E4F  ' AbCdEfGhIjKlMnO'
                       000D8018  70517253 74557657 78597A20 0D0A0000  'pQrStUvWxYz ....'
                       000D8028  00000000 00000000 00000000 00000000  '.....'
                       000D8038  00000000 00000000 00000000 00000000  '.....'
                       000D8048  00000000 00000000 00000000 00000000  '.....'
                       000D8058  00000000 00000000 00000000 00000000  '.....'
                       000D8068  00000000 00000000 00000000 00000000  '.....'
                       000D8078  00000000 00000000 00000000 00000000  '.....'

      * * * End of File Information * * *
      * * * End of Dump * * * * *

```

図 6. PLIDUMP 出力の例

- 1 PLIDUMP が呼び出された時刻と日付。個別の PLIDUMP 呼び出しごとにこの情報が出力されます。
- 2 PLIDUMP 呼び出しで指定された文字ストリング (PLIDUMP に供給される 2 つのストリングのうちの 2 番目)。この情報は、いくつかのダンプが生成される場合、ダンプを識別するのに役立ちます。
- 3 トレース情報。 * * * Calling trace * * * と * * * End of calling trace * * * で囲まれた部分です。この情報によって、PLIDUMP の呼び出し元であるプロシーチャーにトレースバックできます。上記の例では、PLIDUMP は、TESTDUMP プロシーチャーにネストされているプロシーチャー ISSUEDUMP から呼び出されています。各プロシーチャーの 16 進オフセットもトレース情報として提供されます。

デフォルトでは T オプションが使用されるのでトレース情報が提供されますが、PLIDUMP に NT オプションを指定すれば、トレース情報を抑制できます。
- 4 SYSIN (プログラムで明示的にオープンされた) のファイル属性。
- 5 ファイル SYSIN の ENVIRONMENT オプション。

- 6** ファイル SYSIN の関連 I/O 組み込み関数の値。
- 7** SYSIN ファイルの入出力バッファの内容。最初の列は 16 進アドレスで、以降の列はメモリー内容の 16 進表現です。
- 8** SYSPRINT の入出力バッファの内容。PLIDUMP に供給された 2 番目の文字ストリング (AbCd...) が入出力バッファに入っている点に注意してください。入出力バッファのテキスト表現が、行の右側に表示されています。

トレース情報の SNAP ダンプ

厳密に言うと「ダンプ」ではありませんが、SNAP コンパイル時オプションは、プログラム内でどのようなエラー条件がどこで発生したかを調べるときに使用します。SNAP は、PLIDUMP の「T」オプション (182 ページの『定様式 PL/I ダンプ - PLIDUMP』を参照) の場合と同じトレース情報を提供します。PLIDUMP と同様、SNAP は、1 回のプログラム実行の中で複数回発行することができます。

SNAP ダンプの呼び出しの例を以下に示します。

```
on attention snap;
```

このステートメントは、ATTENTION 条件が発生した場合に SNAP ダンプを呼び出します。

デバッグでのエラーおよび条件処理の使用

PL/I 条件処理は、プログラムのデバッグに効果的なツールです。実行時に検出されたすべてのエラーは、条件に関連付けられています。これらの条件は、次のいずれかの方法で処理できます。

- 指定された条件が発生した場合にプログラムがすべき処理を指定する ON ユニットを記述する。
- 標準システム処置を受け入れる。

エラーおよび条件処理の用語

PL/I エラーおよび条件処理の議論の中で使用されるいくつかの用語を十分に理解しておく必要があります。用語を以下に示します。

確立された

ON ステートメントが実行されるとき、ON ユニットは確立された状態になります。同じ条件を参照する ON または REVERT ステートメントが実行されるか、あるいは関連ブロックが終了すると、確立は終了します。

使用可能

条件が発生して ON ユニットまたは標準処置が実行されるとき、その条件は使用可能です。

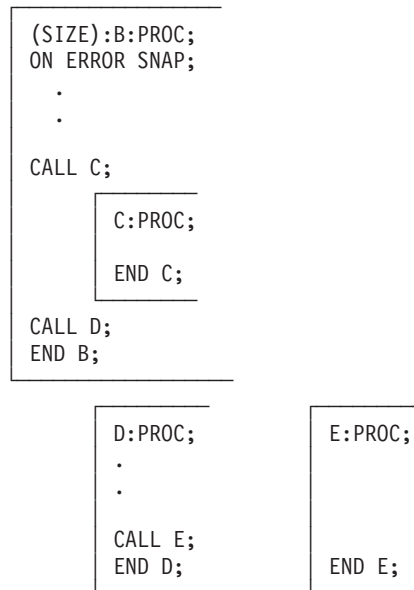
割り込みと PL/I 条件

特定の PL/I 条件は、マシン割り込みによって検出されます。その他の条件は、ランタイム・ライブラリー・モジュールまたはコンパイル済みプログラム内の特別テスト・コードによって検出する必要があります。

静的または動的派生

静的派生および動的派生は、エラー処理機能の有効範囲の定義に使用される用語です。ON ユニットは動的派生です。つまり ON ユニットは、あらゆる状況に

において呼び出し側プロシージャーから継承されます。一方、条件使用可能化は静的派生です。すなわち、ソース・プログラム内の収容ブロックから継承されます。静的派生プロシージャーはコンパイル時に判別できます。動的派生プロシージャーは、実行時までわかりません。図 7 に、静的および動的派生プロシージャーの例を示します。



静的派生:

プロシージャー B の使用可能化の接頭部 SIZE は、どのプロシージャーがどれを呼び出すかにかかわらず、格納されているプロシージャー C のみが継承します。

動的派生:

ON ユニット ON ERROR SNAP は、B によって呼び出されるプロシージャーと、その後に呼び出されるプロシージャーが継承します。つまり、B が D を呼び出し、D が E を呼び出す場合、ON ユニットはプロシージャー E で確立されます。

図 7. 静的および動的派生プロシージャー

通常に戻り

通常に戻りとは、ブロックから GOTO ステートメントへの到達ではなく、END または RETURN ステートメントに到達した後に、呼び出し先ブロックから戻ることです。エラー処理のコンテキストでは、通常に戻りは、ON ユニットからの通常に戻りを意味します。ON ユニットからの通常に戻りの後に行われる処置については、「PL/I 言語解説書」に記載されています。

標準システム処置

標準システム処置は、確立された ON ユニットがない条件が発生した場合に行われる、PL/I 定義のデフォルト処置を指します。

エラー処理の概念

PL/I プログラムをデバッグするときは、以下に示すエラー処理の概念を十分に理解しておく必要があります。条件処理の詳細については、「*PL/I 言語解説書*」を参照してください。

システム機能

オペレーティング・システムには、エラー処理機能があります。さまざまな状態でマシン割り込みが発生し、システム監視プログラムに入る可能性があります。PL/I 制御プログラムは、指定されたルーチンを使って、このような割り込みの後に行われる処置を定義できます。あるいは PL/I 制御プログラムは、PL/I プログラマーが指定した ON ユニットに制御を渡します。

言語機能

PL/I 言語およびその実行環境は、オペレーティング・システムが提供するエラー処理機能をさらに拡張します。PL/I の割り込みが発生する状態は多数あり、中には、エラー処理ではなく通常のプログラム・フローの制御として状態が使用される場合もあります (ENDFILE など)。ON ユニットは、ほとんどの割り込みの後で制御を受け取ることができます。

ON ユニットが割り込み後に制御を受け取るように記述しない場合、以下の処理が可能です。

- ・ 標準システム処置を受け入れる。
- ・ 一定の条件によって割り込みが生じるかどうかを、その条件を使用可能または使用不可にすることによって選択する。条件が使用不可の場合は、その条件が発生したときに、ON ユニットも標準システム処置も実行されません。

ほとんどの PL/I 条件は、プログラム・ロジックまたは提供されたデータの中にエラーがあるために発生します。ただし、エラーとは関連がない場合もあります。このような条件として ENDFILE などがありますが、これらはプログラム実行中いつでも発生する可能性があるので、予測することは困難です。

PL/I には、システム・メッセージと SNAP メッセージの両方があります。

システム・メッセージ

ON ユニットに SNAP と SYSTEM の両方が含まれる場合、結果の PL/I メッセージとしては基本的に PL/I SYSTEM メッセージが出力され、以下に示す 3 行のいずれか (または組み合わせ) がそれに続きます。

From offset xxx in a BEGIN block

From offset xxx in procedure xxx

From offset xxx in a condition_name ON-unit

これらのメッセージは、メイン・プロシージャへのトレースバックに必要な回数だけ繰り返されます。

SNAP メッセージ

ON ユニットに SNAP のみが含まれる場合、結果の PL/I メッセージの先頭部分は、次のようになります。

Condition_name condition was raised
at offset xxx in procedure xxx.

このメッセージの後続部分は、SNAP SYSTEM メッセージと同様です。

オフセットからステートメント番号を判断する: オフセット番号をステートメント番号に変換するには、以下のステップを実行します。

- コンパイルのときに OFFSET コンパイル時オプションを使用します。
- 結果オブジェクト (.cod) のリスト・ファイルをオープンします。
- 最初の列でオフセットを検索して、リストに含まれる最後のソース・ステートメントからそのステートメント番号を見つけます。

条件処理の組み込み関数: PL/I には、条件処理の組み込み関数と疑似変数があります。これらを使用すれば、割り込みに関連したさまざまなフィールドを検査し、特定のケースでは、エラーを引き起こすフィールドの内容を訂正することができます。

このような組み込み関数には、以下のものがあります。

DATAFIELD	ONCOUNT	ONSOURCE
ONCHAR	ONFILE	ONWCHAR
ONCODE	ONGSOURCE	ONWSOURCE
ONCONDCOND	ONKEY	
ONCONDID	ONLOC	

これらの条件処理組み込み関数および疑似変数の詳細については、「PL/I 言語解説書」を参照してください。

修飾条件および非修飾条件の ON ユニット

非修飾条件の ON ユニットはプログラム中の任意の時点で 1 つしか確立できませんが、修飾条件の ON ユニットは複数確立できます。例えば、異なるファイルについて修飾された ENDFILE 条件を処理する場合は、ON ユニットの確立させて、それらのファイルのいずれかについて ENDFILE の発生を一意的に処理することができます。

テストおよびデバッグで使用される条件

プログラムのテストおよびデバッグでは、以下に示す条件が役に立ちます。

- SUBSCRIPTRANGE
- STRINGSIZE
- STRINGRANGE

上記の条件を付けてプログラムを実行すると、パフォーマンスは低下しますが、これらの条件の ON ユニットは、プログラム内のエラーの原因を発見するための強力なツールとして機能します。これらの条件についての ON ユニットの記述することにより、その中の任意の条件を使用可能にできます。そして、その条件が発生した場合は、エラーの原因を知らせる処置を ON ユニットで定義できます。

例えば、プログラムで FIXEDOVERFLOW が発生した場合は、PUT DATA を発行して、発生した条件の原因となったデータの値を知ることが有効です。

ほかにも、PREFIX オプションは、プログラムを編集することなく条件を使用可能にできるので、便利です。

一般的なプログラミング・エラー

PL/I プログラムの実行に失敗するときは、以下の原因が考えられます。

- ソース・プログラムの論理エラー
- PL/I の使用方法が無効 (例: 未初期化変数)
- 未初期化の入力変数の呼び出し
- ループおよびその他の予期しないエラー
- 予期しない入出力データ
- 予期しないプログラム終了
- その他の予期しないプログラム結果
- システム障害
- ローパフォーマンス

ソース・プログラムの論理エラー

ソース・プログラムの論理エラーは、検出が困難なことが多く、ときには、コンパイラーまたはライブラリーに障害があるかのように見える場合もあります。

ソース・プログラムの一般的なエラーの一部を以下に示します。

- 算術データから正しく変換できない。
- 算術演算およびストリング処理演算に誤りがある。
- データ・リストとそのフォーマット・リストが一致しない。

PL/I の使用方法が無効

言語を正しく理解していないと、明らかなプログラム障害が発生することがあります。例えば、以下のプログラミング・エラーがあると、プログラムに障害が発生します。

- 未初期化変数の使用
- 割り振られていない被制御変数の使用
- 不適切な構造体へのレコードの読み込み
- 配列添え字の誤用
- ポインター変数の誤用
- 変換の誤り
- 算術演算の誤り
- ストリング処理演算の誤り
- 割り振られていない、あるいはすでに解放されたストレージの解放または使用

未初期化の入力変数の呼び出し

未初期化の入力変数を呼び出すと、以下の処理が実行されます。

- Windows は即時に記憶保護例外を発生させます。
- ただし、Windows 98 では即時には記憶保護例外は発生しません。つまり、低位アドレス・メモリーで命令を実行できるため、予測不能なプログラム動作が生じる可能性があります。

ループおよびその他の予期しないエラー

PL/I プログラムの実行中にエラーが検出された場合に、実行を終了、またはリカバリーを試行する ON ユニットがそのプログラムに存在しなければ、ジョブは異常終

アします。ただし、下記のステートメントを含む `ERROR ON` ユニットを使用すれば、エラーが発生した時点のプログラムの状況を記録できます。

```
on error
begin;
on error system;
call plidump ('TFBS','This is a dump');
end;
```

`ON` ユニットに含まれる `ON ERROR SYSTEM;` により、未初期化変数を送信しようとしてさらにエラーが発生してもエンドレス・ループにはならないことが保証されます。

処理される条件の特定タイプに基づいた処置を行うようにするには、`ONCONDID` 関数を使用します (この関数の詳細は、「*PL/I* 言語解説書」を参照してください)。

```
on anycondition
begin;
on anycondition system;
select( oncondid() );
when( condid_ofl )
.
.
when( condid_uf1 )
.
.
when( condid_zdiv )
.
.
otherwise
resignal;
end;
end;
```

ループ処理のヒント

`ON` ユニット内で永久ループが発生しないようにするには、以下のコード・セグメントを使用します。

```
on Error begin;
on Error System;
.
.
end;
```

プログラムがエンドレス・ループに陥った場合に、最も重要なのは、マシンをシャットダウンせずにループから抜け出すことです。エンドレス・ループの処理には、以下の解決策をとることをお勧めします。

- ループに入ったときは、**Ctrl-Break** を押して、プログラムを終了します。この環境では、`ATTENTION ON` ユニットは駆動しません。

予期しない入出力データ

プログラムには、障害を引き起こす可能性のある不正な入出力データを事前に検出するための検査が組み込まれている必要があります。

ストリーム指向の入出力によって取得される値を検査したい場合は、GET および PUT ステートメントの COPY オプションを使用します。これらの値は、COPY オプションで指定した名前のファイルにリストされます。ファイル名を指定しない場合は、SYSPRINT とみなされます。

VALID 組み込み関数を使用して、PICTURE および FIXED DECIMAL の ID の妥当性を検査します。

予期しない I/O になる可能性のある機能については、11 ページの『第 4 章 プラットフォーム間でのアプリケーションの移植』を参照してください。移植性の問題(照合シーケンスにおける ASCII と EBCDIC の違いなど)が生じる可能性のある機能の多くは、PL/I プログラムの予期しない I/O につながる可能性があります。

予期しないプログラム終了

プログラムがランタイム診断メッセージを伴わずに異常終了した場合は、その障害を引き起こしたエラーがメッセージの表示も妨げた可能性があります。このように動作した場合、以下の原因が考えられます。

- このバージョンのコンパイラでコンパイルされていないモジュールを実行しようとした。
- export DD= ステートメントに誤りがある。
- 実行可能命令が格納されているストレージ域、特に PL/I 通信域を上書きした。ストレージ域を上書きした原因としては、次のいずれかが考えられます。
 - 存在しない配列エレメントに値を割り当てた。次に例を示します。

```
dc1 array(10);  
      .  
      .  
      .  
do I = 1 to 100;  
  array(I) = value;
```

このタイプのエラーをコンパイル・モジュールで検出するには、SUBSCRIPTRANGE 条件を使用可能にします。宣言された添え字値の範囲を超えるエレメントにアクセスしようとするたびに、SUBSCRIPTRANGE 条件が発生します。この条件の ON ユニットがない場合は、診断メッセージが印刷され、ERROR 条件が発生します。

この方法は、実行時間やストレージ・スペースの面でコストがかかりますが、プログラム・テストの援助機能として有益です。エラー処理の詳細については、186 ページの『デバッグでのエラーおよび条件処理の使用』を参照してください。

- ロケータ (ポインターまたはオフセット) 変数に誤ったロケータ値を使用した。このタイプのエラーは、レコード単位の送信によってロケータ値が取得される場合に生じる可能性があります。

あるプログラムで作成され、データ・セットに送信されたロケータ値を、後に検索して別のプログラムで使用する場合は、そのロケータ値が、2 番目のプログラムで使用する上で妥当であることを確認してください。

- 非 BASED 変数を解放しようとした。このエラーは、BASED 変数の修飾ポインター値が変更された後に BASED 変数を解放すると発生します。次に例を示します。

```
dcl a static,b based (p);
allocate b;
p = addr(a);
free b;
```

- ラベル変数、入り口変数、またはファイル変数に誤った値を使用した。ラベル値、入り口値、およびファイル値が送信され、後に検索される場合、これらの値には前述のロケータ値と同様のエラーが生じる可能性があります。
- SUBSTR 疑似変数を使用して、ターゲット・ストリングの限度を超えるロケーションにストリングを割り当てた。次に例を示します。

```
dcl x char(3);
i = 3
substr(x,2,i) = 'ABC';
```

このタイプのエラーをコンパイル・モジュールで検出するには、STRINGRANGE 条件を使用します (詳しくは、189 ページの『テストおよびデバッグで使用される条件』を参照)。

その他の予期しないプログラム結果

浮動小数点条件に対する Windows の応答方法の違いによって、変更されたプログラム・フローを実感できることがあります。変更されたプログラム・フローの結果の一つとしては、使用不可になっているために発生しない条件があります。

例えば、NOIMPRECISE コンパイル時オプションを使用すれば、IMPRECISE よりも浮動小数点エラーを検出できますが、Windows オペレーティング・システムは、必ずしも浮動小数点例外を即時に検出するとは限りません。浮動小数点例外を発生させる可能性のあるステートメントがプログラム内にある場合は、そのステートメントを単独で BEGIN ブロック内に入れることにより、前記の検出の問題を回避できます。

コンパイラーまたはライブラリー・サブルーチンの障害

障害がコンパイラーの障害またはライブラリー・サブルーチンの障害が原因で発生したことが確実な場合は、IBM に連絡してください。

その一方で、問題を引き起こした操作を実行するための代替方法を検討できます。PL/I 言語には、特定の操作を実行するための代替方法があることが多いので、たいいの場合、バイパスは可能です。

システム障害

システム障害には、マシン誤動作およびオペレーティング・システム・エラーが含まれます。システム・メッセージは、これらの障害をオペレーターに示します。

ローパフォーマンス

ローパフォーマンスは、必ずしもバグが原因というわけではなく、過度のランタイム要件やメモリー所要量が関連しています。留意すべき点は、多数のデバッグ技法 (SUBSCRIPTRANGE の使用可能化など) はパフォーマンスを低下させる傾向があるということです。

パフォーマンスを向上させる機能の一つに、OPTIMIZE コンパイル時オプションがあります (74 ページの『OPTIMIZE』を参照)。プログラム・パフォーマンス向上の詳細については、341 ページの『第 20 章 パフォーマンスの向上』を参照してください。

第 4 部 入出力

第 13 章 データ・セットとファイルの使用

データ・セットのタイプ	198	DELAY	212
ネイティブ・データ・セット	199	DELIMIT	213
従来型テキスト・ファイルおよび装置	199	LRECL	213
固定長データ・セット	199	LRMSKIP	213
その他のデータ・セット	200	PROMPT	213
可変長データ・セット	200	PUTPAGE	214
領域データ・セット	200	RECCOUNT	214
ワークステーション VSAM データ・セット	200	RECSIZE	214
データ・セット特性の設定	201	RETRY	215
レコード	201	SAMELINE	215
レコード・フォーマット	202	SHARE	216
データ・セットの編成	202	SKIPO	216
PL/I ENVIRONMENT 属性の使用による特性の 指定	202	TERMLBUF	217
BKWD	203	TYPE	217
CONSECUTIVE	203	PL/I ファイルとデータ・セットの関連付け	219
CTLASA	204	環境変数の使用	220
GENKEY	204	OPEN ステートメントの TITLE オプションの使 用	220
GRAPHIC	206	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	221
KEYLENGTH	206	PL/I によるデータ・セットの検索方法	221
KEYLOC	207	PL/I ファイルのオープンとクローズ	221
ORGANIZATION	207	ファイルのオープン	221
RECSIZE	208	ファイルのクローズ	222
REGIONAL(1)	208	複数のデータ・セットと 1 つのファイルの関連付 け	222
SCALARVARYING	209	入出力ステートメント、属性、およびオプションの 組み合わせ	222
VSAM	209	DISPLAY ステートメントの入出力	224
DD:ddname 環境変数の使用による特性の指定	209	PL/I 標準ファイル (SYSPRINT と SYSIN).	225
AMTHD	210	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	225
APPEND	211		
ASA	211		
BUFSIZE	211		
レコード入出力用の CHARSET	212		
ストリーム入出力用の CHARSET	212		

PL/I プログラムは、レコードと呼ばれる情報単位を処理し、送信します。レコードの集まりをデータ・セットと呼びますが、PL/I ワークステーション製品では、データ・セットはファイルまたは装置になります。データ・セットは、PL/I プログラムの外部にある情報の論理的な集まりであり、PL/I で書かれたプログラムによって作成、アクセス、または変更できます。

PL/I プログラムは、データ・セットを PL/I ファイルと呼ばれるデータ・セットのシンボリック表現に関連付けることにより、データ・セット内の情報を認識し、処理します。この PL/I ファイルは、入出力操作セットの環境非依存特性を表します。

混同を避けるため、本書では *PL/I ファイル* という語は、PL/I プログラム内で宣言され使用されるファイルを指すものとして使用します。データ・セットおよびワークステーション・ファイル (またはワークステーション装置) という語は、外部入出

力装置上のデータの集まりを指すものとして使用します。データ・セットに名前がない場合もあります。その場合システムは、データ・セットが置かれている装置で認識します。

データ・セットのタイプ

PL/I では、ネイティブ・データ・セットとワークステーション VSAM データ・セットという 2 種類のデータ・セットが定義されています。

- ネイティブ・データ・セットという語は、使用するプラットフォームに関連した従来のテキスト・ファイルおよび装置を示すのに使用される PL/I 用語です。
- ワークステーション VSAM データ・セットという語は、メインフレーム VSAM データ・セットと同種のファイルを指すのに使用されます。PL/I は、DDM、ISAM、または BTRIEVE アクセス方式を使用して、これらのタイプのデータ・セットを作成し、アクセスします。

プラットフォームの区別

この章では、PL/I ワークステーション製品で使用可能なアクセス方式について説明します。ただし、すべてのアクセス方式があらゆるプラットフォームで利用できるわけではありません。この章の情報を参照するときは、以下のガイドラインを参考にしてください。

- DDM - AIX でのみサポートされる
- ISAM - AIX および Windows でサポートされる
- BTRIEVE - Windows でのみサポートされる
- REMOTE - Windows でメインフレームのデータ・ファイルにアクセスする場合にサポートされる

メインフレーム VSAM ファイルを対応する DDM、ISAM、または BTRIEVE ファイルに変換するには、LODVSAM ユーティリティ (AIX ではまだサポートされていない) の始めにある手順に従います。適切なアクセス方式 AMTHD (DDM|ISAM|BTRIEVE) を指定するようにしてください。

DDM、ISAM、または BTRIEVE ファイルをメインフレーム VSAM ファイルに変換するには、RELOAD ユーティリティ (AIX ではまだサポートされていない) の始めにある手順に従います。これらのユーティリティは、PL/I for Windows のサンプル・ディレクトリ内にあります。

メインフレーム上にあるデータ・セットに PL/I プログラムからリモートでアクセスするには、PL/I for Windows コンポーネントの一つである SMARTdata Utilities (SdU) に付属している Distributed FileManager 製品を使用します。SdU の使用方法は、同製品のオンライン・ブックに記載されています。SdU のオンライン・ブックは、このコンポーネントを選択した場合のみインストールされます。

Windows の場合は、「*Distributed FileManager User's Guide*」を参照してください。

ネイティブ・データ・セットには、以下のタイプがあります。

- 従来型テキスト・ファイル
- キャラクター型装置

- 固定長データ・セット

上記のタイプのデータ・セットにアクセスする場合、レコード入出力とストリーム入出力の両方を使用できますが、アクセスは順次方式のみが可能です。

このほかに、PL/I 定義のデータ・セットには以下のタイプがあります。

- 可変長
- 領域
- ワークステーション VSAM データ・セット

領域データ・セットにアクセスする場合は、レコード入出力のみ使用できます。アクセスは順次方式でも直接方式でも可能です。

ネイティブ・データ・セット

PL/I 用語におけるネイティブ・データ・セットとは、使用するプラットフォームに関連した従来型テキスト・ファイルおよび装置を示します。

従来型テキスト・ファイルおよび装置

従来型テキスト・ファイルには、CR - LF (復帰および改行) 文字シーケンスで区切られた論理レコードが入っています。ほとんどのテキスト・エディター・プログラムで従来型テキスト・ファイルを作成でき、変更も可能です。PL/I プログラムは、従来型テキスト・ファイルを作成することも、あるいは他のプログラムで作成されたテキスト・ファイルにアクセスすることもできます。

ワークステーション製品の装置としては、キーボード、画面、およびプリンターがあります。PL/I でこれらの装置を指すときに使用する名前を以下に示します。

NUL: (または NUL)

ヌル出力装置 (出力を廃棄する場合)

STDIN:

標準入力ファイル (デフォルトは CON)

STDOUT:

標準出力ファイル (デフォルトは CON)

STDERR:

標準エラー・メッセージ・ファイル (デフォルトは CON)

注: STDIN:、STDOUT:、および STDERR: はリダイレクトできますが、その他の装置名ではできません。

固定長データ・セット

PL/I では、ファイルを固定長レコードのセットとして処理できます。PL/I プログラムは、固定長データ・セットを作成したり、既存のファイルを固定長データ・セットとしてアクセスしたりできます。このデータ・アクセスでは、復帰 (CR) または改行 (LF) は、特殊な意味を持つ文字としては扱われません。特に、データ・セットに CR - LF 文字が含まれていることがありますが、この文字シーケンスではレコードは区切られません。PL/I がデータ・セット内で 1 レコードとみなす単位は、ユーザーが指定した長さによって決まります。このタイプのデータ・セットには、データ・セット内の合計文字数は指定された長さでちょうど割り切れなければならない、という制限があります。

固定長データ・セットは、順次方式でのみアクセスできます。

その他のデータ・セット

その他のタイプのデータ・セットとして、可変長、領域、およびワークステーション VSAM データ・セットがあります。

可変長データ・セット

PL/I プログラムは、各レコードの先頭にレコードの残りの部分のバイト数を示す 2 バイトの接頭部が付いたデータ・セットを作成し、アクセスすることができます。レコードが CR - LF で区切られているファイルとは異なり、この可変長ファイルには、任意のビット・パターンを含む可能性のあるレコードを保有できます。

領域データ・セット

領域データ・セットの詳細および使用方法については、253 ページの『第 15 章 領域データ・セットの定義と使用』で説明します。

注: このコンテキストにおいて領域は、OS PL/I の REGIONAL(1) と同じ意味です。

ワークステーション VSAM データ・セット

PL/I ワークステーション製品では、VSAM ファイル編成がサポートされています。ワークステーションの VSAM データ・セットには 3 つのタイプがあります。

- 連続。VSAM 入力順データ・セット (ESDS) と同じです。
- 相対。VSAM 相対レコード・データ・セット (RRDS) と同じです。
- 索引付き。VSAM キー順データ・セット (KSDS) と同じです。

PL/I ワークステーション製品は現在、以下の VSAM データ・セット・アクセス方式をサポートしています。

- DDM (AIX のみ)
- ISAM (AIX および Windows)
- BTRIEVE (Windows のみ)
- REMOTE (Windows でメインフレームのデータ・ファイルにアクセスする場合)
- DDM
- ISAM

DDM アクセス方式

DDM データ・セットは、分散データ管理アーキテクチャーによって定義されるレコード単位ファイルです。DDM アクセス方式を使用するワークステーション VSAM データ・セットは、ローカル・システム上に置くことができます。メインフレーム VSAM データ・セットを参照するほとんどの既存メインフレーム・プログラムをコンパイルおよび実行できます。

DDM キー順データ・セットは 2 つのファイルによって表され、一方をベース、もう一方を基本索引 と呼びます。ベースにはデータ・セットのレコードが保持され、基本索引にはデータ・セットの主キーに関する情報が格納されます。DDM キー順データ・セットを作成するときは、ベースの名前を指定してください。DDM は、このベース名を派生させて、基本索引の名前を生成します。

DDM データ・セットを使用する場合、指定された最大長をレコード長が上回ることではできませんが、その点以外でレコード長を気にする必要はありません。

該当する ワークステーション VSAM データ・セットを PC 上に作成してから、プログラムを実行して、メインフレーム VSAM データ・セットを参照するほとんどの既存メインフレーム・プログラムをコンパイルおよび実行できます。

ISAM アクセス方式

特に指示がない限り、本章で *ISAM* という語は、メインフレーム *ISAM* ではなく、*ISAM* ローカル・アクセス方式を指します。*ISAM* データ・セットは 1 つのファイルに格納され、ローカル・ファイル・システムにのみ置くことができます。

BTRIEVE アクセス方式 (Windows のみ)

BTRIEVE アクセス方式は、CICS のもとで作成されたファイルに PL/I 入出力ステートメントを使ってアクセスできるようにするために提供されています。現在 PL/I では、BTRIEVE セグメント化された複数キーはサポートされていません。

BTRIEVE データ・セットは 1 つのファイルに格納され、ローカル・ファイル・システムにのみ置くことができます。

REMOTE アクセス方式 (Windows)

REMOTE アクセス方式は、メインフレーム上のデータ・ファイルにリモートでアクセスできるようにするために提供されています。

ワークステーション VSAM の詳細は、265 ページの『第 16 章 ワークステーション VSAM データ・セットの定義と使用』で説明します。

データ・セット特性の設定

プログラム内でファイルを宣言またはオープンするときは、そのファイルの特性を PL/I に対して記述します。DD:ddname 環境変数、または OPEN ステートメントの TITLE オプションの式を使用して、データ・セット内、あるいはその関連 PL/I ファイル内のデータの特性を PL/I に対して記述することもできます。詳しくは、219 ページの『PL/I ファイルとデータ・セットの関連付け』を参照してください。

必ずしもプログラムの内部と外部の両方でデータを記述する必要はありません。多くの場合は、1 回記述すれば、データ・セットとその関連 PL/I ファイルの両方に対して有効になります。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。

使用するプログラム・データおよびデータ・セットを効率よく記述するには、PL/I 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。

レコード

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。レコードの長さは、以下のいずれかにおける RECSIZE オプションの中で指定できます。

DD 情報

PL/I ENVIRONMENT 属性

OPEN ステートメントの TITLE オプション

データ・セット特性の設定

特定のストリーム・ファイルではデフォルトが適用されますが、このような場合を除き、PL/I プログラムがデータ・セットを作成するときは RECSIZE オプションを指定する必要があります。ストリーム・ファイルの詳細については、227 ページの『第 14 章 連続データ・セットの定義と使用』を参照してください。

PL/I 以外によって作成されたデータ・セットにプログラムがアクセスする場合も、RECSIZE オプションを指定する必要があります。

エディターはデータ・セットを暗黙的に変更することがあるので、注意してください。エディターを使って非 CR - LF ファイルを調べる場合、たいていのエディターは CR - LF、または類似する文字シーケンスを自動的に挿入してしまうので、特に注意が必要です。

レコード・フォーマット

データ・セット内のレコードのフォーマットは、次のいずれかになります。

- 不定長
- 固定長
- 可変長

ネイティブ・ファイルの場合は、DD 情報の TYPE オプションの中で、不定長または固定長レコード・フォーマットを指定します。ワークステーション VSAM データ・セットの場合は、可変長レコードで構成されていることを暗黙的に示すので、レコード・フォーマットを指定する必要はありません。

データ・セットの編成

データ・セット編成を指定する PL/I ENVIRONMENT 属性のオプションは、以下のとおりです。

- CONSECUTIVE
- ORGANIZATION(CONSECUTIVE)
- ORGANIZATION(INDEXED)
- ORGANIZATION(RELATIVE)
- REGIONAL(1)
- VSAM

各オプションについては、『PL/I ENVIRONMENT 属性の使用による特性の指定』で説明します。

ENVIRONMENT 属性でデータ・セット編成オプションを指定しない場合は、デフォルトで CONSECUTIVE に設定されます。

PL/I ENVIRONMENT 属性の使用による特性の指定

DECLARE ステートメントの ENVIRONMENT 属性を使用すれば、特定のデータ・セット特性をプログラム内に指定できます。これらの特性は、PL/I 言語の一部ではありません。そのため、ファイル宣言でこれらの特性を使用すると、そのプログラムは他の PL/I インプリメンテーションに移植できなくなる可能性があります。

プログラムにファイルの環境オプションを指定した例を以下に示します。

```
declare Invoices file environment(regional(1), recsize(64));
```

ENVIRONMENT 属性で指定できるオプションについては、以降のセクションで示します。

BKWD

BKWD オプションは、DDM データ・セットに関連付けられた SEQUENTIAL INPUT ファイルまたは SEQUENTIAL UPDATE ファイルの逆方向処理を指定します。

▶▶—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、その次の低位キーをもつレコードのことです。

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。BKWD オプションは、GENKEY オプションと一緒に指定しないでください。

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。

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。CONSECUTIVE 編成のデータ・セットでは、レコードは物理的順序で配置されます。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。

▶▶—CONSECUTIVE—◀◀

CONSECUTIVE オプションを使用すれば、ストリーム指向のデータ伝送またはレコード単位のデータ伝送を使ってネイティブ・データ・セットにアクセスできます。このオプションは、SEQUENTIAL 属性を使って宣言され、ワークステーション VSAM データ・セットに関連付けられた入力ファイルに対しても使用できます。この場合、ワークステーション VSAM キー順データ・セット内のレコードは、キー順で示されます。

CONSECUTIVE は、デフォルトのデータ・セット編成です。

CTLASA

CTLASA オプションは、レコードの先頭文字を米国標準規格 (ANS) の印刷制御文字として解釈するように指定します。このオプションは、連続データ・セットに関連付けられた RECORD OUTPUT ファイルにのみ適用されます。



ANS 印刷制御文字 (228 ページの表 14 にリストを示す) は、関連するレコードが印刷される前に、指定の処置を実行するための文字です。

CTLASA オプションの使用の詳細は、227 ページの『プリンター向けファイル』を参照してください。

IBM Proprinter 制御文字は、ANS プリンター制御文字で必要な単一バイトよりも多く、最大で 3 バイトが必要です。ただし、論理レコード長の指定 (RECSIZE 環境オプションを参照) は調整しないでください。と言うのも、CTLASA が指定されると、PL/I は論理レコード長に自動的に 3 を追加するからです。

レコードの先頭文字が IBM Proprinter 制御文字に変換されずに残るように、CTLASA の効果を変更することができます。211 ページの『ASA』にある ASA 環境オプションを参照してください。

プリンター向け出力操作に対して SCALARVARYING 環境オプションを指定しないでください。指定してしまうと、PL/I はレコードの先頭データ・バイトの解釈方法を認識できなくなります。

GENKEY

GENKEY (総称キー) オプションは、ワークステーション VSAM 索引付きデータ・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、SEQUENTIAL KEYED INPUT ファイルまたは SEQUENTIAL KEYED UPDATE ファイルを使ってキー・クラス別にレコードにアクセスすることができます。



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も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もつ最初の非ダミー・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も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 (indexed 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' で始まるキーを持つデータ・セット内の最初の非ダミー・レコードが INFIELD に読み込まれます。2 番目の READ ステートメントが実行されるたびに、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も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も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もつ次のレコードに、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、KEYLENGTH 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。KEYLENGTH 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は変わりません。ソース・キーは、必要に応じて右側が切り捨てられますが、このソース・キーによって特定レコード (キーがそのクラスの唯一のメンバーと考えられるレコード) が識別されます。

GRAPHIC

リスト指示 I/O やデータ指示 I/O の GET および PUT ステートメントで DBCS 変数または DBCS 定数を使用する場合は、GRAPHIC オプションを指定する必要があります。GRAPHIC オプションは、編集指示 I/O にも指定できます。

▶▶—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、PL/I は、リスト指示 I/O およびデータ指示 I/O の ERROR 条件を発生させます。

グラフィック・データ型および編集指示 I/O 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。

KEYLENGTH

KEYLENGTH オプションは、KEYED ファイルの記録済みキーの長さ n を指定します。KEYLENGTH は、INDEXED ファイルについてのみ指定できます (このセクションの後の方にある ORGANIZATION を参照)。

▶▶—KEYLENGTH—(n)—◀◀

ファイル宣言に KEYLENGTH オプションが含まれていて、その関連データ・セットがすでに存在する場合は、その値が検査に使用されます。このオプションで指定したキーの長さがデータ・セットに定義された値と矛盾する場合は、UNDEFINEDFILE 条件が発生します。

ISAM および BTREIEVE

キーは、ISAM ファイルまたは BTREIEVE ファイルの索引ページに保持されます。ファイルを作成するときは、キーの長さを PL/I に定義する必要があります。

KEYLOC

KEYLOC オプションは、KEYED ファイルのレコードにおける埋め込みキーの開始位置 n を指定します。KEYLOC は、INDEXED ファイルについてのみ指定できます (このセクションの後の方にある ORGANIZATION を参照)。

►►—KEYLOC—(n)—————◄◄

位置 n は、次の範囲内でなければなりません。

$$1 \leq n \leq \text{recordsize} - \text{keylength} + 1$$

つまり、キーがレコード長を超えることはできないので、レコード内にキーが完全に収まるようにする必要があります。

したがって、SCALARVARYING オプションを指定する場合は、埋め込みキーがレコードの先頭の 2 バイトと重複しないようにしてください。それには、KEYLOC に指定する値を 2 よりも大きい値にする必要があります。

索引付きデータ・セットの作成時に KEYLOC を指定しない場合、キーはレコードの先頭バイトから始まるとみなされます。

ファイル宣言に KEYLOC オプションが含まれていて、その関連データ・セットがすでに存在する場合は、その値が検査に使用されます。このオプションで指定したキー位置がデータ・セットに定義された値と矛盾する場合は、UNDEFINEDFILE 条件が発生します。

ISAM および BTREIEVE

キーは、ISAM ファイルまたは BTREIEVE ファイルの索引ページに保持されます。ファイルを作成するときは、キーの位置を PL/I に定義する必要があります。

ORGANIZATION

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

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。連続ファイルとなるのは、ネイティブ・データ・セットか、ワークステーション VSAM の順次、直接、またはキー順データ・セットです。

INDEXED

ファイルを索引付きデータ・セットに関連付けることを指定します。INDEXED は、データ・セット内のレコードが、各レコードに埋め込まれたキーによる論理的順序で配置されていることを指定します。論理レコードは、ASCII 照合シー

データ・セット特性の設定

ケンスに従った昇順キー配列でデータ・セット内に配置されます。索引付きファイルは、ワークステーション VSAM のキー順データ・セットです。

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、ワークステーション VSAM の直接データ・セットです。相対キーの範囲は、1 から nnnn です。

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。



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。

ファイル宣言に RECSIZE オプションが含まれていて、そのファイルが、すでに存在するワークステーション VSAM データ・セットに関連付けられている場合は、その値が検査に使用されます。このオプションで指定したレコード長がデータ・セットに定義された値と矛盾する場合は、UNDEFINEDFILE 条件が発生します。

テキスト・エディターなどの非 PL/I プログラムで作成されたデータ・セットにアクセスするときは、RECSIZE オプションを指定してください。

ISAM および BTREVE

BTREVE または ISAM アクセス方式を使用するときは、RECSIZE を指定する必要があります。

REGIONAL(1)

REGIONAL(1) オプションは、領域編成のファイルを定義します。



領域編成のデータ・セットには、記録済みキーを持たない固定長レコードが入っています。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)。

領域データ・セットの使用の詳細は、253 ページの『第 15 章 領域データ・セットの定義と使用』を参照してください。

SCALARVARYING

SCALARVARYING オプションは、VARYING スtringの入出力で使用されます。



VARYING スtring用にストレージが割り振られると、コンパイラは、Stringの現行の長さを指定する 2 バイトの接頭部を組み込みます。エレメント可変長Stringの場合は、ファイルに SCALARVARYING を指定したときのみ、この接頭部は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 VARYING String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。

このオプションを指定して、エレメント VARYING String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。

同じファイルに対して SCALARVARYING と CTLASA を指定しないでください。これらを一緒に指定すると、先頭データ・バイトがあいまいになってしまいます。

VSAM

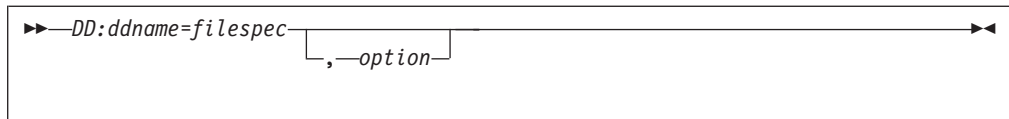
VSAM オプションは、OS PL/I との互換性を果たせるために提供されています。



DD:ddname 環境変数の使用による特性の指定

SET コマンドを使用すれば、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できます。さらにオプションで、そのデータ・セットの追加特性も指定できます。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。



この構文では空白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、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、220 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。

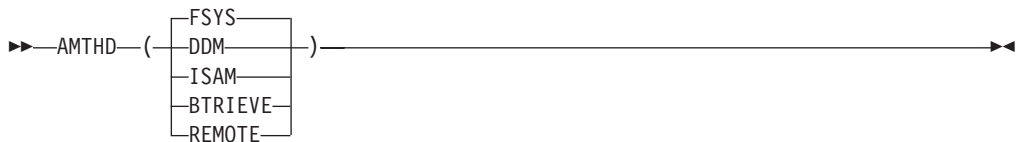
代替 ddname を使用する場合に、その長さが 31 文字を超えるときは、最初の 31 文字だけが環境変数名に指定されます。

option

DD 情報として指定できるオプションは、『AMTHD』から 217 ページの『TYPE』で説明します。

AMTHD

AMTHD オプションは、データ・セットのアクセスに使用するアクセス方式を指定します。



FSYS

PL/I がネイティブ・アクセス方式を使用してネイティブ・ファイルにアクセスすることを指定します。これはデフォルトです。

ISAM

ISAM アクセス方式を使用して ISAM ファイルにアクセスすることを指定します。

BTRIEVE (Windows)

BTRIEVE アクセス方式を使用して BTRIEVE ファイルにアクセスすることを指定します。

REMOTE (Windows)

ファイルがリモート DDM ターゲット・システム (MVS など) 上にあることを指定します。

Windows の場合、ファイルの名前は LU 別名または完全修飾 SNA ネットワーク名で修飾する必要があります。

AMTHD オプションを指定せず、次のいずれの ENVIRONMENT オプションも適用しない場合は、デフォルトで FSYS が使用されます。

ORGANIZATION(INDEXED)
 ORGANIZATION(RELATIVE)
 VSAM

上記オプションのいずれかを指定した場合、Windows では AMTHD(ISAM) がデフォルトになり、AIX では AMTHD(DDM) がデフォルトになります。

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、再作成されるのかが指定されます。

▶▶ APPEND—()—▶▶

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。これはデフォルトです。

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

ASA

ASA オプションは、プリンター向けファイルに適用されるオプションです。このオプションは、各レコード内の ANS 制御文字が解釈されるときを指定します。

▶▶ ASA—()—▶▶

N レコードがデータ・セットに書き込まれる際、ANS 印刷制御文字が IBM Proprinter 制御文字に変換されることを指定します。これはデフォルトです。

Y ANS 印刷制御文字が変換されないことを指定します。この制御文字は、指定したプロセスにより、後で変換されるようにそのまま残されます。

ファイルがプリンター向けファイルでない場合、このオプションは無視されます。プリンター向けファイルについては、227 ページの『プリンター向けファイル』で説明します。

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 EBCDIC 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。



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。

CHARSET(ASIS) がデフォルトです。

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 EBCDIC 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 ASCII として扱われます。



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。

CHARSET(ASCII) がデフォルトです。

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—(n)—————◄◄
```

DELAY のデフォルト値は 0 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—( ☐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—( ☐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—($\begin{array}{|c|} \hline N \\ \hline Y \\ \hline \end{array}$)——▶▶

PROMPT(N) がデフォルトです。

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。このオプションは、プリンター向けファイルにのみ適用されます。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—($\begin{array}{|c|} \hline NOCR \\ \hline CR \\ \hline \end{array}$)——▶▶

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。これはデフォルトです。

CR

用紙送り文字の後ろに復帰文字を入れることを指します。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 オプションは適用されるがその指定が省略されている場合、領域ファイルおよび相対ファイルのデフォルトは 50 です。

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。

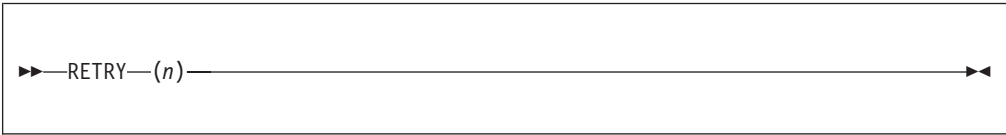
▶▶ RECSIZE—(n)——▶▶

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。

n のデフォルト値は 512 です。

RETRY

RETRY オプションは、システムがファイル・ロックまたはレコード・ロックを取得できない場合に操作を再試行する回数を指定します。



RETRY のデフォルト値は 10 です。このオプションを適用できるのは、DDM ファイルだけです。

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。



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)	ENTER: (cursor)
prompt(n), sameline(y)	ENTER: (cursor)
prompt(y), sameline(n)	ENTER: (cursor)
prompt(n), sameline(n)	ENTER: (cursor)

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)	ENTER: (cursor)
prompt(n), sameline(y)	ENTER (cursor)
prompt(y), sameline(n)	ENTER : (cursor)
prompt(n), sameline(n)	ENTER (cursor)

SHARE

SHARE オプションは、許可するファイル共用のレベルを指定します。



NONE

ファイルを他のプロセスと共用しないことを指定します。これはデフォルトです。

READ

ファイルは他のプロセスによる読み取りが可能であることを指定します。

ALL

ファイルは他のプロセスによる読み取りまたは書き込みが可能であることを指定します。データ保全性を確保することはユーザーの責任であり、PL/I はデータ保守の支援を行いません。

このオプションは、DDM ファイルでのみ有効です。

レコード・レベルでロックできるようにするには、SHARE(ALL) を指定し、ファイルを更新ファイルとして宣言します。CICS アプリケーションを実行するときは、この方法をお勧めします。

要求されたファイル共用レベル、あるいはデフォルトのファイル共用レベルを取得できない場合は、UNDEFINEDFILE 条件が発生します。

SKIPO

SKIPO 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。SKIPO 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。



SKIPO(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。これはデフォルトです。

SKIPO(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。

```

set dd:sysprint=stdout:,SKIPO(Y)
set dd:sysprint=con,SKIPO(Y)
  
```

TERMLBUF

TERMLBUF オプションは、PL/I Presentation Manager (PM) 端末のウィンドウの最大行数を指定します。

```
▶▶—TERMLBUF—(n)————▶▶
```

ファイルが PM 端末に関連付けられていない場合、このオプションは無視されます。デフォルトは 512 行です。

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。

```
▶▶—TYPE—(
  CRLF
  LF
  TEXT
  FIXED
  VARLS
  VARLS4X4
  VARMS
  LL
  LLZZ
  CRLFEOF
  U
)————▶▶
```

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。19 ページにある制約事項を参照してください) 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。

これは、ISAM および BTRIEVE のデフォルトです。

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。19 ページにある制約事項を参照してください) 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 CRLF 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。

VARLS

レコードの先頭にレコードの残りの部分のバイト数を指定する 2 バイトの接頭部が付き、その長さ接頭部がリトル・エンディアン・フォーマットで保持されることを示します。これらのレコードは、NATIVE CHAR VARYING スtring のようになります。

TYPE(VARLS) データ・セットを使用すれば、可変長で任意のバイト・パターンのレコードが含まれるデータ・セットを PL/I で読み書きする場合に、最も高速で処理できます。TYPE(CRLF) データ・セットでは、このような処理はできません。というのも、ビット・String '0d0a'b4 を付けて書き込まれたレコードを読み取るときに、誤変換が発生するからです。

VARLS4X4

レコードに 4 バイトの接頭部と 4 バイトの接尾部があることを示します。接頭部と接尾部にはそれぞれ、レコードの残りの部分のバイト数が入っています。このバイト数は NATIVE フォーマットです。その数には、接頭部で使用する 4 バイトも、接頭部で使用する 4 バイトも含まれません。

Type(VARLS4X4) データ・セットを使用すれば、FORTRAN の順次不定形式ファイルを処理できます。

VARMS

レコードの先頭にレコードの残りの部分のバイト数を指定する 2 バイトの接頭部が付き、その長さ接頭部がビッグ・エンディアン・フォーマットで保持されることを示します。これらのレコードは、NONNATIVE CHAR VARYING String のようになります。

TYPE(VARMS) データ・セットを使用すれば、メインフレームからダウンロードされた SCALARVARYING ファイルを読み取ることができます。

LL

レコードの先頭にレコードの合計バイト数 (接頭部を含む) を指定する 2 バイトの接頭部が付くことを示します。長さは、ビッグ・エンディアン・フォーマットで保持されます。

TYPE(LL) データ・セットを使用すれば、2 バイトを追加するツール (VRECGEN.PLI サンプル・プログラムを参照) を使ってメインフレームからダウンロードされたファイルを読み取ることができます。

LLZZ

レコードには S/390 上の可変レコードと同様に保持される 4 バイトの接頭部があることを指定します。

LLZZ サブオプションを使用すれば、TYPE(CRLF) データ・セットでは不可能な、可変長で任意のバイト・パターンのレコードが含まれるデータ・セットの読み書きが可能になります。CRLF の場合、ビット・ストリング '0d0a'b4 を付けて書き込まれたレコードは、読み取り時に誤変換されます。

TYPE(LLZZ) データ・セットには、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入れることはできません。

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 レコードがフォーマットされていないことを表します。このような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では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(CRLF) が使用されている場合、SKIP は末尾ブランクなしに、CRLF で置き換えられます。

PL/I ファイルとデータ・セットの関連付け

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

SET コマンドを使用すれば、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できます。さらにオプションで、そのデータ・セットの特性も指定できます。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。

環境変数名の形式は DD:ddname です。ここで *ddname* は、PL/I ファイル定数 (または 代替 *ddname* (後述)) の名前です。以下に例を示します。

```
declare MyFile stream output;
```

SET コマンドのオプションを指定するには、コマンド行でそれらのオプションを組み込みます。

```
set dd:myfile=c:¥datapath¥mydata.dat,APPEND(N)
```

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。

MVS の DD ステートメント
TSO の ALLOCATE ステートメント
CMS の FILEDEF コマンド

DD:ddname 環境変数で使用する構文およびオプションの詳細については、209 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。

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————▶▶
    |  
    /filespec  
    |  
    |, —dd_option—
```

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。

dd_option

209 ページの『DD:ddname 環境変数の使用による特性の指定』 DD:ddname 環境変数で許可されている 1 つまたは複数のオプション。DD:ddname 環境変数のオプションの詳細については、209 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),reclsize(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。

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) 使用しようとすると、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。この 2 つのファイルはデフォルトでそれぞれ CON 装置になります。

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
- DPATH 環境変数の中で定義されているパス

PL/I ファイルのオープンとクローズ

このトピックでは、アプリケーションが OPEN ステートメントおよび CLOSE ステートメントを実行するときの PL/I の処理について要約します。

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。情報は、以下の優先順位でマージされます。

1. OPEN ステートメントの属性
2. ファイル宣言の ENVIRONMENT オプション
3. 「I」が使用されときの OPEN ステートメントの TITLE オプションの値
4. DD:ddname 環境変数の値
5. IBM デフォルト

PL/I ファイルのオープンとクローズ

オープンするデータ・セットがワークステーション装置でないときは、DPATH 環境変数に指定されたパスに従ってデータ・セットが検索されます。このデータ・セットが検出されず、ファイルに OUTPUT 属性がある場合、データ・セットは現行ディレクトリーに作成されます。

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。

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。

複数のデータ・セットと 1 つのファイルの関連付け

PL/I ファイルは、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。TITLE オプションを使用すれば、ファイル・オープン時に複数のデータ・セットの中から 1 つを動的に選択して、特定の PL/I ファイルに関連付けることができます。次の例を見てください。

```
do Ident='A','B','C';
  open file(Master) title('/MASTER1'||Ident||'.DAT');
  .
  .
  .
  close file(Master);
end;
```

この例では、DO グループの最初の反復処理で Master がオープンされるときに、ファイルは MASTER1A.DAT という名前のデータ・セットに関連付けられます。処理が終了すると、このファイルはクローズされ、PL/I ファイル MASTER は MASTER1A.DAT データ・セットから切り離されます。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。今度は、MASTER は MASTER1B.DAT という名前のデータ・セットに関連付けられます。同様に、DO グループの最後の反復処理では、MASTER はデータ・セット MASTER1C.DAT に関連付けられます。

入出力ステートメント、属性、およびオプションの組み合わせ

次に示す図は、さまざまな PL/I ファイル操作に使用できる入出力ステートメント、ファイル属性、ENVIRONMENT オプション、および DD:ddname 環境変数オプションをリストしたものです。223 ページの表 12 はネイティブ・データ・セット用、224 ページの表 13 はワークステーション VSAM データ・セット用のリストです。

表 12. ネイティブ・データ・セット用のステートメント、属性、およびオプション

ステートメント	ファイル属性	ENVIRONMENT オプション	DD_DDNAME オプション
PUT	ENVIRONMENT FILE OUTPUT PRINT STREAM	CONSECUTIVE GRAPHIC RECSIZE(n)	AMTHD(FSYS) APPEND(YIN) ASA(YIN) file_spec RECSIZE(n) SHARE(NONE READ ALL) TERMLBUF(n) TYPE(CRLF TEXT FIXED)
GET	ENVIRONMENT FILE STREAM INPUT	CONSECUTIVE GRAPHIC RECSIZE(n)	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE READ ALL) TERMLBUF(n) TYPE(CRLF TEXT FIXED)
WRITE	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE KEYED RECORD OUTPUT UPDATE	CONSECUTIVE REGIONAL(1) CTLASA RECSIZE(n) SCALAR VARYING	AMTHD(FSYS) APPEND(YIN) file_spec RECSIZE(n) SHARE(NONE READ ALL) TERMLBUF(n) TYPE(CRLF TEXT FIXED)
LOCATE	BUFFERED ENVIRONMENT FILE KEYED RECORD OUTPUT SEQUENTIAL	CONSECUTIVE REGIONAL(1) CTLASA RECSIZE(n)	AMTHD(FSYS) APPEND(YIN) file_spec RECSIZE(n) SHARE(NONE READ ALL) TYPE(CRLF TEXT FIXED)
READ	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE INPUT UPDATE KEYED RECORD	CONSECUTIVE REGIONAL(1) RECSIZE(n) SCALAR VARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE READ ALL) TERMLBUF(n) TYPE(CRLF TEXT FIXED)
REWRITE	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	CONSECUTIVE REGIONAL(1) RECSIZE(n) SCALAR VARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE READ ALL) TYPE(CRLF TEXT FIXED)
DELETE	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	REGIONAL(1) RECSIZE(n) SCALAR VARYING	AMTHD(FSYS) file_spec RECSIZE(n) SHARE(NONE READ ALL)

注:

- ¹ 新規データ・セットを作成する場合
- ² プリンター向け PL/I ファイルの場合
- ³ PM 端末に関連付けられる場合
- ⁴ データ・セットが PL/I プログラムで作成されていない場合
- ⁵ DIRECT は REGIONAL(1) にのみ適用できる
- ⁶ REGIONAL(1) 用
- ⁷ REGIONAL(1) には適用されない

ステートメント、属性、オプション

表 13. ワークステーション VSAM データ・セット用のステートメント、属性、およびオプション

ステートメント	ファイル属性	ENVIRONMENT オプション	DD_DDNAME オプション
PUT	ENVIRONMENT FILE OUTPUT PRINT STREAM	ORGANIZATION(CONSECUTIVE) GRAPHIC RECSIZE(n)	AMTHD(DDMIISAMIBTRIEVE) APPEND(YIN) ASA(YIN) file_spec RECSIZE(n) SHARE(NONE READ ALL)
GET	ENVIRONMENT FILE STREAM INPUT	ORGANIZATION(CONSECUTIVE) GRAPHIC RECSIZE(n)	AMTHD(DDMIISAMIBTRIEVE) file_spec RECSIZE(n) SHARE(NONE READ ALL)
WRITE	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE KEYED RECORD OUTPUT UPDATE	ORGANIZATION VSAM CTLASA RECSIZE(n) SCALARVARYING	AMTHD(DDMIISAMIBTRIEVE) ASA(YIN) APPEND(YIN) file_spec RECSIZE(n) SHARE(NONE READ ALL)
LOCATE	BUFFERED ENVIRONMENT FILE KEYED RECORD OUTPUT SEQUENTIAL	ORGANIZATION VSAM CTLASA RECSIZE(n) SCALARVARYING	AMTHD(DDMIISAMIBTRIEVE) APPEND(YIN) file_spec RECSIZE(n) SHARE(NONE READ ALL)
READ	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE INPUT UPDATE KEYED RECORD	ORGANIZATION VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDMIISAMIBTRIEVE) file_spec RECSIZE(n) SHARE(NONE READ ALL)
REWRITE	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	ORGANIZATION VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDMIISAMIBTRIEVE) file_spec RECSIZE(n) SHARE(NONE READ ALL)
DELETE	BUFFERED UNBUFFERED DIRECT SEQUENTIAL ENVIRONMENT FILE UPDATE KEYED RECORD	ORGANIZATION VSAM RECSIZE(n) SCALARVARYING	AMTHD(DDMIISAMIBTRIEVE) file_spec RECSIZE(n) SHARE(NONE READ ALL)

注:

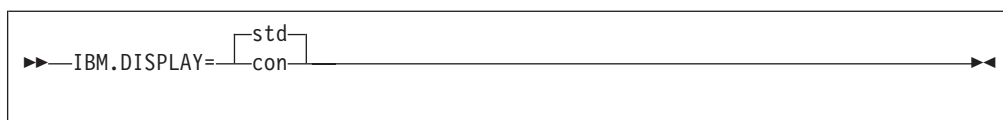
¹ 新規データ・セットを作成する場合

² プリンター向け PL/I ファイルの場合

³ VSAM データ・セットには適用されない

DISPLAY ステートメントの入出力

DISPLAY の REPLY は、stdin から読み取られます。DISPLAY ステートメントからの出力は、デフォルトで stdout に送信されます。IBM.DISPLAY 環境変数の構文は、次のとおりです。

**std**

DISPLAY ステートメントを標準出力装置に関連付けることを指定します。これはデフォルトです。

con

DISPLAY ステートメントを CON 装置に関連付けることを指定します。

表示ステートメントをファイルにリダイレクトすることができます。以下に例を示します。

```
set ibm.display=std
```

```
Hello: proc options(main);
  display('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。

```
hello > hello1.out
```

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 HELLO1.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 HELLO1.OUT に書き込まれることを意味します。

PL/I 標準ファイル (SYSPRINT と SYSIN)

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 オプションを使用するか、あるいはデータ・セットまたは別の装置の名前を指定した DD:ddname 環境変数を設定する必要があります。

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

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。ただし、このリダイレクトを機能させるには、最初に 2 つの SET DD: ステートメントを発行する必要があります。その記述は、以下のとおりです。

```
set dd:sysprint=stdout:
set dd:sysin=stdin:
```

```
Hello: 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 >
```

```
hello > hello2.out
```

リダイレクト装置

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できますが、その場合は、値が 1 である LRECL を指定しなければなりません。

第 14 章 連続データ・セットの定義と使用

プリンター向けファイル	227	データのフォーマット	240
ストリーム指向データ伝送の使用	228	ストリーム・ファイルおよびレコード・ファイル	241
ストリーム入出力を用いたファイルの定義	229	大文字と小文字	241
ストリーム指向データ伝送用 ENVIRONMENT		ファイルの終わり	242
オプション	229	コンソールへの出力の制御	242
ストリーム入出力によるデータ・セットの作成	229	PRINT ファイルのフォーマット	242
必須情報	230	ストリーム・ファイルおよびレコード・ファイル	242
例	230	対話式プログラムの例	243
ストリーム入出力によるデータ・セットへのアク		レコード単位入出力の使用	244
セス	232	レコード入出力の使用によるファイルの定義	246
必須情報	233	レコード単位データ伝送用 ENVIRONMENT オ	
例	233	プション	246
PRINT ファイルの使用	234	レコード入出力によるデータ・セットの作成	246
印刷する行の長さの制御	235	必須情報	246
タブ制御テーブルの指定変更	237	レコード入出力によるデータ・セットへのアクセ	
SYSIN ファイルおよび SYSPRINT ファイルの		スと更新	247
使用方法	239	必須情報	248
コンソールからの入力 of 制御	239	連続データ・セットの例	248
会話型のファイルの使用	240		

以下のセクションでは、連続データ・セットの編成と、連続データ・セットの作成方法、アクセス方法、および更新方法を説明します。

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。すなわち、データ・セットが作成されるときに、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。

本章の情報は、ネイティブ・データ・セットまたは DDM データ・セットのいずれかに関連付けられた、ENVIRONMENT 属性の CONSECUTIVE オプションを使用するファイルに適用されます。PL/I Presentation Manager ネイティブ・データ・セットのみをサポートします。

プリンター向けファイル

プリンター向けファイルは、PRINT 属性を持つ PL/I ファイルであり、ENVIRONMENT 属性の CTLASA オプションを用いて宣言されたレコード・ファイルです。ワークステーションでこれらのファイルを印刷するか、またはメインフレームにアップロードすることができます。

各レコードの先頭文字は米国標準規格 (ANS) の紙送り制御文字です (228 ページの表 14 参照)。

STREAM ファイルの場合、PL/I は、PUT ステートメントの SKIP、LINE、または PAGE オプション (または制御フォーマット項目) に基づいて文字を挿入します。CTLASA を含む RECORD ファイルの場合、プログラムは各レコードの先頭のバイトに制御文字を挿入する必要があります。

ワークステーションからデータ・セットを印刷したい場合は、ASA(N) オプションを選択します (これがデフォルトです)。メインフレームでの印刷用のフォーマットを保持するには、ASA(Y) を選択します。これにより制御文字は変換されずにそのまま残ります。

表 14. ANS 印刷制御文字

文字	意味
(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。
ハイフン (-)	3 行スキップしてから印刷する。
+	1 行もスキップせずに印刷する。
1	次のページへスキップしてから印刷する。
2	3 行スキップしてから印刷する。
3	3 行スキップしてから印刷する。
4	3 行スキップしてから印刷する。
5	3 行スキップしてから印刷する。
6	3 行スキップしてから印刷する。
7	3 行スキップしてから印刷する。
8	3 行スキップしてから印刷する。
9	3 行スキップしてから印刷する。
A	3 行スキップしてから印刷する。
B	3 行スキップしてから印刷する。
C	3 行スキップしてから印刷する。

IBM Proprinter 制御文字への変換は、次のように行われます。

表 15. ANS 制御文字に対応する IBM Proprinter 制御文字

ANS 文字	Proprinter 文字 (16 進数)
(ブランク)	0A
0	0A 0A
-	0A 0A 0A
+	0D
1	0C
2 ~ 9、A ~ C	0A 0A 0A

注: ここで、
0A = 改行
0C = 用紙送り
0D = 復帰

リストの最初の 5 文字のみが PL/I によって変換されます。他の文字はハイフン (-) として処理されます。

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。また、DD:ddname 環境変数でデータ・セットの作成およびアクセス時に使用する必須パラメーターをまとめ、PL/I プログラムの例もいくつか掲載しています。

STREAM 属性をもつデータ・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、レコードの境界を無視して、各データ・セットを、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。データ値は、文字フォーマットまたはグラフィック・フォーマットになっています。グラフィック・フォーマットは、すなわち、DBCS (2 バイト文字セット) 形式です。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フォーマットに変換し、文字または DBCS のストリームを、データ・セットへ伝送する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てのに適した形に変換します。

ストリーム指向データ伝送は、DBCS データ (グラフィック) の読み取りと書き込みに使用できます。適切な装置が DBCS をサポートしていれば、DBCS データを入力、表示、および印刷できます。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。

ストリーム入出力を用いたファイルの定義

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。

```
declare
  Filename file stream
    input | {output [print]}
    environment(options);
```

FILE 属性については、「PL/I 言語解説書」で説明されています。PRINT 属性の詳細については、234 ページの『PRINT ファイルの使用』に説明があります。

ストリーム指向データ伝送用 ENVIRONMENT オプション

ストリーム指向データ伝送で使える ENVIRONMENT オプションは、次のとおりです。

- CONSECUTIVE
- RECSIZE
- GRAPHIC
- ORGANIZATION(CONSECUTIVE)

これらのオプションおよび構文については、202 ページの『PL/I ENVIRONMENT 属性の使用による特性の指定』で説明されています。

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、以下のいずれか 1 つを使用します。

- ENVIRONMENT 属性
- DD:ddname 環境変数
- OPEN ステートメントの TITLE オプション

TITLE オプションについて詳しくは、220 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。

必須情報

ユーザーのアプリケーションが STREAM ファイルを作成する場合は、アプリケーションはそのファイルの行サイズの値を次の入手源の 1 つから選んで与える必要があります。

- 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

値を与えないと、PL/I のデフォルト値が用いられます。LINESIZE オプションを選択すると、他の入手源からの値がすべて指定変更されます。ENVIRONMENT 属性の RECSIZE オプションは、他の RECSIZE オプションを指定変更します。OPEN ステートメントの TITLE オプションに指定した RECSIZE の方が、DD:ddname 環境変数の RECSIZE に指定した値よりも優先されます。

例えば、LINESIZE の値を与えずに RECSIZE の値だけを与えると、PL/I は次のように RECSIZE の値から行サイズの値を導き出します。

- ASA(N) オプションが適用された PRINT ファイルでは、RECSIZE 値は 4 です。
- ASA(Y) オプションが適用された PRINT ファイルでは、RECSIZE 値は 1 です。
- 上記以外の場合は、RECSIZE の値が行サイズの値に割り当てられます。

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、PL/I は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 属性をもっている。この場合、値はタブ制御テーブルから得られます (238 ページの図 11 参照)。
- 関連データ・セットが端末 (CON:、STDOUT:、または STDERR:) である。この場合、行サイズの値は 120 です。

PL/I は、行サイズの値からデータ・セットのレコード長を導き出します。また、レコード長の値は、次のように行サイズの値から導き出されます。

- ASA(N) オプションが適用された PRINT ファイルでは、値は行サイズ + 4 です。
- ASA(Y) オプションが適用された PRINT ファイルでは、値は行サイズ + 1 です。
- 上記以外の場合は、行サイズの値がレコード長の値に割り当てられます。

例

ストリーム指向データ伝送を使用して連続データ・セットを作成する方法が、232 ページの図 8 に示されています。最初に、数人の名前と誕生日のリストを含むデー

タ・セット BDAY.INP からデータが読み取られます。次に、誕生日が 10 月である人の名前と誕生日を含む連続データ・セット BDAY.OCT が書き込まれます。

ディスク・ファイル BDAY.INP を入力データ・セットと関連付けるには、コマンド SET DD:SYSIN=BDAY.INP を使用してください。PL/I プログラムによってこのファイルが作成されなかった場合、RECSIZE オプションも指定する必要があります。

連続出力ファイル WORK をディスク・データ・セット BDAY.OCT と関連付けるには、コマンド SET DD:WORK=BDAY.OCT を使用してください。

```

/*****
/*
/* DESCRIPTION
/* Create a CONSECUTIVE data set with 30-byte records containing
/* names and birthdays of people whose birthdays are in October.
/*
/* USAGE
/* The following commands are required to establish
/* the environment variables to run this program:
/*
/* SET DD:WORK=BDAY.OCT
/* SET DD:SYSIN=BDAY.INP,RECSIZE(80)
*****/

BDAY: proc options(main);

    dcl Work file stream output,
        1 Rec,
            3 Name char(19),
            3 BMonth char(3),
            3 Pad1 char(1),
            3 BDate char(2),
            3 Pad2 char(1),
            3 BYear char(4);

    dcl Eof bit(1) init('0'b);
    dcl In char(30) def Rec;

    on endfile(sysin) Eof='1'b;

    open file(Work) linesize(400);
    get file(sysin) edit(In)(a(30));
    do while (~Eof);
        if BMonth = 'OCT'
            then put file(Work) edit(In)(a(30));
        else;
            get file(sysin) edit(In)(a(30));
        end;
        close file(Work);
    end BDAY;

BDAY.INP contains the input data used at execution time:

```

```

LUCY  D.      MAR 15 1950
REGINA W.     OCT 09 1971
GARY   M.     DEC 01 1964
PETER  T.     MAY 03 1948
JANE   K.     OCT 24 1939

```

図8. ストリーム指向データ伝送によるデータ・セットの作成

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。ただし、CONSECUTIVE 編成データ・セットでなければならず、また、データ・セット内の全データが文字またはグラフィックの形でなければなりません。入力用の関連ファイルをオープン

し、データ・セットに含まれる項目を読み取るか、あるいは出力用のファイルを開き、終わりに項目を追加してデータ・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

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
- データ・セットの拡張属性
- 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、229 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 (CON:, SCREEN\$, STDOUT:, または STDERR:) に関連付けられており、値が 120 の場合

例

234 ページの図 9 のプログラムは、232 ページの図 8 のプログラムで作成したデータを読み取り、データ・セット SYSPRINT を使用してそのデータを表示します。SYSPRINT データ・セットは CON 装置に関連付けられているため、プログラム実行の前に関連付けが解除されない場合、出力は画面上に表示されます。

(SYSPRINT の詳細については、239 ページの『SYSIN ファイルおよび SYSPRINT ファイルの使用方法』を参照してください。)

```
/******  
/*  
/* DESCRIPTION  
/*   Read a CONSECUTIVE data set and print the 30-byte records  
/*   to the screen.  
/*  
/* USAGE  
/*   The following command is required to establish  
/*   the environment variable to run this program:  
/*  
/*       SET DD:WORK=BDAY.OCT  
/*  
/*   Note: This sample program uses the CONSECUTIVE data set  
/*         created by the previous sample program BDAY.  
/*  
/******  
  
BDAY1: proc options(main);  
  
    dcl Work file stream input;  
  
    dcl Eof bit(1) init('0'b);  
  
    dcl In char(30);  
  
    on endfile(Work) Eof='1'b;  
  
    open file(Work);  
    get file(Work) edit(In)(a(30));  
    do while (~Eof);  
        put file(sysprint) skip edit(In)(a);  
        get file(Work) edit(In)(a(30));  
    end;  
    close file(Work);  
end BDAY1;
```

図9. 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

PRINT ファイルの使用

PL/I プログラムでは、PRINT ファイルを使用すると、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ために便利です。PL/I は、PAGE、SKIP、LINE の各オプション、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。

PL/I は、PRINT ファイルによって伝送される各レコードの最初のバイトを米国標準規格 (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 (227 ページの『プリンター向けファイル』参照)。

PL/I は、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 オプションで 4 行以上のスペースが指定されている場合は、PL/I は、適切な制御文字を使用し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。

PRINT ファイルの伝送先が端末装置の場合は、出力フォーマットを指定しない限り、PAGE、SKIP、LINE の各オプションによって 3 行を超えてスキップされることはありません。

印刷する行の長さの制御

次のいずれかの方法によって、PRINT ファイルが生成した印刷行の長さを制限することができます。

- ENVIRONMENT 属性の RECSIZE オプションを使用して、PL/I プログラム内でレコード長を指定する。
- LINESIZE オプションを使用して、OPEN ステートメント内で行サイズを指定する。
- RECSIZE オプションを使用して、OPEN ステートメントの TITLE オプションでレコード長を指定する。

RECSIZE 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、RECSIZE は印刷される行の長さより 1 バイト長くなければなりません。LINESIZE は、印刷される行の文字数を参照します。PL/I は印刷制御文字を追加します。

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ることによって、実行中にファイルの行サイズを変更しないでください。

PRINT ファイルは 120 文字のデフォルトの行サイズをもっています。したがって、PRINT ファイルのレコード長を指定する必要はありません。

例: 236 ページの図 10 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。

```

/*****
/*
/* DESCRIPTION
/* Create a SEQUENTIAL data set.
/*
/* USAGE
/* The following command is required to establish
/* the environment variable to run this program:
/*
/* SET DD:TABLE=MYTAB.DAT,ASA(Y)
/*
/*
*****/

SINE: proc options(main);

/* Build a table of SINE values.
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;
dcl I          fixed dec(2);

on error
begin;
on error system;
display ('oncode = '|| Oncode);
end;

```

図 10. ストリーム・データ伝送による印刷ファイルの作成 (1/2) : (252 ページの図 15 の例によってこのファイルが印刷されます)

```

on endpage(Table)
begin;
  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);

      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(102);
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 10. ストリーム・データ伝送による印刷ファイルの作成 (2/2): (252 ページの図 15 の例によってこのファイルが印刷されます)

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。

252 ページの図 15 のプログラムは、レコード単位データ伝送を使って、図 10 のプログラムが作成するテーブルを印刷します。

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、あらかじめ設定されているタブ位置に合わせて配置されます。タブ位置は、PL/I が定義するタブ制御テーブル内で定義されます。タブ制御テーブルは、PLITABS という名前の外部構造体です。238 ページの図 11 に PLITABS の宣言を示します。

```

dc1 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 11. *PLITABS* の宣言：(標準ページ・サイズ、行サイズ、およびタブ位置を設定しています)

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

PLITABS の先頭からの *Number_of_tabs* のオフセットで表された 2 進整数。
Number_of_tabs 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 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 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 2 進整数。値が 0 の場合は、不定形式出力となります。

Fill1、Fill2、Fill3

3 の 2 進整数。将来の利用のために予約済み。

Number_of_tabs

テーブル内のタブ位置エントリーの数 (最大 255) を定義する 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 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。

リンカーを用いて *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。この変更は、メインルーチンが含まれるソース・プログラム内に、*PLITABS* という名前と *EXTERNAL STATIC* という属性を持つ *PL/I* 構造体を記述することによって実行します。

図 12 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。

```

dc1 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 12. 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

FILE オプションを使用しないで GET または PUT ステートメントをコーディングした場合、PL/I は文脈に従って、それぞれファイルが SYSIN および SYSPRINT であることを前提とします。

SYSPRINT を宣言しないと、PL/I 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。完全な属性セットは次のようになります。

```
file stream print external
```

SYSPRINT は PRINT ファイルの一種であるため、ファイルのオープン時にはデフォルトの行サイズ (120 文字) が適用されます。

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、PL/I が SYSPRINT に与えた属性を指定変更することができます。ただし、SYSPRINT が STREAM OUTPUT ファイルとして宣言またはオープンされた場合は、INTERNAL 属性も宣言されている場合を除き、デフォルトで PRINT 属性が適用されます。

PL/I は入力ファイル SYSIN のために特別な属性は提供しません。宣言を行わなければ、SYSIN はデフォルトの属性のみを受け取ります。

コンソールからの入力の制御

入力ファイル用のデータを入力するには、以下の操作を両方実行してください。

- 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)。
- 入力ファイルを端末に割り振る。

コンソールからの入力の制御

ユーザーは通常、標準のデフォルト入力ファイル `YSIN` 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、コンソール装置に割り振ることができるからです。

`PROMPT(Y)` を指定すると、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が、コロン (:) で示されます。213 ページの『`PROMPT`』を参照してください。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。入力されたデータが `GET` ステートメントの実行を完了するために十分出ない場合、さらにプロンプトが表示されます。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。

`PROMPT(Y)` を指定していない場合は、デフォルトでは行の先頭にコロンが表示されません。

継続させたい行の最後にハイフンを付け加えると、もう 1 行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。ハイフンは、明示的な継続文字です。

ユーザーのプログラムで、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の末尾をコロンにすることによって、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。

- 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。
- 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。`COPY` オプションを使用して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。

会話型のファイルの使用

プログラムがユーザーと会話形式でやり取りできるようにするためには、プログラム内でコンソールを連続ファイルの入出力装置として使用します。会話型入出力は特別な `PL/I` コードを必要としないため、任意のストリーム・ファイルを会話型で使うことができます。

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ありません。`PL/I` は各行の終わりにコンマを挿入します。

例として、次のようなステートメントを見てみましょう。

```
get list(I,J,K);
```

各項目の後に ENTER キーを押して、次の応答を返すことができます。
(PROMPT(Y) を指定した場合のみコロンが表示されます。)

```
:
1
:
2
:
3
```

別々の行にデータを入力するのは、次のように指定するのと同事です。

```
:
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 (ハイフン) 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ます。GET EDIT ステートメントでは、項目への埋め込みが行なわれます。

- *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。

次の PL/I ステートメントを見てください。

```
get edit(Name)(a(15));
```

例えば次の 5 文字を入力し、直後に ENTER を入力します。

```
SMITH
```

プログラムが 15 文字からなるストリングを受け取るようにするため、この項目には 10 個のブランクが埋め込まれます。ある項目を後続の行に続けたければ、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。連結文字を入れなければ、伝送される最初の行に埋め込み処理が行われ、その行が完結したデータ項目として扱われます。

- *SKIP* オプションまたはフォーマット項目: GET ステートメント内で SKIP を使用すると、まだ入力されていないデータを無視します。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せん。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、ファイルの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。

大文字と小文字

ストリーム・ファイルとレコード・ファイルのどちらの場合も、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。

ファイルの終わり

桁 1 および桁 2 に /* があり、ほかの文字を含まない行は、ファイル・マークとして扱われ、ENDFILE 条件を発生させます。

コンソールへの出力の制御

画面上に、次の条件を両方とも満たす PL/I ファイルのデータを表示することができます。

- 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。
- 端末に割り振られている (CON:、STDOUT:、SCREEN\$:、または STDERR:)。

標準の出力ファイル 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、端末で列および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、復帰によってインプリメントされます。

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 という名前でなければならず、また、テーブルの内容は 237 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。標準レイアウトと異なる場合は、238 ページの図 11 の PLITABS の説明を参照してください。PLITABS を使って、リスト指示およびデータ指示出力のタブ位置を変更することもできます。

リスト指示出力およびデータ指示出力のタブは、ブランク (スペース) 文字の伝送によって実現されます。

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。

ストリーム・ファイルおよびレコード・ファイルの場合、端末で表示される文字は、プログラムに入っているとおりに表示されます。大文字と小文字の両方が表示可能です。

対話式プログラムの例

244 ページの図 13 のプログラム例では、ユーザーとのダイアログを使用して連続データ・セット PHONES を作成します。デフォルトでは、SYSIN が CON 装置に関連付けられています。SYSIN ファイル用の環境変数を設定するか、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って、この関連付けを指定変更することができます。出力データ・セットは、ディスク・ファイル INT1.DAT に関連付けられ、ユーザーがキーボードから入力した名前と電話番号を格納します。

```
/* **** */
/*
/* DESCRIPTION
/* Create a SEQUENTIAL data set using a console dialog.
/*
/* USAGE
/* The following command is required to establish
/* the environment variable to run this program:
/*
/* SET DD:PHONES=INT1.DAT,APPEND(Y)
/*
/* **** */

INT1: proc options(main);

    dcl Phones stream env(recsize(40));

    dcl Eof bit(1) init('0'b);

    dcl 1 PhoneBookEntry,
        3 NameField char(19),
        3 PhoneNumber char(21);
    dcl InArea char(40);

    open file (Phones) output;

    on endfile(sysin) Eof='1'b;

    /* start creating phone book */
    put list('Please enter name:');
    get edit(NameField)(a(19));
    if ~Eof then
    do;
        put list('Please enter number:');
        get edit(PhoneNumber)(a(21));
    end;
    do while (~Eof);
        put file(Phones) edit(PhoneBookEntry)(a(40));
        put list('Please enter name:');
        get edit(NameField)(a(19));
        if ~Eof then
        do;
            put list('Please enter number:');
            get edit(PhoneNumber)(a(21));
        end;
    end;

    close file(Phones);

end INT1;
```

図 13. 対話式プログラムのサンプル

レコード単位入出力の使用

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。このセクションでは、連続データ・セットを用いてレコード単位入出力を使用する方法について説明します。

245 ページの表 16 は、レコード単位入出力を使って、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。

DDM 直接データ・セットまたはキー・データ・セットに関連付けられた CONSECUTIVE ファイルは、INPUT 用にのみオープンすることができます。このようなファイルを OUTPUT または UPDATE 用にオープンしようとすると、PL/I は UNDEFINEDFILE を発生させます。

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、 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 based-variable FILE(file-reference);	SET(pointer reference)
SEQUENTIAL OUTPUT UNBUFFERED	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 UNBUFFERED	READ FILE(file-reference) INPUT(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 UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	

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 (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
---------------------	--	--------------------

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。

```
declare
  Filename file record
    input | output | update
    sequential
    buffered | unbuffered
    environment(options);
```

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。

レコード単位データ伝送用 ENVIRONMENT オプション

レコード単位データ伝送用の連続データ・セットに適用できる ENVIRONMENT オプションは、次のとおりです。

- CONSECUTIVE
- CTLASA
- ORGANIZATION(CONSECUTIVE)
- RECSIZE
- SCALARVARYING

これらのオプションおよび構文については、202 ページの『PL/I ENVIRONMENT 属性の使用による特性の指定』で説明されています。

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 の関連ファイルをオープンする必要があります。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・セットを作成するには、ENVIRONMENT 属性、DD:ddname 環境変数、または OPEN ステートメントの TITLE オプションのいずれかで、PL/I に特定の情報を与える必要があります。

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（221 ページの『データ・セットに関連付けられていないファイルの使用の試み』参照）。
- 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 オプションのいずれかの 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 (CON:、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。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については、249 ページの図 14 を参照してください。

出力用のファイルをオープンし、その終わりにレコードを追加してデータ・セットを拡張したい場合、DD:ddname 環境変数内に APPEND(Y) を指定する必要はありません。これがデフォルトであるためです。APPEND(N) を指定すると、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、その既存の順序のままレコードを更新することのみが可能であり、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。既存のデータ・セットのレコード長を変更することはできません。

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、検索された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、OPEN ステートメントの TITLE オプションまたは DD:ddname 環境変数を用いて、PL/I に対してデータ・セットを識別する必要があります。

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。

必須情報

ユーザー・アプリケーションで既存の RECORD 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。

- 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

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 である。この場合、使用される値は 80 です。
- ファイルが端末に関連付けられている。この場合、使用される値は 120 です。

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、249 ページの図 14 のプログラムに示されています。このプログラムでは、INPUT1 および INPUT2 という 2 つの PL/I ファイルの内容をマージし、OUT という新規 PL/I ファイルに書き込んでいます。INPUT1 および INPUT2 は、それぞれディスク・ファイル EVENS.INP および ODDS.INP に関連付けられており、ASCII 照合シーケンスに配置された 6 バイトのレコードを格納しています。

```

/*****
/*
/* DESCRIPTION
/* Merge 2 data sets creating a CONSECUTIVE data set.
/*
/* USAGE
/* The following commands are required to establish
/* the environment variables to run this program:
/*
/* SET DD:OUT=CON4.DAT
/* SET DD:INPUT1=EVENS.INP
/* SET DD:INPUT2=ODDS.INP
/*
*****/

MERGE: proc options(main);

    dc1 Input1 file record sequential input env(recsize(6));
    dc1 Input2 file record sequential input env(recsize(6));
    dc1 Out file record sequential env(recsize(15));
    dc1 Sysprint file print; /* normal print file */

    dc1 Input1_Eof bit(1) init('0'b); /* eof flag for Input1 */
    dc1 Input2_Eof bit(1) init('0'b); /* eof flag for Input2 */
    dc1 Out_Eof bit(1) init('0'b); /* eof flag for Out */
    dc1 True bit(1) init('1'b); /* constant True */
    dc1 False bit(1) init('0'b); /* constant False */

    dc1 Item1 char(6) based(a); /* item from Input1 */
    dc1 Item2 char(6) based(b); /* item from Input2 */
    dc1 A pointer; /* pointer var */
    dc1 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),
        file(Input2),
        file(Out) output;

    read file(Input1) set(A); /* priming read */
    read file(Input2) set(B);

```

図 14. 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 (1/3)

```
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;

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(Item1);              /* display Out file */
do while (Out_Eof = False);
  put file(Sysprint) skip edit(Item1) (a);
  read file(Out) into(Item1);
end;
close file(Out);

end MERGE;
```

図 14. 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 (2/3)

Here is a sample of EVENS.INP:

BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ

Here is a sample of ODDS.INP:

AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
KKKKKK

図 14. 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 (3/3)

図 15 のプログラムは、レコード単位データ伝送を使って、236 ページの図 10 のプログラムが作成するテーブルを印刷します。

```

/*****
/*
/* DESCRIPTION
/*   Print a SEQUENTIAL data set created by the SINE program.
/*
/* USAGE
/*   The following commands are required to establish
/*   the environment variables to run this program:
/*
/*       SET DD:TABLE=MYTAB.DAT
/*       SET DD:PRINTER=PRN
/*
*****/

PRT: proc options(main);

    dcl Table      file record input sequential;
    dcl Printer    file record output seql
                  env(recsize(200) ctlasa);
    dcl Line       char(102) 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);
        if Line='' then                     /* insert blank lines */
            Line= ' ';
            write file(Printer) from(Line);
            read file(Table) into(Line);
        end;

        close file(Table),
            file(Printer);
    end PRT;

```

図 15. レコード単位データ伝送の印刷

第 15 章 領域データ・セットの定義と使用

領域データ・セット用のファイルの定義	255	REGIONAL(1) データ・セットの作成	257
ENVIRONMENT オプションの指定	256	例	257
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	256	REGIONAL(1) データ・セットへのアクセスと更新	259
領域データ・セットでのキーの使用	256	新	259
REGIONAL(1) データ・セットの使用	256	順次アクセス	259
ダミー・レコード	257	直接アクセス	260
		例	260

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。領域データ・セットの作成および領域データ・セットへのアクセスについても説明します。

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 1 つ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した番号が付けられ、データ伝送ステートメント内に領域番号を指定することによって、レコードに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、また、データ・アクセス時間を最適化することができます。連続編成では、連続したレコードは厳密に物理的順序で書き込まれるため、このタイプの最適化を行うことはできません。

領域データ・セットは、連続データ・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な順序で提示し、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 属性をも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、任意の順序でレコードを検索、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。

PL/I は REGIONAL(1) データ・セットをサポートします。REGIONAL(1) データ・セットの作成またはアクセスのために使用可能なデータ伝送ステートメントおよびオプションのリストについては、254 ページの表 17 を参照してください。

表 17. 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 BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression); LOCATE based-variable FROM(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression);	KEYTO(reference) KEYTO(reference)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression);	KEYTO(reference)
SEQUENTIAL UPDATE ³ BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference);	KEYTO(reference) KEYTO(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) IGNORE(expression); REWRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

表 17. 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 INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	
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。

```
declare
  Filename file record
    input | output | update
    sequential
    buffered | unbuffered
    [keyed]
    environment(options);
```

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。

```
declare
  Filename file record
    input | output | update
```

```
direct
unbuffered
[keyed]
environment(options);
```

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 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)
RECSIZE
SCALARVARYING
```

これらのオプションについては、202 ページの『*PL/I ENVIRONMENT 属性の使用による特性の指定*』で説明されています。

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、ENVIRONMENT 属性または DD:ddname 環境変数のいずれかで、PL/I に特定の情報を与える必要があります。

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。

- PL/I ファイルに関連付けられたデータ・セットの名前。REGIONAL(1) 編成のデータ・セットは、直接アクセス記憶装置上のみに存在できます (221 ページの『データ・セットに関連付けられていないファイルの使用の試み』参照)。
- 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 オプションのいずれかの RECSIZE オプションを使って指定することができます。
- データ・セットのエクステンツ (領域の数)。DD:ddname 環境変数の RECCOUNT オプションを用いて指定します。

RECCOUNT のデフォルトは 50 です。

領域データ・セットでのキーの使用

ソース・キーは、REGIONAL(1) データ・セットにアクセスするために使用されます。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・キーは領域番号です。

REGIONAL(1) データ・セットの使用

REGIONAL(1) データ・セットでは、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・キーの文字値は符号なし 10 進整数を表し、その値は 2147483647 を超えないようにする必要があります。領域番号がこの数値を超える場合、領域番号はモジュロ 2147483648 として扱われ、例えば、2147483658 は 10 として扱われます。

0 から 9 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。領域番号に埋め込みブランクを使用することはできません。したがって、埋め込み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 10 文字以上存在する場合は、右側の 10 文字のみが領域番号として使用され、10 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 X'FF' 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は無視されません。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する必要があります。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。

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。254 ページの表 17 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。

データ・セット作成時に、ファイルをオープンすると、データ・セットにダミー・レコードが埋められます。レコードは、SEQUENTIAL OUTPUT ファイル用に領域番号の昇順で提示する必要があります。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。DIRECT OUTPUT ファイルを使用してデータ・セットを作成するとレコードをランダム順で提示できます。重複する領域番号を提示した場合は、既存のレコードが上書きされます。

バッファ付きファイルを使ってデータ・セットを作成し、ファイルのクローズの前の最後の WRITE または LOCATE ステートメントで、そのデータ・セットの限界を超えてレコードを伝送しようとする、CLOSE ステートメントで ERROR 条件が生じることがあります。

例

REGIONAL(1) データ・セットの作成例が、258 ページの図 16 に示してあります。この例のデータ・セットは、内線電話番号とその電話番号を割り当てる加入者の氏名のリストです。内線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。

```
/* **** */
/* DESCRIPTION */
/* Create a REGIONAL(1) data set. */
/* USAGE */
/* The following commands are required to establish */
/* the environment variables to run this program: */
/* SET DD:SYSIN=CRG.INP,RECSIZE(30) */
/* SET DD:NOS=NOS.DAT,RECCOUNT(100) */
/* **** */
CRR1: proc options(main);

    dcl Nos file record output direct keyed
        env(regional(1) recsize(20));

    dcl Sysin file input record;
    dcl 1 In_Area,
        2 Name char(20),
        2 Number char( 2);
    dcl IoField char(20);
    dcl Sysin_Eof bit (1) init('0'b);
    dcl Ntemp fixed(15);
    on endfile (Sysin) Sysin_Eof = '1'b;
    open file(Nos);
    read file(Sysin) into(In_Area);
    do while(~Sysin_Eof);
        IoField = Name;
        Ntemp = Number;
        write file(Nos) from(IoField) keyfrom(Ntemp);
        put file(sysprint) skip edit (In_Area) (a);
        read file(Sysin) into(In_Area);
    end;
    close file(Nos);
end CRR1;
```

図 16. REGIONAL(1) データ・セットの作成 (1/2)

The execution time input file, CRG.INP, might look like this:

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 16. REGIONAL(1) データ・セットの作成 (2/2)

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。254 ページの表 17 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 属性をもつことができます。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 10 文字を超えると、返される値 (10 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 10 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・セットの使用

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。READ ステートメントおよび REWRITE ステートメントの使用については、247 ページの『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』で説明されています。

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・キーをも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、261 ページの図 17 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。

```

/*****
/*
/* DESCRIPTION
/*   Update a REGIONAL(1) data set.
/*
/* USAGE
/*   The following commands are required to establish
/*   the environment variables to run this program:
/*
/*       SET DD:SYSIN=ACR.INP,RECSIZE(30)
/*       SET DD:NOS=NOS.DAT,APPEND(Y)
/*
/*   Note: This sample program is using the regional data set,
/*         NOS.DAT, created by the previous sample program CRR1.
/*
*****/

ACR1: proc options(main);

    dcl Nos file record keyed env(regional(1));
    dcl Sysin file input record;
    dcl Sysin_Eof bit (1) init('0'b);
    dcl Nos_Eof bit (1) init('0'b);
    dcl 1 In_Area,
        2 Name char(20),
        2 (CNewNo,COldNo) char( 2),
        2 In_Area_1 char( 1),
        2 Code char( 1);
    dcl IoField char(20);
    dcl Byte char(1) def IoField;
    dcl NewNo fixed(15);
    dcl OldNo fixed(15);

    on endfile (Sysin) Sysin_Eof = '1'b;
    open file (Nos) direct update;
    read file(Sysin) into(In_Area);

```

図 17. REGIONAL(1) データ・セットの更新 (1/3)

```

do while(~Sysin_Eof);
  if CNewNo ^= ' ' then
    NewNo = CNewNo;
  else
    NewNo = 0;
  if COldNo ^= ' ' then
    OldNo = COldNo;
  else
    OldNo = 0;
  select(Code);
  when('A','C')
  do;
    if Code = 'C' then
      delete file(Nos) key(OldNo);
    read file(Nos) key(NewNo) into(IoField);
    /* we must test to see if the record exists */
    /* if it doesn't exist we create a record there */
    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(In_Area);
close file(Sysin),file(Nos);
put file(sysprint) page;
open file(Nos) sequential input;
on endfile (Nos) nos_Eof = '1'b;
read file(Nos) into(IoField) keyto(CNewNo);
do while(~Nos_Eof);
  if unspec(Byte) ^= (8)'1'b then
    put file(sysprint) skip
      edit (CNewNo,' ',IoField)(a(2),a(1),a);
  read file(Nos) into(IoField) keyto(CNewNo);
end;
close file(Nos);
end ACR1;

end;

```

図 17. REGIONAL(1) データ・セットの更新 (2/3)

At execution time, the input file, ACR.INP, could look like this:

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	X
ROLF,D.F.	14	D
ELLIOTT,D.	4285	C
HASTINGS,G.M.	31	D
BRAMLEY,O.H.	4928	C

図 17. REGIONAL(1) データ・セットの更新 (3/3)

第 16 章 ワークステーション VSAM データ・セットの定義と使用

ワークステーションとメインフレーム間のデータの移動	266	VSAM ファイルを使用するプログラムの修正	272
ワークステーション VSAM 編成	266	ワークステーション VSAM 順次データ・セットの使用	273
ワークステーション VSAM データ・セットの作成とアクセス	266	順次ファイルを用いたワークステーション VSAM 順次データ・セットへのアクセス	274
必要なワークステーション VSAM データ・セットの判別	267	ワークステーション VSAM 順次データ・セットの定義とロード	275
ワークステーション VSAM データ・セット内のレコードへのアクセス	267	順次データ・セットの更新	276
ワークステーション VSAM データ・セットのキーの使用	268	ワークステーション VSAM キー順データ・セット	277
ワークステーション VSAM キー順データ・セットのキーの使用	268	ワークステーション VSAM キー順データ・セットのロード	280
順次レコード値の使用	268	SEQUENTIAL ファイルを用いたワークステーション VSAM キー順データ・セットへのアクセス	282
相対レコード番号の使用	269	DIRECT ファイルを用いたワークステーション VSAM キー順データ・セットへのアクセス	283
データ・セット・タイプの選択	269	ワークステーション VSAM 直接データ・セット	286
ワークステーション VSAM データ・セットのファイルの定義	269	ワークステーション VSAM 直接データ・セットのロード	289
PL/I ENVIRONMENT 属性のオプションの指定	270	SEQUENTIAL ファイルを使用したワークステーション VSAM 直接データ・セットへのアクセス	291
既存のプログラムのワークステーション VSAM 向けの修正	270	READ ステートメントの使用	291
CONSECUTIVE ファイルを使用するプログラムの修正	271	WRITE ステートメントの使用	291
INDEXED ファイルを使用するプログラムの修正	271	REWRITE または DELETE ステートメントの使用	292
REGIONAL(1) ファイルを使用するプログラムの修正	271	DIRECT ファイルを使用したワークステーション VSAM 直接データ・セットへのアクセス	292

本章では、レコード単位のデータ伝送のために、ワークステーション上で分散データ管理 (DDM)、ISAM、BTRIEVE データ・セットなどの仮想記憶アクセス方式 (VSAM) データ・セットを使用する方法について説明します。

プラットフォームの違い

PL/I ワークステーション製品に関連して、3 種類のアクセス方法を説明します。ただし、どのプラットフォームでも 3 種類の方法がすべてサポートされているとは限りません。以下をガイドラインとして参照してください。

- DDM - AIX でのみサポートされる
- ISAM - AIX および Windows でサポートされる
- BTRIEVE - Windows でのみサポートされる

本章ではまた、3 種類の VSAM データ・セット (順次、キー順、直接) にアクセスするために使用するステートメントについても説明します。ワークステーション VSAM は、多くの点でメインフレーム上の VSAM と類似しています。ワークステ

ーション上では、順次、キー順、および直接という用語は、VSAM の入力順データ・セット、キー順データ・セット、および相対レコード・データ・セットに類似しています。

そしてこの章の終わりには、ワークステーション VSAM データ・セットを作成し、それにアクセスするために必要な、PL/I ステートメントおよび DD:ddname 環境変数の一連の例を示しています。

ワークステーションとメインフレーム間のデータの移動

メインフレーム VSAM ファイルを、対応する DDM ファイル、ISAM ファイル、または BTRIEVE ファイルに変換するには、LODVSAM ユーティリティのプログラムで文書化された手続きに従います。適切なアクセス方式 AMTHD (DDMIISAMIBTRIEVE) を指定してください。

DDM ファイル、ISAM ファイル、または BTRIEVE ファイルを、対応するメインフレーム VSAM ファイルに変換するには、RELOAD ユーティリティのプログラムで文書化された手続きに従います。これらのユーティリティは、PL/I for Windows ではサポートされていますが、現在 PL/I for AIX ではサポートされていません。

ワークステーション VSAM 編成

PL/I は、ワークステーション VSAM の順次データ・セット、キー順データ・セット、および直接データ・セットをサポートします。上記のデータ・セットはそれぞれ、PL/I の連続データ・セット編成、索引付きデータ・セット編成、および相対データ・セット編成と対応しています。

3 つのタイプすべてのデータ・セットにおいて、順次アクセスとキー順アクセスの両方を行うことができます。キー順データ・セットを使用すると、論理レコードに含まれるキーを使用して、キー順アクセスを行います。キー順アクセスは相対レコード番号を使用する直接データ・セットに対して実行可能です。キー順アクセスは、順次レコード値をキーとして使用する順次データ・セットに対しても可能です。

すべてのワークステーション VSAM データ・セットは、直接アクセス記憶装置に保管されます。ワークステーション 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。

ワークステーション VSAM データ・セットの作成とアクセス

PL/I アプリケーションは、ワークステーション VSAM データ・セットを作成するか、または他のプログラムが作成した VSAM データ・セットにアクセスすることができます。ファイルをワークステーション VSAM データ・セットと関連付けるためにオープンする場合、そのデータ・セットが存在しなければ、PL/I は DECLARE ステートメントまたは DD:ddname 環境変数内で指定された属性とオプションを使用して、データ・セットを作成します。

アプリケーションが既存の VSAM データ・セットにアクセスすると、PL/I はそのタイプ (順次、直接、またはキー順) を判別します。

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。

正しいアクセス方法の使用

アクセス方式 DDM、ISAM、または BTRIEVE を使用して作成されたデータ・セットは、それぞれ同じアクセス方式を使用してアクセスしてください。例えば、BTRIEVE アクセス方式によって作成したデータ・セットにアクセスするために、ISAM アクセス方式を使用することはできません。

必要なワークステーション VSAM データ・セットの判別

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。

- 順次データ・セット は、主としてレコードが作成された順序 (またはその逆の順序) でアクセスされ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)。
- 直接データ・セット は、各項目が特定の番号を持ち、通常その番号によって関連レコードにアクセスされ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)。

ワークステーション VSAM データ・セット内のレコードへのアクセス

どのタイプのワークステーション VSAM データ・セット内のレコードでも、キーを使用して直接アクセスでき、また順次に (逆方向または順方向に) アクセスすることもできます。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。

表 18 に、データをどのようにこの 3 つの異なるタイプのワークステーション VSAM データ・セット内に保管できるかを示し、それぞれの利点と欠点を示しています。

表 18. ワークステーション 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
順次	順次 (順方向のみ)。	SEQUENTIAL 逆方向または順方向。	新しいレコードは終わりにのみ。	利点 簡単迅速に作成。
	各レコードの順次レコード値を入手して、キーとして使用可能。	順次レコード値を使用して 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。	順次または KEYED のアクセスが可能。 レコードの削除が許可される。	用途 データが主として順次式でアクセスされる場合。

表 18. ワークステーション VSAM データ・セットのタイプと利点 (続き)

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、またはキーによってランダム。	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。	キーを指定して KEYED。 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。 用途 アクセスがキーと関連する場合に使用。
直接	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。 用途 レコードが番号によってアクセスされる場合に使用。

ワークステーション VSAM データ・セットのキーの使用

ワークステーション 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・セットの場合、キーは、論理レコード内部の定義済みフィールドです。順次データ・セットの場合、キーは、レコードの順次レコード値です。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。

ワークステーション VSAM キー順データ・セットのキーの使用

キー順データ・セットのキーは、データ・セットに記録された論理レコードの一部です。データ・セットの作成時に、キーの長さや位置を定義します。

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『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。

順次レコード値の使用

順次レコード値を用いると、KEYED SEQUENTIAL ファイルに関連した順次データ・セット上で、キー順アクセスを使用することができます。

BTRIEVE および ISAM

順次レコード値 (すなわちキー) は、長さが 7 文字の文字ストリングで、その値はワークステーション VSAM によって定義されます。

PL/I では順次レコード値を構成または操作することはできません。ただし、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。順次レコード値は通常は印刷できません。

レコードの順次レコード値を求めるには、データ・セットをロードまたは拡張する場合には WRITE ステートメント上で、データ・セットを読み取る場合は 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順次レコード値を後から使用することができます。

相対レコード番号の使用

直接データ・セット内のレコードは、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によって識別します。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 10 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 10 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 10 の文字ストリングです。

データ・セット・タイプの選択

アプリケーションを計画するとき、最初に使用するデータ・セットのタイプを決定する必要があります。使用できるワークステーション VSAM データ・セットには 3 つのタイプがあります。ワークステーション 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、ワークステーション VSAM だけで使用できる機能が備わっています。ワークステーション VSAM は通常、他のデータ・セットのタイプと同等か、場合によっては改善されたパフォーマンスを示します。ただし、ワークステーション 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。

267 ページの表 18 では、ワークステーション VSAM データ・セットの各タイプで使用可能な機能を示しています。各種ワークステーション VSAM データ・セットのうちのいずれかを選択するときには、プログラムが個々のデータをアクセスする最も一般的な順序に基づいて決定する必要があります。

273 ページの表 19、277 ページの表 20、および 286 ページの表 21 はそれぞれ、順次データ・セット、キー順データ・セット、および直接データ・セットを示しています。

ワークステーション VSAM データ・セットのファイルの定義

ワークステーション VSAM 順次データ・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。

```
dc1 Filename file record
      input | output | update
      sequential
```

ワークステーション VSAM データ・セットのファイルの定義

```
buffered
[keyed]
environment(organization(consecutive));
```

ワークステーション VSAM キー順データ・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 | direct
      buffered | unbuffered
[keyed]
environment(organization(indexed));
```

ワークステーション VSAM 直接データ・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 | sequential
      unbuffered | buffered
[keyed]
environment(organization(relative));
```

ファイル属性については、この製品の「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。

PL/I ENVIRONMENT 属性のオプションの指定

データ・セット構造に影響を与える PL/I ENVIRONMENT 属性のオプションの多くは、ワークステーション 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されるだけです。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。

ワークステーション 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
CONSECUTIVE
CTLASA
GENKEY
GRAPHIC
KEYLENGTH
KEYLOC
ORGANIZATION(CONSECUTIVE|INDEXED|RELATIVE)
RECSIZE
SCALARVARYING
VSAM
```

これらの ENVIRONMENT オプションとその使用方法の完全な説明については、202 ページの『PL/I ENVIRONMENT 属性の使用による特性の指定』を参照してください。このリストの ENVIRONMENT オプションに加え、DD ステートメントで使用されるオプションのセットがあります。209 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。

既存のプログラムのワークステーション VSAM 向けの修正

このセクションは、主に、プログラムをワークステーションに転送しようとする OS PL/I ユーザーを対象としています。

ほとんどの場合、ENVIRONMENT(CONSECUTIVE) または ENVIRONMENT (INDEXED) を指定して宣言されたか、または PL/I ENVIRONMENT 属性を指定せずに宣言されたファイルを、PL/I プログラムが使用しているのであれば、このプログラムは修正しなくてもワークステーション VSAM データ・セットにアクセスできます。PL/I は、ワークステーション VSAM データ・セットがオープンされることを検出するため、正しいアクセスを提供できます。

ワークステーション VSAM データ・セットと共に使用できるように、CONSECUTIVE、INDEXED、REGIONAL(1)、または VSAM ファイルを用いる既存のプログラムを修正することが可能です。連続ファイルを用いるプログラムは修正が必要ない場合があります、また常に、ロジックが EXCLUSIVE ファイルに依存していなければ、索引付きファイルを用いるプログラムを修正する必要はありません。REGIONAL(1) データ・セットを用いるプログラムでは、小さな改訂のみ必要です。

以下のセクションでは、ファイルをワークステーション向けに修正するために必要な変更について説明します。

CONSECUTIVE ファイルを使用するプログラムの修正

DDM には固定長レコードの概念はありませんが、ISAM および BTREVEE にはあります。DDM には固定長レコードの概念はありません。プログラムが RECORD 条件に依存して不正な長さのレコードを検出している場合、プログラムが ワークステーション VSAM データ・セットを用いるときの動作は、ワークステーション VSAM 以外のデータ・セットを用いるときとは異なります。

プログラムのロジックが、不正な長さのレコードの検出時に RECORD 条件を発生させることに依存している場合、レコード長を検査して必要な処理を行う独自のコードを作成する必要があります。これは、指定された最大長さまでの任意の長さのレコードをワークステーション VSAM データ・セットでできるようにするためです。

INDEXED ファイルを使用するプログラムの修正

INDEXED ファイルについては、互換性があります。INDEXED ENVIRONMENT オプションを指定して宣言したファイルの場合、PL/I はそのファイルをワークステーション VSAM キー・データ・セットに関連付けます。データ・セットが別のタイプである場合は UNDEFINEDFILE が発生します。

メインフレーム ISAM レコード処理はワークステーション VSAM レコード処理と細部で異なるため、ワークステーション VSAM 処理によって必要な結果が得られるとは限りません。

RECORD 条件への依存関係を除去し、必要であればレコード長を検査する独自のコードを挿入する必要があります。また、削除済みレコードの検査をすべて除去する必要があります。

REGIONAL(1) ファイルを使用するプログラムの修正

REGIONAL(1) データ・セットを使用するプログラムを、ワークステーション VSAM 直接データ・セットを使用するように修正することができます。ファイル宣言から REGIONAL(1) およびその他のインプリメンテーション依存のオプションを除去し、ENV(ORGANIZATION(RELATIVE)) で置き換えます。また、ワークステー

ション VSAM の削除済みレコードにはアクセスできないため、削除済みレコードの検査をすべて除去する必要があります。

VSAM ファイルを使用するプログラムの修正

VSAM ENVIRONMENT オプションを使用する場合、ファイルをオープンする前に、関連するワークステーション VSAM データ・セットが存在していなければなりません。単純なプログラムでデータ・セットを作成することができます。図 18 は、ワークステーション VSAM キー順データ・セット作成の例です。

```
/******  
/*  
/* NAME - ISAM0.PLI  
/*  
/* DESCRIPTION  
/* Create an ISAM Keyed data set  
/*  
/*  
/******  
  
NewVSAM: proc options(main);  
    declare  
        NewFile keyed record output file  
            env(organization(indexed)  
                recsize(80)  
                keylength(8)  
                keyloc(17)  
            );  
        open file(NewFile) title('/KEYNAMES.DAT');  
        close file(NewFile);  
End NewVSAM;
```

図 18. ワークステーション VSAM キー順データ・セットの作成

KEYNAMES.DAT という名前のデータ・セットがまだ存在していない場合、OPEN ステートメントが実行されたときに PL/I はその名前でデータ・セットを作成します。

ワークステーション VSAM 順次データ・セットの使用

ワークステーション VSAM データ・セット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 19 に示します。

表 19. ワークステーション VSAM 順次データ・セットのロードとおよび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);	KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(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 INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) ³ または KEYTO(reference)
	READ FILE(file-reference); ²	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
	DELETE FILE(file-reference);	KEY(expression)

表 19. ワークステーション VSAM 順次データ・セットのロードとおよび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 UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) ³ または KEYTO(reference)
	READ FILE(file-reference); ²	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	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順次レコード値でなければなりません。

順次ファイルを用いたワークステーション VSAM 順次データ・セットへのアクセス

順次データ・セット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管されます。

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順次レコード値を取得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。

ワークステーション VSAM 順次データ・セットにアクセスするために使用する 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順次レコード値を回復するには、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順次レコード値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 ステートメントによって再書き込みの

行われるレコードは、KEY オプションを使用する場合は、指定された順次レコード値を持つものであり、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。

ワークステーション VSAM 順次データ・セットの定義とロード

276 ページの図 19 は、ワークステーション VSAM 順次データ・セットを定義およびロードするプログラムの例です。

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。

レコードの順次レコード値は、KEYED ファイル内のキーとして後で使用するために、書き込み時に取得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。

```
dc1 Chars char(7); /*DDM uses 4; BTRIEVE and ISAM use 7 as shown */
write file(Famfile) from (String)
  keyto(Chars);

dc1 Chars char(4); /* DDM uses 4 */
write file(Famfile) from (String)
  keyto(Chars);
```

通常、キーは印刷できませんが、後で使用するのためのために保存しておくことができます。

```

/*****
/*
/*
/* DESCRIPTION
/*   Define and load an ISAM sequential data set.
/*
/*
/* USAGE
/*   The following commands are required to establish
/*   the environment variables to run this program:
/*
/*       SET DD:IN=ISAM1.INP,RECSIZE(38)
/*       SET DD:FAMFILE=ISAM1.OUT,AMTHD(ISAM),RECSIZE(38)
/*
*****/

```

```

CREATE: proc options(main);

    dcl
        FamFile file sequential output
                env(organization(consecutive)),
        In file record input,
        Eof bit(1) init('0'b),
        i fixed(15),
        String char(38);

    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 CREATE;

```

The input data for this program might look like this:

Fred	69	M
Andy	70	M
Susan	72	F

図 19. ワークステーション VSAM 順次データ・セットの定義とロード

順次データ・セットの更新

図 19 に示すプログラムを使用して、ワークステーション VSAM 順次データ・セットを更新することができます。プログラムが再度実行されると、データ・セットの末尾に新規のレコードが追加されます。

レコードの長さを変えないかぎり、順次データ・セット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、キーは直前の WRITE または READ ステートメントから取得した順次レコード値でなければなりません。

ワークステーション VSAM キー順データ・セット

ワークステーション VSAM キー順データ・セットで利用できるステートメントとオプションを、表 20 で示しています。

表 20. ワークステーション VSAM キー順データ・セットのロードおよび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 based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference) SET(pointer-reference); READ FILE(file-reference); ²	KEY(expression) または KEYTO(reference) KEY(expression) または KEYTO(reference) IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference); READ FILE(file-reference); ²	KEY(expression) または KEYTO(reference) IGNORE(expression)

表 20. ワークステーション VSAM キー順データ・セットのロードおよび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	KEYTO(reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	IGNORE(expression)
	REWRITE FILE(file-reference);	FROM(reference)
	DELETE FILE(file-reference)	および/または KEY(expression)
		KEY(expression)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-reference); ²	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	REWRITE FILE(file-reference) FROM(reference);	KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
DIRECT ³ INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT ³ INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	

表 20. ワークステーション VSAM キー順データ・セットのロードおよび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 OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT ³ 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);	
DIRECT ³ UPDATE UNBUFFERED	READ FILE(file-reference) INTO(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 20. ワークステーション VSAM キー順データ・セットのロードおよび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 を組み込むことができます。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 DIRECT ファイルを、重複キー機能を持つ ワークステーション VSAM データ・セットに関連付けしないでください。

ワークステーション VSAM キー順データ・セットのロード

キー順データ・セット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。

キー順データ・セットにすでにレコードがあって、SEQUENTIAL 属性と OUTPUT 属性をもつ関連ファイルをオープンする場合は、データ・セットの末尾のみにレコードを追加することができます。この場合も、レコードは昇順のキー順で提示しなければならず、KEYFROM オプションを使用する必要があります。さらに、提示する最初のレコードのキーは、データ・セット上に存在する最高位のキーより大きくなければなりません。

281 ページの図 20 は、ワークステーション VSAM キー順データ・セットをロードするプログラムの例です。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。キー順データ・セットはこの方法でロードしなければなりません。

```

/*****
/*
/* DESCRIPTION
/*   Load an ISAM keyed data set.
/*
/*
/* USAGE
/*   The following commands are required to establish
/*   the environment variables to run this program:
/*
/*       SET DD:DIREC=ISAM2.OUT,AMTHD(ISAM)
/*       SET DD:SYSIN=ISAM2.INP,RECSIZE(80)
/*
*****/

NAMELD: proc options(main);

    dcl Direc file record keyed sequential output
        env(organization(indexed)
            recsize(23)
            keyloc(1)
            keylength(20)
        );

    dcl Eof bit(1) init('0'b);

    dcl 1 IoArea,
        5 Name char(20),
        5 Number char(3);

    on endfile(sysin) Eof = '1'b;

    open file(Direc);

    get file(sysin) edit(Name,Number) (a(20),a(3));
    do while (~Eof);
        write file(Direc) from(IoArea) keyfrom(Name);
        get file(sysin) edit(Name,Number) (a(20),a(3));
    end;

    close file(Direc);
end NAMELD;

```

図 20. ワークステーション VSAM キー順データ・セットの定義とロード (1/2)

The input file for this program could be:

ACTION,G.	162
BAKER,R.	152
BRAMLEY,O.H.	248
CHEESMAN,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
SURCLIFFE,M.	472
TAYLOR,G.C.	407
WILTON,L.W.	404
WINSTONE,E.M.	307

図 20. ワークステーション VSAM キー順データ・セットの定義とロード (2/2)

SEQUENTIAL ファイルを用いたワークステーション VSAM キー順データ・セットへのアクセス

キー順データ・セットへのアクセスに使用する 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。KEYTO オプションを使用することによって、このように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、指定したキーをもつレコードになります。この 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配置し、その後の順次読み取りによって後続のレコードがキー順に回復されます。

KEYFROM オプションのある WRITE ステートメントを、KEYED SEQUENTIAL UPDATE ファイルで使うことができます。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。そのデータ・セットにすでに存在す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と、KEY 条件が発生します。

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。

DIRECT ファイルを用いたワークステーション VSAM キー順データ・セットへのアクセス

ワークステーション VSAM キー順データ・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 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・セットにすでにあるレコードと同じキーをもつレコードを挿入しようとする、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。

284 ページの図 21 に、キー順データ・セットを更新するために使用可能なメソッドの 1 つを示します。

```

/*****
/*
/*
/* DESCRIPTION
/*   Update an ISAM keyed data set by key.
/*
/*
/*
/* USAGE
/*   The following commands are required to establish
/*   the environment variables to run this program:
/*
/*       SET DD:DIREC=ISAM2.OUT,AMTHD(ISAM)
/*       SET DD:SYSIN=ISAM3.INP,RECSIZE(80)
/*
/*   Note: This program is using ISAM2.OUT file created by the
/*         previous sample program NAMELD.
/*
*****/

DIRUPDT: proc options(main);

  dcl Direc file record keyed update
    env(organization(indexed)
        recsize(23)
        keyloc(1)
        keylength(20)
    );

  dcl 1 IoArea,
    5 NewArea,
    10 Name char(20),
    10 Number char(3),
    5 Code char(1);

  dcl oncode builtin;
  dcl 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;

```

図 21. ワークステーション VSAM キー順データ・セットの更新 (1/2)

```

get file(sysin) edit (Name,Number,Code) (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(NewArea) keyfrom(Name);
    when('C') rewrite file(Direc) from(NewArea) 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) (a(20),a(3),a(1));
end;

close file(Direc);
put file(sysprint) page;

/* Display the updated file */

open file(Direc) sequential input;

Eof = '0'b;
on endfile(Direc) Eof = '1'b;

read file(Direc) into(NewArea);
do while(~Eof);
  put file(sysprint) skip edit(Name,Number)(a,a);
  read file(Direc) into(NewArea);
end;
close file(Direc);
end DIRUPDT;

```

An input file for this program might look like this one:

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.	X
ROLF,D.E.	D
ELLIOTT,D.	291C
HASTINGS,G.M.	D
BRAMLEY,O.H.	439C

図 21. ワークステーション VSAM キー順データ・セットの更新 (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

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置がとられます。更新が終了すると、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。

ワークステーション VSAM 直接データ・セット

ワークステーション VSAM 直接データ・セットで使用するステートメントとオプションを、次に示します。

表 21. ワークステーション VSAM 直接データ・セットのロードおよび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 based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) または KEYTO(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
	READ FILE(file-reference); ²	KEYTO(reference)
		IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(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
	READ FILE(file-reference); ²	KEYTO(reference)
	WRITE FILE(file-reference) FROM(reference);	IGNORE(expression)
	REWRITE FILE(file-reference);	KEYFROM(expression) または KEYTO(reference)
	DELETE FILE(file-reference);	FROM(reference) および/または KEY(expression)
		KEY(expression)

表 21. ワークステーション VSAM 直接データ・セットのロードおよび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 UNBUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) または KEYTO(reference)
	READ FILE(file-expression); ²	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) または KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	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);	
DIRECT INPUT UNBUFFERED	READ FILE(file-reference) KEY(expression);	

表 21. ワークステーション VSAM 直接データ・セットのロードおよび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);	
DIRECT UPDATE UNBUFFERED	READ FILE(file-reference) INTO(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。

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。

ワークステーション VSAM 直接データ・セットのロード

直接データ・セット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 (268 ページの『ワークステーション VSAM データ・セットのキーの使用』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 オプションを使用することにより、書き込まれたレコードの相対レコード番号を取得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、直接データ・セットを順次にロードする場合は、KEYED 属性を使用する必要はありません。

すで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のは誤りです。KEYFROM オプションを使用すると KEY 条件が生じ、それを省略すると ERROR 条件が発生します。

290 ページの図 22 は、ワークステーション VSAM 直接データ・セットを定義およびロードするプログラムの例です。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ます。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。そして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ます。

```

/*****
/*  DESCRIPTION
/*    Load an ISAM direct data set.
/*
/*  USAGE
/*    The following commands are required to establish
/*    the environment variables to run this program:
/*
/*      SET DD:SYSIN=ISAM4.INP,RECSIZE(80)
/*      SET DD:NOS=ISAM4.OUT,AMTHD(ISAM),RECCOUNT(100)
/*****
CREATD: proc options(main);

    dcl Nos file record output direct keyed
        env(organization(relative) reccsize(20) );

    dcl Sysin file input record;
    dcl 1 In_Area,
        2 Name char(20),
        2 Number char( 2);
    dcl Sysin_Eof bit (1) init('0'b);
    dcl Ntemp fixed(15);

    on endfile (Sysin) Sysin_Eof = '1'b;

    open file(Nos);
    read file(Sysin) into(In_Area);
    do while(~Sysin_Eof);
        Ntemp = Number;
        write file(Nos) from(Name) keyfrom(Ntemp);
        put file(sysprint) skip edit (In_Area) (a);
        read file(Sysin) into(In_Area);
    end;

    close file(Nos);
end CREATD;

```

図 22. ワークステーション VSAM 直接データ・セットのロード (1/2)

This could be the input file for this program:

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 22. ワークステーション VSAM 直接データ・セットのロード (2/2)

SEQUENTIAL ファイルを使用したワークステーション VSAM 直接データ・セットへのアクセス

直接データ・セットにアクセスするために使用する 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、プログラムで属性 KEYED も使用する必要があります。

READ ステートメントの使用

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配置します。その後の順次読み取りによって後続のレコードが順番に回復されます。

WRITE ステートメントの使用

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・セットにすでに存在するレコードと同じ相対レコード番号をも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 または DELETE ステートメントの使用

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 オプションの有無に関係なく、データ・セットからレコードを削除するために使用することもできます。

DIRECT ファイルを使用したワークステーション VSAM 直接データ・セットへのアクセス

直接データ・セットにアクセスするために使用する DIRECT ファイルは、OUTPUT 属性、INPUT 属性、または 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まったく同様に、レコードの読み取り、書き込み、再書き込み、または削除が可能です。

293 ページの図 23 は、直接データ・セット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。ワークステーション 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。

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) にあるような空のレコードの検査は必要ありません。

```

/*****
/*
/*
/* DESCRIPTION
/*   Update an ISAM direct data set by key.
/*
/*
/*
/* USAGE
/*   The following commands are required to establish
/*   the environment variables to run this program.
/*
/*   SET DD:SYSIN=ISAM5.INP,RECSIZE(80)
/*   SET DD:NOS=ISAM4.OUT,AMTHD(ISAM),APPEND(Y)
/*
/* Note: This sample program is using the direct ISAM data set
/*       ISAM4.OUT created by the previous sample program CREATD.
/*
/*
*****/

UPDATD: proc options(main);

    dc1 Nos    file record keyed
              env(organization(relative));
    dc1 Sysin  file input record;

    dc1 Sysin_Eof bit (1) init('0'b);
    dc1  Nos_Eof bit (1) init('0'b);

    dc1 1  In_Area,
        2  Name    char(20),
        2  (CNewNo,COldNo) char( 2),
        2  In_Area_1 char( 1),
        2  Code    char( 1);

    dc1 IoField char(20);
    dc1 NewNo fixed(15);
    dc1 OldNo fixed(15);

    dc1 oncode builtin;

    on endfile (Sysin) sysin_Eof = '1'b;
    open file (Nos) direct update;

```

図 23. キーによるワークステーション VSAM 直接データ・セットの更新 (1/3)

```

/* trap errors */

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;

/* update the direct data set */

read file(Sysin) into(In_Area);

do while(~Sysin_Eof);
  if CNewNo~=' ' then
    NewNo = CNewNo;
  else
    NewNo = 0;
  if COldNo~=' ' then
    OldNo = COldNo;
  else
    OldNo = 0;
  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 list ('Invalid code:',Name);
end;
read file(Sysin) into(In_Area);
end;

close file(Sysin),file(Nos);

/* open and print updated file */

open file(Nos) sequential input;
on endfile (Nos) Nos_Eof = '1'b;

```

図 23. キーによるワークステーション VSAM 直接データ・セットの更新 (2/3)

```
read file(Nos) into(IoField) keyto(CNewNo);
do while(~Nos_Eof);
  put file (sysprint) skip
  edit (CNewNo,IoField)(a(5),a);
  read file(Nos) into(IoField) keyto(CNewNo);
end;
close file(Nos);
end UPDATD;
```

An input file for this program might look like this:

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	X
ROLF,D.F.	14	D
ELLIOTT,D.	4285	C
HASTINGS,G.M.	31	D
BRAMLEY,O.H.	4928	C

図 23. キーによるワークステーション VSAM 直接データ・セットの更新 (3/3)

第 5 部 データベースとの PL/I の使用

第 17 章 Open Database Connectivity

ODBC の紹介	297	接続ストリングの使用	300
背景	297	エラー・メッセージ	300
ODBC ドライバー・マネージャー	298	PL/I からの ODBC API	301
組み込み SQL または ODBC の選択	298	CALL インターフェースの規則	302
ODBC ドライバーの使用	298	提供されるインクルード・ファイルの使用	302
オンライン・ヘルプ	299	ODBC C タイプのマッピング	304
環境固有の情報	299	ODBC ドライバー・マネージャー/ドライバの	
ドライバー名	299	ライセンス情報の設定	304
データ・ソースの構成	299	提供されるインクルード・ファイルを使用したサン	
データ・ソースへの接続	299	プル・プログラム	305
ログオン・ダイアログ・ボックスの使用	300		

この章では、PL/I アプリケーション内で Open Database Connectivity (ODBC) インターフェースを使用するための情報を提供します。ODBC を使用すると、ODBC インターフェースをサポートするさまざまなデータベースやファイル・システムのデータにアクセスできるだけでなく、アクセスを動的に行うことができます。

データベース・アクセス用の組み込み SQL を使用する PL/I アプリケーションは、特定のデータベース用のプリプロセッサによって処理する必要があり、ターゲット・データベースが変更された場合は再コンパイルする必要があります。ODBC は呼び出しインターフェースであるため、組み込み SQL を使用した場合のように、ターゲット・データベースをコンパイル時に指定することはありません。複数のデータベース用に複数バージョンのアプリケーションを用意する必要がないだけでなく、アプリケーションがターゲット・データベースを動的に決定することができます。

ODBC の紹介

ODBC は、アプリケーション・プログラム・インターフェース (API) の仕様であり、アプリケーションが構造化照会言語 (SQL) を用いて複数のデータベース管理システムにアクセスできるようにするものです。

ODBC は、最大限のインターオペラビリティを可能にします。単一のアプリケーションが、多くの異なるデータベース管理システムにアクセスできます。これにより、特定のタイプのデータ・ソースをターゲットにしなくても、アプリケーションを開発、コンパイル、および出荷することができます。ユーザーはその後でデータベース・ドライバーを追加することができます。データベース・ドライバーは、アプリケーションをユーザーが選択したデータベース管理システムにリンクします。

背景

X/Open Company および SQL Access Group は、*X/Open Call Level Interface* と呼ばれる、呼び出し可能 SQL インターフェースの仕様を共同で開発しました。このインターフェースの目的は、アプリケーションが特定のデータベース・ベンダーのプログラミング・インターフェースに依存しないようにすることにより、アプリケーションの移植性を高めることです。

ODBC は元来、Microsoft によって、X/Open CLI の素案に基づき、Microsoft オペレーティング・システム向けに開発されました。それ以降、他のベンダーが、OS/2 システムや UNIX システムなどの他のプラットフォームで動作する ODBC ドライバーを提供してきました。

この章の説明および例は、ODBC バージョン 3.0 に適用されます。ODBC インクルード・ファイルについて詳しくは、302 ページの『提供されるインクルード・ファイルの使用』を参照してください。

ODBC ドライバー・マネージャー

ODBC インターフェースを使用する場合、アプリケーションはドライバー・マネージャーを通じて呼び出しを行います。ドライバー・マネージャーは、アプリケーションが接続するデータベース・サーバー用に必要なドライバーを、動的にロードします。するとドライバーは、呼び出しを受け入れ、SQL を指定されたデータ・ソース (データベース) に送信し、結果を戻します。

組み込み SQL または ODBC の選択

組み込み SQL および ODBC には、特有の利点があります。組み込み SQL の利点の一部を以下に挙げます。

- 静的 SQL は通常、動的 SQL よりも高いパフォーマンスを示します。実行時に準備する必要が無いため、処理とネットワーク・トラフィックの両方を削減します。
- 静的 SQL を使用すれば、データベース管理者は、パッケージへのアクセスをユーザーに認可するだけでよく、使用される各テーブルやビューへのアクセスを認可する必要はありません。

ODBC の利点の一部を以下に挙げます。

- 使用されるデータベース・サーバーの種類にかかわらず、一貫したインターフェースを提供します。
- 複数の同時接続が可能です。
- アプリケーションを、処理対象の各データベースにバインドする必要はありません。PL/I for Windows が自動的にユーザーに代わってこのバインドを行います。自動的に 1 つのデータベースにのみバインドします。実行時に動的に接続するデータベースを選択したい場合は、異なるデータベースにバインドするための追加のステップを実行する必要があります。

ODBC ドライバーの使用

ODBC が PL/I でデータをアクセスできるようにするためには、インストール時に「ODBC Drivers」コンポーネントを選択して ODBC ドライバー・マネージャーおよびドライバーをインストールする必要があります。

重要: インストール・プロセスの間に、ODBC ドライバーのライセンス・ファイルがご使用のシステム上にインストールされます。

ivib.lic という名前のファイルが `x:¥plidir¥ODBC` にインストールされます。ここで、`x` および `plidir` はそれぞれ PL/I for Windows がインストールされるドライブとディレクトリーです。

このファイルは、アプリケーション実行時に ODBC ドライバー使用のライセンス交付を受けていることを検査する際に使用されるため、インストール・ディレクトリー内に保持しておく必要があります。304 ページの『ODBC ドライバー・マネージャー/ドライバーのライセンス情報の設定』で、関数呼び出しを使用してこの検査を起動する方法を説明しています。

オンライン・ヘルプ

ODBC ドライバー用のオンライン・ヘルプが提供されており、解説書としても状況依存ヘルプとしても利用できます。具体的なファイル名などは異なる場合があります。このセクションで示す名前は PL/I 用のファイル名であるため注意してください。

環境固有の情報

ODBC ドライバーは 32 ビットのドライバーです。データベース・システム・ベンダーが提供する必要なネットワーク・ソフトウェアは、32 ビット対応でなければなりません。

ドライバー名

Windows 用ドライバーは ODBC 3.0 以降のレベルでなければなりません。

ODBC.INI は、Windows レジストリー内の

`HKEY_CURRENT_USER¥¥SOFTWARE¥¥ODBC` キーのサブキーです。ODBC.INI サブキーは、ODBC アドミニストレーターが保守します。ODBC アドミニストレーターはメインの PL/I プログラム・グループ内にあります。Windows は複数ユーザーをサポートできるため、ODBC.INI サブキーはレジストリー内の固有ユーザー・キーの下に保管されます。

データ・ソースの構成

データ・ソースは、DBMS と、DBMS にアクセスするために必要なリモート・オペレーティング・システムおよびネットワークで構成されています。ドライバーのインストール後、ODBC アドミニストレーター・プログラムを使用してデータ・ソースを構成する必要があります。ODBC アドミニストレーターはメインの PL/I プログラム・グループ内にあります。Windows は複数のユーザーをホスティングできるため、各ユーザーが自分のデータ・ソースを構成する必要があります。構成したい特定のドライバーの詳細な構成情報については、オンライン・ヘルプの該当するセクションを参照してください。

データ・ソースへの接続

ODBC アプリケーションは、データ・ソースに応じて、ログオン・ダイアログ・ボックスまたは接続ストリングのいずれかを使用して、データ・ソースに接続する必要があります。

ログオン・ダイアログ・ボックスの使用

一部の ODBC アプリケーションは、データ・ソースへ接続するときにログオン・ダイアログ・ボックスを表示します。このような場合、データ・ソース名はすでに指定されています。

ログオン・ダイアログ・ボックスでは、次の操作を実行します。

1. リモート・データベースの名前を入力するか、または「**データベース名**」ドロップダウン・リストからリモート・データベースの名前を選択します。

クライアントからアクセスしたいデータベースが、すべてカタログされていなければなりません。

2. 必要に応じて、ユーザー名 (許可 ID) を入力します。
3. 必要に応じて、パスワードを入力します。

ユーザー名とパスワードをブランクのままにした場合、ODBC アプリケーションは、ユーザーが SQLLOGN2 (DOS 環境下) を使用するか、またはユーザー・プロファイル管理を使用して、すでにログオンしているものと見なします。ログオンしていない場合、アプリケーションはエラーを戻します。ダイアログ・ボックスでユーザー名とパスワードを入力するか、あるいは SQLLOGN2 と STARTDRQ (DOS 環境下) またはユーザー・プロファイル管理を使用してログオンする必要があります。

4. 「OK」をクリックしてログオンを完了し、ODBC.INI 内の値を更新します。

接続ストリングの使用

アプリケーションがデータ・ソースへの接続に接続ストリングを必要とする場合は、データ・ソース名を指定して、どの ODBC.INI セクションをデフォルト接続情報として使用するかをドライバーに知らせる必要があります。任意で、接続ストリング内に attribute=value ペアを指定し、ODBC.INI に格納されているデフォルト値をオーバーライドすることができます。これらの値は ODBC.INI には書き込まれません。

接続ストリングには、長い名前と短い名前のどちらを指定してもかまいません。接続ストリングの書式は次のようになります。

```
DSN=data_source_name[;attribute=value[;attribute=value]...]
```

INFORMIX 5 用の接続ストリングの例を次に示します。

```
DSN=INFORMIX TABLES;DB=PAYROLL
```

エラー・メッセージ

エラー・メッセージは以下のソースから出されます。

- ODBC ドライバー
- データベース・システム
- ドライバー・マネージャー

ODBC ドライバーについて報告されたエラーは、次のようなフォーマットになっています。

```
[vendor] [ODBC_component] message
```

ODBC_component は、エラーが発生したコンポーネントです。例えば、INTERSOLV の SQL Server ドライバーからのエラー・メッセージは、次のようになります。

```
[INTERSOLV] [ODBC SQL Server driver] Login incorrect.
```

このタイプのエラーが出された場合は、アプリケーションが実行した最後の ODBC 呼び出しに問題があるかどうかを検査するか、または ODBC アプリケーション・ベンダーに問い合わせてください。

データ・ソースで発生したエラーには、データ・ソース名が含まれており、次のようなフォーマットになっています。

```
[vendor] [ODBC_component] [data_source] message
```

このタイプのメッセージでは、ODBC_component は指示されたデータ・ソースからエラーを受け取ったコンポーネントです。例えば、Oracle データ・ソースから以下のメッセージが出される場合があります。

```
[INTERSOLV] [ODBC Oracle driver] [Oracle] ORA-0919: specified
length too long for CHAR column
```

このタイプのエラーが出された場合は、データベース・システムで何かを誤って実行したということです。詳細についてご使用のデータベース・システムの資料を調べるか、またはデータベース管理者にご相談ください。この例では、Oracle の資料を調べます。

ドライバー・マネージャーは、ドライバーとの接続を確立し、ドライバーへ要求を実行依頼し、結果をアプリケーションに戻すアプリケーションです。ドライバー・マネージャーで発生したエラーは、次のようなフォーマットになっています。

```
[vendor] [ODBC DLL] message
```

vendor は、Microsoft または INTERSOLV です。例えば、Microsoft ドライバー・マネージャーからのエラーは、次のようになります。

```
[Microsoft] [ODBC DLL] Driver does not support
this function
```

PL/I からの ODBC API

VA PL/I には、ODBC インクルード・ファイルが含まれています。ODBC インクルード・ファイルによって、PL/I プログラムからの ODBC 呼び出しを使用した、ODBC ドライバーによるデータベースへのアクセスが容易になります。このセクションでは、提供される ODBC インクルード・ファイル、ODBC API 引数タイプによる PL/I データ記述へのマッピング方法、および ODBC API に適用可能な追加の PL/I 関数と考慮事項について説明します。

ODBC API について詳しくは、オンライン・ヘルプを参照してください。

ODBC ドライバーに関連する固有情報 (そのドライバーがサポートする ODBC レベルまたは拡張機能) については、そのドライバーとともに提供される仕様を参照してください。

以下のセクションで、PL/I プログラムから ODBC にアクセスする方法を示します。

302 ページの『CALL インターフェースの規則』

『提供されるインクルード・ファイルの使用』
304 ページの『ODBC C タイプのマッピング』
304 ページの『ODBC ドライバー・マネージャー/ドライバのライセンス情報の設定』

LIB ファイル:

ODBC アプリケーションをリンクするときに、インポート・ライブラリー ODBC32.LIB を組み込む必要があります。このライブラリーは ODBC SDK (Microsoft 提供) に含まれています。

CALL インターフェースの規則

ODBC 呼び出しを行うプログラムは DEFAULT(BYVALUE) および LIMITS(EXTNAME(31)) コンパイル時オプションを指定してコンパイルする必要があります。

提供されるインクルード・ファイルの使用

ここで説明およびリストされているインクルード・ファイルは、ODBC パージョン 3.0 向けのものです。

表 22. ODBC 用に提供されるインクルード・ファイル

ファイル名	説明
ODBCSQL.CPY	ODBC 関数用のメインのインクルード・ファイル
ODBCEXT.CPY	Microsoft の ODBC 拡張機能用のインクルード・ファイル
ODBCTYPE.CPY	ODBC 型定義用のインクルード・ファイル
ODBCUCOD.CPY	ユニコード・インクルード・ファイル
ODBCSAMP.PLI	サンプル・プログラム

提供されるインクルード・ファイルは、ODBC API 用に記載された定数値の記号を定義します。引数 (入出力) と関数の戻り値を指定してテストできるように、ODBC API への呼び出しで使用される定数を ODBC ガイドで指定された記号にマップしています。ODBC API 呼び出しを使用するためには、これらのファイルを PL/I プログラムに組み込む必要があります。

PL/I では、31 文字を超える長さの名前は、31 文字になるように切り捨てられるか、または省略されます。303 ページの表 23 に、31 文字を超える長さの名前と、対応する PL/I 名を示します。

表 23. PL/I 用に切り捨てられたかまたは省略された ODBC 名

31 文字を超える ODBC C #define 記号 long	対応する PL/I 名
SQL_AD_ADD_CONSTRAINT_DEFERRABLE	SQL_AD_ADD_CONSTR_DEFERRABLE
SQL_AD_ADD_CONSTRAINT_INITIALLY_DEFERRED	SQL_AD_ADD_CONSTR_INITLY_DEFERD
SQL_AD_ADD_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_AD_ADD_CONSTR_INITLY_IMMEDT
SQL_AD_ADD_CONSTRAINT_NON_DEFERRABLE	SQL_AD_ADD_CONSTR_NON_DEFERRABL
SQL_AD_CONSTRAINT_NAME_DEFINITION	SQL_AD_CONSTR_NAME_DEFINITION
SQL_API_ODBC3_ALL_FUNCTIONS_SIZE	SQL_API_ODBC3_ALL_FUNCTIONS_SZ
SQL_AT_CONSTRAINT_INITIALLY_DEFERRED	SQL_AT_CONSTR_INITIALLY_DEFERD
SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_AT_CONSTR_INITIALLY_IMMED
SQL_AT_CONSTRAINT_NAME_DEFINITION	SQL_AT_CONSTR_NAME_DEFINITION
SQL_AT_CONSTRAINT_NON_DEFERRABLE	SQL_AT_CONSTR_NON_DEFERRABLE
SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE	SQL_AT_DROP_TBL_CONSTR_CASCADE
SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT	SQL_AT_DROP_TBL_CONSTR_RESTRICT
SQL_CA_CONSTRAINT_INITIALLY_DEFERRED	SQL_CA_CONSTR_INITLY_DEFERRED
SQL_CA_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_CA_CONSTR_INITLY_IMMEDIATE
SQL_CA_CONSTRAINT_NON_DEFERRABLE	SQL_CA_CONSTR_NON_DEFERRABLE
SQL_CDO_CONSTRAINT_NAME_DEFINITION	SQL_CDO_CONSTR_NAME_DEFINITION
SQL_CDO_CONSTRAINT_INITIALLY_DEFERRED	SQL_CDO_CONSTR_INITLY_DEFERRED
SQL_CDO_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_CDO_CONSTR_INITLY_IMMEDIAT
SQL_CDO_CONSTRAINT_NON_DEFERRABLE	SQL_CDO_CONSTR_NON_DEFERRABLE
SQL_CT_CONSTRAINT_INITIALLY_DEFERRED	SQL_CT_CONSTR_INITLY_DEFERRED
SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE	SQL_CT_CONSTR_INITLY_IMMEDIATE
SQL_CT_CONSTRAINT_NON_DEFERRABLE	SQL_CT_CONSTR_NON_DEFERRABLE
SQL_CT_CONSTRAINT_NAME_DEFINITION	SQL_CT_CONSTR_NAME_DEFINITION
SQL_DESC_DATETIME_INTERVAL_PRECISION	SQL_DESC_DATETIME_INTERVAL_PREC
SQL_DL_SQL92_INTERVAL_DAY_TO_HOUR	SQL_DL_SQL92_INTERVAL_DAY_TO_HR
SQL_DL_SQL92_INTERVAL_DAY_TO_MINUTE	SQL_DL_SQL92_INTERVAL_DY_TO_MIN
SQL_DL_SQL92_INTERVAL_DAY_TO_SECOND	SQL_DL_SQL92_INTERVAL_DY_TO_SEC
SQL_DL_SQL92_INTERVAL_HOUR_TO_MINUTE	SQL_DL_SQL92_INTERVAL_HR_TO_MIN
SQL_DL_SQL92_INTERVAL_HOUR_TO_SECOND	SQL_DL_SQL92_INTERVAL_HR_TO_SEC
SQL_DL_SQL92_INTERVAL_MINUTE_TO_SECOND	SQL_DL_SQL92_INTERVAL_MN_TO_SEC
SQL_DL_SQL92_INTERVAL_YEAR_TO_MONTH	SQL_DL_SQL92_INTERVAL_YR_TO_MTH
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	SQL_FORWARD_ONLY_CURSOR_ATTRIB1
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	SQL_FORWARD_ONLY_CURSOR_ATTRIB2
SQL_MAX_ASYNC_CONCURRENT_STATEMENTS	SQL_MAX_ASYNC_CONCURRENT_STMTS
SQL_MAXIMUM_CONCURRENT_ACTIVITIES	SQL_MAXIMUM_CONCURRENT_ACTIVITI
SQL_SQL92_FOREIGN_KEY_DELETE_RULE	SQL_SQL92_FOREIGN_KEY_DEL_RULE
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE	SQL_SQL92_FOREIGN_KEY_UPD_RULE
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS	SQL_SQL92_NUMERIC_VALUE_FUNCT
SQL_SQL92_RELATIONAL_JOIN_OPERATORS	SQL_SQL92_RELATIONAL_JOIN_OPER
SQL_TRANSACTION_ISOLATION_OPTION	SQL_TRANSACTION_ISOLATION_OPTN
SQL_TRANSACTION_READ_UNCOMMITTED	SQL_TRANSACTION_READ_UNCOMMITTD

ODBC C タイプのマッピング

ODBC API で指定されるデータ型は、API 定義内の ODBC C タイプに対応して定義されます。次の表では、引数の ODBC C タイプに対応する PL/I 宣言を示します。

表 24. ODBC C タイプの PL/I データ宣言へのマッピング

ODBC C タイプ	PL/I の書式	説明
SQLSMALLINT	FIXED BIN(15)	16 ビット符号付き整数 (2 バイト・バイナリー)
SQLUSMALLINT	FIXED BIN(16) UNSIGNED	16 ビット符号なし整数 (2 バイト・バイナリー)
SQLINTEGER	FIXED BIN(31)	32 ビット符号付き整数 (4 バイト・バイナリー)
SQLUIINTEGER	FIXED BIN(31) UNSIGNED	32 ビット符号なし整数 (4 バイト・バイナリー)
SQLREAL	FLOAT	浮動小数点 (4 バイト)
SQLFLOAT	DOUBLE	浮動小数点 (8 バイト)
SQLDOUBLE	DOUBLE	浮動小数点 (8 バイト)
SQLCHAR *	CHAR(*) VARZ BYADDR	符号なし 8 ビットへのポインター
SQLHDBC	POINTER	接続ハンドル
SQLHENV	POINTER	環境ハンドル
SQLHSTMT	POINTER	ステートメント・ハンドル
SQLHWND	POINTER	ウィンドウ・ハンドル

ODBC ドライバー・マネージャー/ドライバーのライセンス情報の設定

ODBC ドライバー・マネージャー/ドライバーを使用する場合、SQLConnect 関数、SQLDriverConnect 関数、または SQLBrowseConnect 関数を呼び出した直後に ibmODBCLicInfo を呼び出す必要があります。次のように、引数「hdbc」を ibmODBCLicInfo に渡す必要があります。

```
sql_rc = ibmODBCLicInfo(myHDBC);
```

ibmODBCLicInfo ルーチンは、プログラムのリンク段階で組み込まなければならない ibmodlic.lib ライブラリーに含まれています。詳しくは、サンプル・プログラム odbcsamp.pli を参照してください。

提供されるインクルード・ファイルを使用したサンプル・プログラム

提供されるサンプル PL/I プログラムで、以下のようないくつかの一般的な ODBC 関数の使用法を示しています。

SQLAllocEnv	SQLExecute
SQLAllocConnect	SQLFetch
SQLAllocStmt	SQLFreeConnect
SQLBindCol	SQLFreeEnv
SQLBindParameter	SQLFreeStmt
SQLConnect	SQLGetInfo
SQLDisconnect	SQLNativeSQL
SQLError	SQLPrepare
SQLExecDirect	SQLTransact

例の注:

1. DEFAULT(BYVALUE) および LIMITS(EXTNAME(31)) オプションを使用して ODBC プログラムをコンパイルしてください。
2. Windows の場合、サンプル PL/I プログラムは `..¥samples¥` ディレクトリーで提供されています。同じディレクトリーにあるコマンド・ファイル `blododbc.bat` を使用して、テスト・プログラムをコンパイルおよびリンクしてください。
3. ODBC インクルード・ファイルは、`¥include¥` サブディレクトリーで提供されています。

第 18 章 java Dclgen の使用

java Dclgen 用語の理解	307	テーブルの選択と PL/I 宣言の生成	310
PL/I java Dclgen のサポート	308	生成された PL/I 宣言の変更と保管	310
テーブル宣言とホスト構造の作成	309	java Dclgen の終了	311
データベースの選択	309	プログラムへのデータ宣言の組み込み	311

PL/I for Windows には、ユーザーの PL/I アプリケーションで使用可能な DECLARE ステートメントを生成する宣言生成プログラム (java Dclgen) が付属します。

java Dclgen ユーザー

Windows 環境で java Dclgen を使用するためには、Java Developer's Toolkit (V1.3 以降) および DB2 をご使用のシステム上にインストールしておく必要があります。

java Dclgen ツールには、以下の機能があります。

- テーブル宣言を生成し、ユーザーのプログラムに組み込むことができるファイルに格納する。
- データベース・カタログから、テーブルおよびテーブル内の各列の定義に関する情報を取得する。

Java Dclgen は、DB2 Universal Database バージョン 8 以上にアップグレードすると、Lightweight Directory Access Protocol (LDAP) ディレクトリー・サービスをサポートするようになりました。

- その情報を使用して、テーブル (またはビュー) の完全な SQL DECLARE ステートメントおよび一致する PL/I 構造体宣言を生成する。

この宣言をプログラム内で使用するには、SQL INCLUDE ステートメントを使用します。

java Dclgen を起動したい場合に、テーブル名に DBCS 文字が含まれているのであれば、2 バイト文字の入力と表示が可能な端末を使用する必要があります。

java Dclgen 用語の理解

以下に、「java Dclgen」ダイアログ・ボックスで使用される用語を説明します。

テーブル

java Dclgen によって SQL データ宣言を生成したい非修飾テーブル名。任意で、「**テーブル修飾子 (Table Qualifier)**」入力フィールドにテーブル修飾子を入力して、テーブル名を修飾することができます。ツールは、テーブル名とテーブル修飾子から、2 つの部分からなるテーブル名を生成します。

テーブル修飾子 (Table qualifier)

テーブル名修飾子。この値を指定しない場合、ログオン ID がテーブル修飾子とされます。

保管用出力パス (Output Path for Save)

java Dclgen が生成する宣言の宛先となるパス。

保管用出力ファイル名 (Output Filename for Save)

java Dclgen が生成する宣言の宛先となるファイル名。

構造体名 (Structure name)

生成されたデータ構造体の名前。長さは 31 文字以内とします。

このフィールドをブランクにしておいた場合、java Dclgen は、テーブル名またはビュー名に DCL 接頭部を付加したものを含む名前を生成します。テーブル名またはビュー名が DBCS スtringで構成されている場合、接頭部は DBCS 文字が使用されます。

フィールド名接頭部

javaDclgen の出力でのフィールド用に生成された接頭部の名前。選択した値は、長さが最大 28 文字であり、フィールド名の接頭部として使用されます。

例えば、ABCDE を選択すると、生成されるフィールド名は、ABCDE001、ABCDE002 (以下同様) になります。

このフィールドをブランクにしておいた場合、フィールド名はテーブルまたはビューの列名と同じになります。名前が DBCS スtringの場合、接尾部の数字を DBCS で表したものが生成されます。

DECLARE ステートメント内のテーブル名または列名は、名前に特殊文字が含まれていて DBCS スtringでないという場合を除き、区切られていない ID として生成されます。

SQL 予約語を ID として使用している場合、適切な SQL 区切り文字を追加するために java Dclgen 出力を編集する必要があります。

PL/I java Dclgen のサポート

java Dclgen が生成した変数名とデータ属性は、データベースに格納されている情報から派生しています。

表 25. java Dclgen が生成した宣言

SQL データ型	PL/I
SMALLINT	BIN FIXED(15)
INTEGER	BIN FIXED(31)
DECIMAL(p,s) または NUMERIC(p,s)	DEC FIXED(p,s)
FLOAT	BIN FLOAT(53)
CHAR(1)	CHAR(1)
CHAR(n)	CHAR(n)
VARCHAR(n)	CHAR(n) VARYING
LONG VARCHAR	CHAR(32700) VARYING
GRAPHIC(n)	GRAPHIC(n)
VARGRAPHIC(n)	GRAPHIC(n) VARYING
LONG VARGRAPHIC	GRAPHIC(16350) VARYING
DATE	CHAR(10)

表 25. java Dclgen が生成した宣言 (続き)

SQL データ型	PL/I
TIME	CHAR(8)
TIMESTAMP	CHAR(26)
CLOB(nnn)	SQL TYPE IS CLOB(nnn)
BLOB(nnn)	SQL TYPE IS BLOB(nnn)
DBCLOB(nnn)	SQL TYPE IS DBCLOB(nnn)

テーブル宣言とホスト構造の作成

次の 2 つの方法のいずれかで、java Dclgen を始動できます。

1. MS/DOS プロンプトで 'java javaDclgen' と入力する。
2. メイン PL/I プログラム・グループ内の「java Dclgen」アイコンをダブルクリックする。

データベースの選択

ウィンドウが表示され、「データベース (Databases)」リスト・ボックス内に使用可能なデータベースのリストが示されます。データベースを選択するには、マウス・ポインターをデータベース項目に移動し、左マウス・ボタンを 1 回クリックします。これにより、選択した項目が強調表示されます。

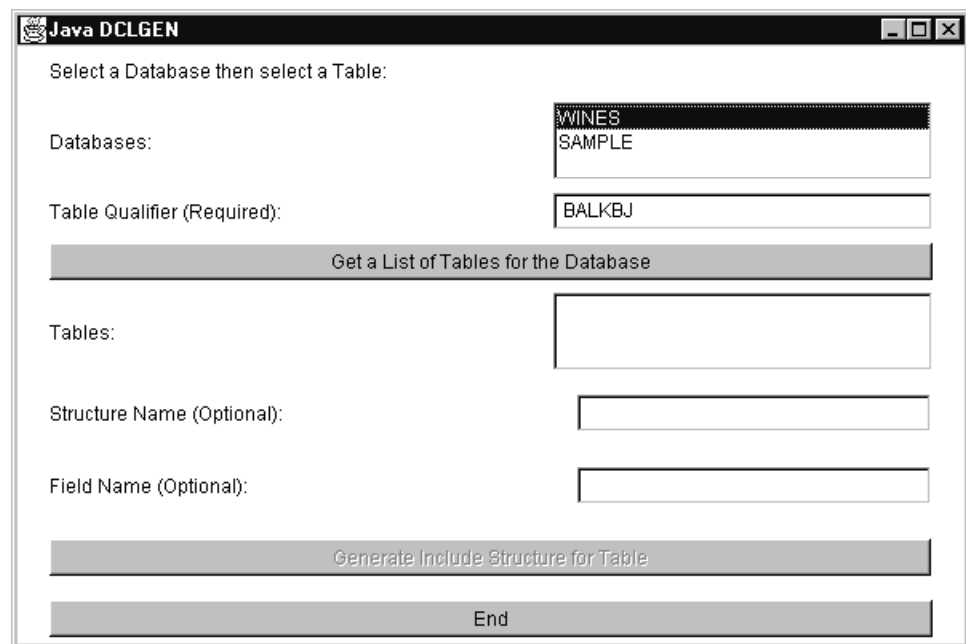


図 24. データベースの選択

「データベース (Databases)」リスト・ボックスのすぐ下に、「テーブル修飾子 (必須) (Table Qualifier (Required))」入力フィールドがあります。このフィールドには、現在のユーザーの ID が入っています。このデフォルト・テーブル修飾子を使用しても、別の有効なテーブル修飾子に置き換えてもかまいません。

操作を続行するには、「データベースのテーブルのリストを取得 (Get a List of Tables for the Database)」ボタンをクリックします。

テーブルの選択と PL/I 宣言の生成

「テーブル」リスト・ボックスには、テーブル修飾子によってデータベース内に作成されたテーブルが表示されます。マウス・ポインターを合わせてクリックすることによって、データベース内のテーブルを選択します。

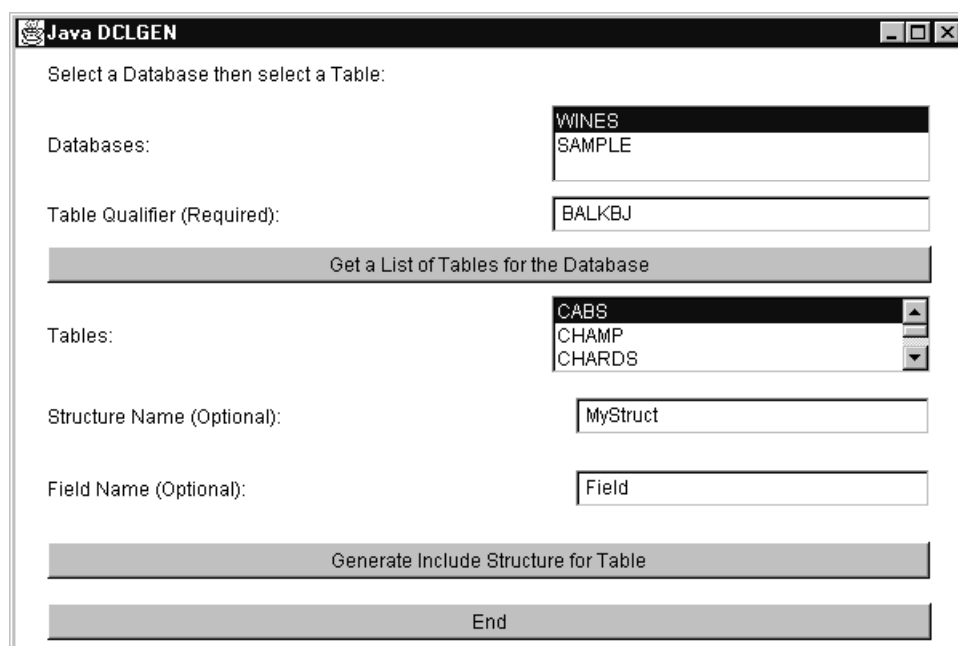


図 25. 修飾子によって作成されたテーブルの表示

「構造名 (Structure Name)」フィールド内でレベル 1 の名前を指定し、同時に構造のレベル 2 の名前のそれぞれに使用されるフィールド名接頭部を指定する方法も選択可能です。例えば、MYSTRUCT フィールド名接頭部として指定した場合、レベル 2 の名前は MYSTRUCT001、MYSTRUCT002、(以下同様) となります。

「テーブルの組み込み構造を生成 (Generate Include Structure for Table)」ボタンをクリックして、次に進みます。

生成された PL/I 宣言の変更と保管

次に開くウィンドウには、生成された PL/I 宣言を示すテキスト域が表示されます。必要に応じて、この領域の内容を直接編集することができます。

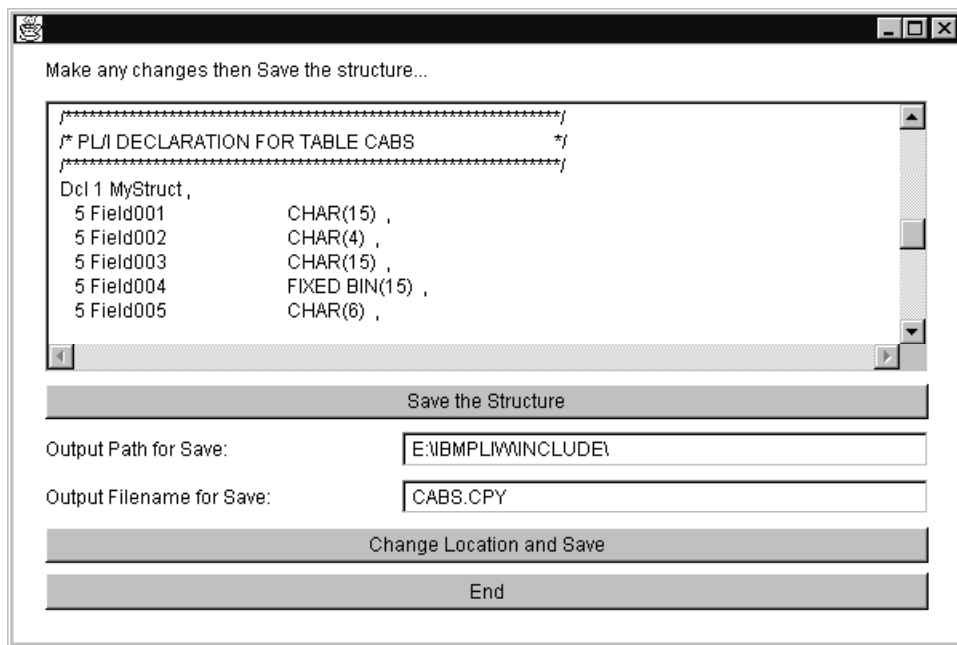


図 26. 生成された PL/I 宣言

PL/I 向けに、java Dclgen は、テーブル名に .CPY という拡張子を付加してファイル名として使用し、ibmpliwininclude 内のファイルにテキスト域の内容を保管します。

生成された宣言をこのディレクトリーとは異なる場所に保管する場合は、「場所を変えて保管 (Change Location and Save)」をクリックします。「別名保管...」ダイアログを使用して、出力ディレクトリーおよびファイル名を変更することができます。

デフォルト情報に上書き入力することによって、テーブル修飾子やエディター名 (拡張子を含む) を変更できます。

注: テーブル名にファイル名で使用されていない特殊文字が含まれている場合は、新規ファイル名を指定する必要があります。

java Dclgen の終了

java Dclgen を終了するには、アプリケーションが終了するまで「終了 (End)」ボタンを何度かクリックします。

プログラムへのデータ宣言の組み込み

次の SQL INCLUDE ステートメントを使用して、java Dclgen が生成したテーブル宣言および PL/I 構造体宣言を、ソース・プログラムに挿入します。

```
exec sql
  include name ;
```

例えば、テーブル BALKBJ.ORG の記述を組み込むには、次のようにコーディングします。

```
exec sql
  include org ;
```

テーブル宣言とホスト構造の作成

何らかの理由で `java Dclgen` の実行結果が予期しないものになっている場合、エディターを使用して固有のニーズに合わせて出力を調整することができます。

第 6 部 上級トピック

第 19 章 Program Maintenance Utility NMAKE の使用

NMAKE を使用する理由	316	定義されたマクロ	325
NMAKE の実行	316	マクロ置換	326
コマンド行の使用	316	特殊マクロ	326
コマンド行構文	316	特殊マクロの例	327
コマンド行ヘルプ	317	ファイル指定の各部分	328
NMAKE コマンド・ファイルの使用	317	特殊マクロを変更する文字	328
コマンド・ファイルを使用する理由	318	変更された特殊マクロの例	328
コマンド・ファイル構文	318	マクロ優先順位規則	329
例	318	推論規則	329
NMAKE オプション	318	特殊機構	330
エラー・ファイルの生成 (/X)	319	推論規則の例	331
すべてのターゲットのビルド (/A)	319	推論規則パス指定	331
メッセージの抑制 (/C)	319	事前定義推論規則	331
変更日の表示 (/D)	319	ディレクティブ	332
環境変数のオーバーライド (/E)	319	ディレクティブの例	334
記述ファイルの指定 (/F)	319	疑似ターゲット	334
ヘルプの表示 (/HELP または /?)	320	事前定義疑似ターゲット	334
終了コードの無視 (/I)	320	.SILENT 疑似ターゲット	334
コマンドの表示 (/N)	320	.IGNORE 疑似ターゲット	335
サインオン・バナーの抑制 (/NOLOGO)	320	.SUFFIXES 疑似ターゲット	335
マクロ定義とターゲット定義の出力 (/P)	320	.PRECIOUS 疑似ターゲット	335
終了コードのリターン (/Q)	320	インライン・ファイル	336
TOOLS.INI ファイルの無視 (/R)	321	インライン・ファイルの例	336
コマンド表示の抑制 (/S)	321	エスケープ文字	337
ターゲット変更日の変更 (/T)	321	コマンドを変更する文字	337
記述ファイル	321	エラー・チェックのオフへの切り替え (-)	338
記述ブロック	321	ダッシュ・コマンド修飾子の例	338
特殊機構	322	コマンド表示の抑制 (@)	338
複数の記述ブロックでのターゲット	323	アットマーク (@) コマンド修飾子の例	338
マクロの使用	323	従属ファイルに対するコマンド実行 (!)	339
マクロの例	324	感嘆符 (!) コマンド修飾子の例	339
特殊機構	324	EXTMAKE 構文	339
記述ファイル内のマクロ	324	TOOLS.INI のマクロと推論規則	340
コマンド行でのマクロ	325	TOOLS.INI の例	340
継承されたマクロ	325		

Program Maintenance Utility (NMAKE) は、プロジェクト・ファイル更新のプロセスを自動化します。NMAKE は、あるファイル・セット (ターゲット・ファイル) の変更日を、別のファイル・セット (従属ファイル) の変更日と比較します。ターゲット・ファイルよりも後に変更された従属ファイルがある場合、NMAKE は一連のコマンドを実行してターゲットを最新の状態にします。

NMAKE を使用する理由

NMAKE は、ソース・ファイルに変更を加えた後でプロジェクトを更新するプロセスを自動化するために最も一般的に使用されます。大規模なプロジェクトでは、ソース・ファイルの数が多くなる傾向があります。多くの場合、変更を加えたときにコンパイルが必要なソース・ファイルの数は少ないものです。記述 ファイル (*makefile*) と呼ばれる特殊なテキスト・ファイルをセットアップして、NMAKE に以下の情報を通知します。

- 他のファイルに従属するファイル
- `compile` コマンドや `link` コマンドなど、プログラムを最新の状態にするために実行する必要のあるコマンド

このような使用法は、NMAKE の能力の一例に過ぎません。適切な記述ファイルを作成することによって、NMAKE を使用して以下の処理を実行することができます。

- バックアップを作成する。
- データ・ファイルを構成する。
- データ・ファイルが変更されたときにプログラムを実行する。

NMAKE の実行

オペレーティング・システムのコマンド行で `nmake` と入力して NMAKE を実行します。次の 2 通りの方法のいずれかによって、NMAKE に入力データを提供します。

- コマンド行で直接データを入力する。
- データをコマンド・ファイル (テキスト・ファイル、応答ファイル と呼ばれる) に入力し、そのファイル名をコマンド行に入力する。

NMAKE 実行中の任意のタイミングで **Ctrl+C** を押すと、オペレーティング・システムに戻ります。

コマンド行の使用

コマンド行で NMAKE を使用する場合、以下の点に留意してください。

- すべてのフィールドはオプションです。
- NMAKE は常に、まず現行ディレクトリで `makefile` という記述ファイルを探します。`makefile` が存在しない場合、NMAKE は `/F` (記述ファイルの指定) オプションと一緒に指定される *filename* を使用します (319 ページの『記述ファイルの指定 (/F)』参照)。

コマンド行構文

```
→ nmake [options] [macrodefinitions] [targets] [/F filename] →
```

options

NMAKE の動作を変更するオプションを指定する。

macrodefinitions

使用する NMAKE 用のマクロ定義をリストする。スペースを含むマクロ定義は二重引用符で囲む必要がある。

targets

ビルドする 1 つ以上のターゲット・ファイルの名前を指定する。ターゲットを指定しない場合、NMAKE は記述ファイル中の最初のターゲットをビルドする。

/F filename

ファイルの従属関係と、ファイルが最新の状態でない場合に実行するコマンドを指定している記述ファイルの名前を指定する。

次の例の意味を以下に示します。

```
nmake /s "program = flash" sort.exe search.exe
```

- /s オプションを使用して NMAKE を起動します。
- マクロを定義し、ストリング "flash" をマクロ "program" に割り当てます。
- sort.exe および search.exe という 2 つのターゲットを指定します。

デフォルトでは、NMAKE は makefile という名前のファイルを記述ファイルとして使用します。

コマンド行ヘルプ

NMAKE のヘルプを表示するには、プロンプトで `nmake /?` と入力します。適切な著作権文が表示されるとともに、以下のように表示されます。

Usage:

```
NMAKE @commandfile
NMAKE /help
```

```
NMAKE [/nologo] [/acdeinpqrst?] [/f makefile] [/x stderrfile]

[macrodefs][targets]
```

What the options stand for

```
/a      Force all targets to be built
/c      Cryptic mode; suppress sign-on banner & warning messages
/d      Display modification dates
/e      Environment variables override macros in the makefile
/i      Ignore exit codes of commands invoked
/n      No execute mode; display commands only
/p      Print macro definitions & target descriptions
/q      Query if target is up to date; for use in batch files
/r      Inference Rules from 'TOOLS.INI' to be ignored
/s      Silent execution of commands
/t      Touch targets with current date & time
/?      Help message
/help   Help message
/nologo Do not display sign-on banner
```

NMAKE コマンド・ファイルの使用

コマンド・ファイルは、NMAKE へのコマンド行入力を拡張するために使用する応答ファイルです。

NMAKE への入力をコマンド行とコマンド・ファイルに分割することもできます。コマンド行において、通常は情報を入力する部分で、コマンド・ファイルの名前(先頭に @ を付加)を使用することができます。

コマンド・ファイルを使用する理由

コマンド・ファイルは、以下のような場合に使用します。

- 複雑で長いコマンドを頻繁に入力する場合。
- マクロ定義などのコマンド行引数のストリングが、コマンド行の長さの制限を超える場合。

コマンド・ファイルは、記述ファイルと同じものではありません。記述ファイルについては、321 ページの『記述ファイル』を参照してください。

コマンド・ファイル構文

コマンド・ファイルを用いて NMAKE に入力データを提供するには、次のように入力します。

```
nmake @commandfile
```

commandfile フィールドには、通常コマンド行で入力されるものと同じ情報を含むファイルの名前を入力します。

NMAKE は、引数の間に現れる改行をスペースと見なします。最後の行以外の行の末尾に円記号 (¥) を付加すれば、マクロ定義を複数行にまたがって記述できます。スペースを含むマクロ定義は、コマンド行で直接入力する場合と同様に引用符で囲む必要があります。

例

次に示すのは、update というコマンド・ファイルです。

```
/s "program ¥  
= flash" sort.exe search.exe
```

このコマンド・ファイルを使用するには、次のコマンドを入力します。

```
nmake @update
```

これにより、以下の項目を使用して NMAKE を実行します。

- /s オプション
- マクロ定義 "program = flash"
- sort.exe および search.exe として指定されたターゲット
- デフォルトの記述ファイル makefile

円記号 (¥) を使用すると、マクロ定義を 2 行にわたって記述することができます。

NMAKE オプション

NMAKE では、いくつかのオプションを使用することができます。オプションを使用する際には、以下の点に留意してください。

- オプションの文字は、大/小文字の区別をしません。/I と /i は同じです。
- オプション文字の前にスラッシュを使用してもダッシュを使用してもかまいません。-a と /a は同じです。

エラー・ファイルの生成 (/X)

構文:/X stderrfile

このオプションは標準エラー・ファイルを生成します。

すべてのターゲットのビルド (/A)

構文:/A

このオプションは、ターゲットが従属ファイルと比較して日付が新しい場合でも、指定されたすべてのターゲットをビルドします。

321 ページの『記述ファイル』を参照。

メッセージの抑制 (/C)

構文:/C

このオプションは、NMAKE サインオン・バナー、致命的でないエラー・メッセージ、および警告メッセージの表示を抑制します。他のメッセージを抑制せずにサインオン・バナーを抑制するためには、/NOLOGO オプションを使用します。

変更日の表示 (/D)

構文: /D

このオプションは、ターゲット・ファイルと従属ファイルの日付をチェックするときに、各ファイルの変更日を表示します。

321 ページの『記述ファイル』を参照。

環境変数のオーバーライド (/E)

構文:/E

このオプションを指定すると、継承されたマクロの再定義はできません。

NMAKE は、すべての現行の環境変数をマクロとして継承します。マクロは記述ファイル内で再定義することができます。/E オプションは、再定義ができないようにします。すなわち、継承されたマクロは常に環境変数の値を持ちます。

記述ファイルの指定 (/F)

構文:/F filename

このオプションは、使用する記述ファイルの名前として *filename* を指定します。ファイル名の変わりにダッシュ (-) を入力した場合、NMAKE は標準入力装置 (通常はキーボード) から記述ファイルを読み取ります。

ファイル名が指定されない場合は、デフォルトの makefile を使用します。

ヘルプの表示 (/HELP または /?)

構文: /HELP or /?

このオプションは、NMAKE 構文の簡潔な要約を表示します。

終了コードの無視 (/I)

構文:/I

このオプションは、NMAKE が呼び出したコンパイラまたはリンカーから戻された終了コード (エラー・レベル・コードまたは戻りコードとも呼ばれる) を無視します。このオプションが指定されない場合は、何らかのプログラムがゼロ以外の終了コードを戻したときに NMAKE は終了します。

コマンドの表示 (/N)

構文:/N

このオプションは NMAKE コマンドを表示しますが、実行はしません。次のような場合に /N を使用してください。

- 従属ファイルと比べて古いターゲットをチェックする。
- 記述ファイルをデバッグする。

サインオン・バナーの抑制 (/NOLOGO)

構文:/NOLOGO

このオプションは、NMAKE 始動時のサインオン・バナーの表示を抑制します。致命的でないエラー・メッセージと警告メッセージも同様に抑制したい場合は、メッセージの抑制 (/C) オプションを使用してください。

マクロ定義とターゲット定義の出力 (/P)

構文:/P

このオプションは、すべてのマクロ定義とターゲット定義を出力します。出力は、標準出力装置 (通常はディスプレイ) に送られます。

終了コードのリターン (/Q)

構文:/Q

このオプションを指定すると、NMAKE は次のいずれかのコードを戻します。

- NMAKE 実行時にビルドされたすべてのターゲットが最新である場合は、ゼロの終了コード
- ターゲットが最新でない場合は、ゼロ以外の終了コード

バッチ・ファイル内から NMAKE を実行する場合には、このオプションを使用してください。

TOOLS.INI ファイルの無視 (/R)

構文:/R

このオプションは、以下を無視します。

- TOOLS.INI ファイルに含まれるすべての推論規則とマクロ
- すべての事前定義推論規則とマクロ

コマンド表示の抑制 (/S)

構文:/S

このオプションは、NMAKE によるコマンド実行時のコマンドの表示を抑制します。コマンド自体によって生成されたメッセージの表示は抑制しません。

/N コマンド (コマンドの表示) は、/S オプションよりも優先します。/N と /S を一緒に使用した場合、コマンドは表示されますが、実行されません。

ターゲット変更日の変更 (/T)

構文:/T

このオプションは、日付の古いターゲット・ファイルの変更日を現在の日付に変更 (「タッチ」) します。コマンドは実行されず、ターゲット・ファイルは変更されません。

記述ファイル

NMAKE は、記述ファイルを使用して実行する処理を決定します。最も単純な形態では、記述ファイルは、他のファイルに従属するファイルと、ファイルが変更された場合に実行する必要のあるコマンドを、NMAKE に通知します。

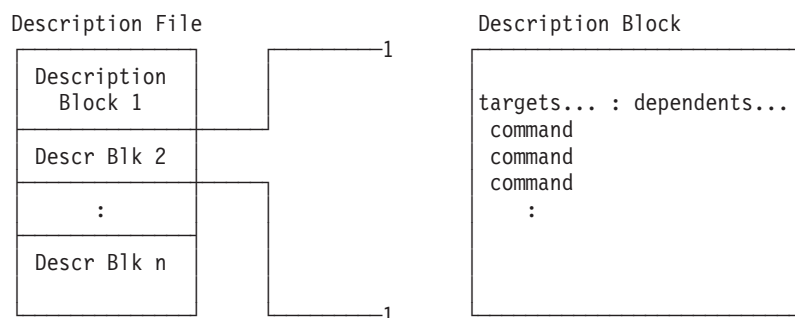
記述ファイルの内容は、次のようになります。

```
targets...: dependents...
      command
      :

targets... : dependents...
      command
```

記述ブロック

ファイル間の従属関係は、記述ブロック で定義されます。記述ブロックは、プログラムのさまざまな部分間の関係を指示します。すべてのコンポーネントを最新の状態にするコマンドを含んでいます。記述ファイルは、最大 1048 個の記述ブロックを含むことができます。



特殊機構

以下に記述ファイルと記述ブロックの特殊機構を示します。

- 記述ファイルは、マクロ定義を含むことができ、また、記述ブロック内でマクロを使用することができます。マクロによって、あるテキスト・ストリングを別のテキスト・ストリングに容易に置換することができます。
- 記述ファイルには、推論規則を記述することができます。推論規則が指定されると、NMAKE は、ターゲットおよび従属ファイルで用いられるファイル名拡張子に基づいて、どのコマンドを使用するかを推論することができます。
- 次の構文を使用して、NMAKE が従属ファイルを検索するディレクトリーを指定できます。

```
targets : {directory1;directory2...}dependents
```

NMAKE は、最初に現行ディレクトリーを検索し、次に *directory1*、*directory2*、(以下同様) というように検索していきます。

- セミコロン (;) を使用すると、ターゲット・ファイルや従属ファイルと同じ行にコマンドを記述できます。

```
targets... : dependents... ; command
```

- 各行の末尾に円記号 (¥) を付加すれば、長いコマンドを複数行にまたがって記述できます。

```
command ¥
  continuation of command
```

- コマンドの実行は、コマンドの前に特殊文字を付加すれば変更可能です。
- 記述ブロックでコマンドを指定しない場合、NMAKE は推論規則を検索してターゲットをビルドします。
- ワイルドカード文字 (* および ?) は、記述ブロックで使用可能です。例えば、次の記述ブロックは、.PLI 拡張子の付いたすべてのソース・ファイルをコンパイルします。

```
astro.exe : *.pli
  pli $**
```

- NMAKE は、*.pli という指定を、現行ディレクトリー内のすべての PL/I ファイルのリストに展開します。\$** は、現行のターゲットに対して指定されたすべての従属ファイルのリストです。
- NMAKE は、構文の中で数種類の句読文字を使用します。これらの文字の 1 つをリテラル文字として使用するには、その前にエスケープ文字 (^) を付加します。句読文字のリストについては、337 ページの『エスケープ文字』を参照してください。

- ターゲット・ファイルは、通常 1 つの記述ブロック内にもみ記述されます。特殊な構文によって、ターゲットを複数の記述ブロックで使用できます。
- 特殊な構文によって、記述ブロック内の最初の従属ファイルのドライブ、パス、ベース名、および拡張子を判別できます。

複数の記述ブロックでのターゲット

1 つのファイルを複数の記述ブロックでターゲットとして使用すると、NMAKE は終了します。この制限は、ターゲット/従属ファイルの区切り文字としてコロンを 1 つではなく、2 つ (::) 使用することによって克服することができます。

以下のような記述ブロックは許可されます。

```
X :: A
    command
X :: B
    command
```

次の記述ブロックでは、NMAKE が終了します。

```
X : A
    command
X : B
    command
```

ターゲット/従属ファイルの行が同じコマンドの上にグループ化されている場合は、単一コロンの使用が認められます。次の記述ブロックは認められます。

```
X : A
X : B
    command
```

2 重コロン (::) ターゲット/従属ファイルの区切り文字の例

```
target.lib :: a.asm b.asm c.asm
    ml a.asm b.asm c.asm
    ilib target a.obj b.obj c.obj

target.lib :: d.pli e.pli
    pli d.pli
    pli e.pli
    ilib target d.obj e.obj
```

これら 2 つの記述ブロックは、どちらも target.lib という名前のライブラリーを更新します。アセンブリ言語ファイルの中に、ライブラリー・ファイルよりも後に変更されたものがある場合、NMAKE は最初のブロックのコマンドを実行し、ソース・ファイルをアセンブルしてライブラリーを更新します。同様に、PL/I 言語ファイルに変更されたものがある場合、NMAKE は 2 番目のグループのコマンドを実行し、PL/I ファイルをコンパイルしてライブラリーを更新します。

マクロの使用

マクロは、記述ファイル内のあるストリングを別のストリングに置き換えるために便利な手段です。テキストは、NMAKE が実行されるたびに自動的に置き換えられます。この機能は、記述ファイル全体でのテキストの変更を容易にします。そのテキストを使用するすべての行を編集する必要はありません。

マクロは一般に、以下の 2 通りの用途があります。

- 複数のプロジェクト用に標準記述ファイルを作成するため。マクロは、コマンド内のファイル名を表します。これらのファイル名は、NMAKE の実行時に定義されます。異なるプロジェクトに切り替える場合に、マクロを変更すれば、NMAKE の使用するファイル名が、記述ファイル全体で変更されます。
- NMAKE がコンパイラ、アセンブラ、またはリンカーに渡すオプションを制御するため。マクロを使用してオプションを指定すると、1 つの簡単なステップにより、記述ファイル全体で素早くオプションを変更することができます。

マクロは、以下の場所で定義できます。

- 記述ファイル
- コマンド行
- TOOLS.INI
- 環境変数からの継承を通じて

マクロの例

```
program = flash
c = ilink
options =

$(program).exe : $(program).obj
    $c $(options) $(program).obj;
```

上記の例は、3 つのマクロを定義しています。記述ブロックは、次のコマンドを実行します。

```
flash.exe : flash.obj
    ilink flash.obj;
```

特殊機構

マクロには、以下の特殊機構があります。

- マクロの使用時に、マクロ自体の中のテキストを置き換えることができます。
- 特別な目的のために、いくつかのマクロが事前定義されています。
- マクロが複数回定義された場合は、優先順位規則によって、どの定義を用いるかを決定します。
- マクロを TOOLS.INI ファイルに記述することもできます。

記述ファイル内のマクロ

マクロを使用する前に、NMAKE コマンド行または記述ファイル内でマクロを定義する必要があります。記述ファイルのマクロ定義は、次のようになります。

```
macroname = macrostring
```

マクロ名は、英数字および下線文字 (_) の任意の組み合わせで指定できます。大文字と小文字は区別されます。マクロ・ストリングは任意の文字ストリングです。

マクロ名の最初の文字が、その行の最初の文字でなければなりません。NMAKE は、等号 (=) の前後のスペースを無視します。

マクロ・ストリングは、ヌル・ストリングでもかまいませんし、埋め込みスペースが入っていてもかまいません。記述ファイルでは、マクロ・ストリングを引用符で囲まないでください。コマンド行でマクロを定義する場合のみ引用符を使用します。

コマンド行でのマクロ

マクロを使用する前に、NMAKE コマンド行または記述ファイル内でマクロを定義する必要があります。コマンド行マクロ定義は、次のようになります。

```
macroname=macrostring
```

等号の両側にスペースを入れてはなりません。スペースを埋め込んだ場合、NMAKE はマクロを誤って解釈する場合があります。マクロ・ストリングに埋め込みスペースが含まれる場合は、次のようにそれを二重引用符 (") で囲みます。

```
macroname="macro string"
```

次のようにマクロ定義全体を二重引用符 (") で囲むこともできます。

```
"macroname = macro string"
```

マクロ名は、英数字および下線文字 (_) の任意の組み合わせで指定できます。大文字と小文字は区別されます。マクロ・ストリングは任意の文字ストリングまたはヌル・ストリングです。

継承されたマクロ

NMAKE は、すべての現行の環境変数をマクロとして継承 します。例えば、PATH 環境変数を PATH = C:¥TOOLS¥BIN と定義した場合、記述ファイル内の PATH を使用すると、ストリング C:¥TOOLS¥BIN に置き換えられます。

記述ファイル内に上述の例のような行を加えることによって、継承されたマクロを再定義することができます。NMAKE の実行中は、マクロは再定義された定義を使用します。ただし、NMAKE が終了すると、環境変数は元の値に戻ります。

環境変数の指定変更 (/E) オプションを指定すると、継承されたマクロの再定義はできません。このオプションを使用した場合、継承されたマクロの再定義を試みても、NMAKE に無視されます。

定義したマクロすべてについて、マクロ名では大/小文字を区別します。例えば、次のようなマクロを見てみましょう。

```
UPPER=UpperCase
```

この例では、\$(UPPER) は値を戻しますが、\$(upper) は戻しません。継承されたマクロ名 (すなわち環境変数から自動的に作成されたマクロ名) は、常に UPPERCASE でなければなりません。

定義されたマクロ

マクロを定義すると、以下の構文を用いて記述ファイルの任意の場所でそのマクロを使用することができます。

```
$(macroname)
```

マクロ名の長さが 1 文字のみの場合は、括弧は必要ありません。マクロを使用せずにドル記号 (\$) を使用するには、2 つのドル記号 (\$\$) を入力するか、またはドル記号の前に脱字記号 (^) をエスケープ文字として使用します。

NMAKE を実行すると、検出された \$(macroname) はすべて、定義されたマクロ・ストリングに置き換えられます。マクロが定義されていない場合は、何も置き換えられません。マクロの定義後にそれを取り消すことができる手段は、!UNDEF ディレクティブのみです。

マクロ置換

マクロを使用して記述ファイル内のテキストを置換するのと同様に、次の構文を使用してマクロ内のテキストを置換します。

```
$(macroname: string1 = string2)
```

macroname 内で検出された *string1* はすべて、*string2* に置き換えられます。コロンと *string1* の間のスペースは、*string1* の一部と見なされます。*string2* がヌル・ストリングの場合、検出された *string1* はすべてマクロから削除されます。コロン (:) は *macroname* の直後に記述する必要があります。

マクロ内の *string1* を *string2* に置き換えても、その変更は永続的ではありません。次にそのマクロを置換を行わずに使用した場合、未変更の元のマクロになります。

例

```
SOURCES = one.pli two.pli three.pli
program.exe : $(SOURCES:.pli=.obj)
    ilink **;
```

上記の例では、SOURCES というマクロを定義しており、3 つの PL/I ソース・ファイルの名前が含まれています。このマクロを使用すると、ターゲット行/従属行では .pli 拡張子を .obj に置換します。そのため、NMAKE は次のコマンドを実行します。

```
ilink one.obj two.obj three.obj;
```

\$\$ は、指定されたターゲットのすべての従属ファイルに変換される特殊なマクロです。

特殊マクロ

NMAKE では、いくつかのマクロが事前定義されています。下記の最初の 6 つのマクロは、記述ブロックのターゲット行/従属行内のファイルについて、1 つ以上のファイル指定を戻します。注記がある場合を除き、ファイル指定には、ファイルのパス、ベース・ファイル名、およびファイル名拡張子が含まれています。

マクロ 値

\$@ ターゲット・ファイルを指定。

\$* ターゲット・ファイルのベース名 (拡張子なし)。ターゲット・ファイル名の一部としてパスが指定された場合は、パス情報も戻されます。このマクロは、従属リスト内では使用できません。

\$\$\$ 従属ファイルを指定。

- \$?** ターゲットと比較して日付の古い従属ファイルのみを指定。
- \$<** ターゲットと比較して日付の古い単一の従属ファイルを指定。このマクロは推論規則でのみ使用されます。
- \$\$@** NMAKE が現在評価中のターゲットのファイル指定。これは動的な従属関係パラメーターで、従属リストでのみ使用されます。
- \$(AS)** スtring MASM。マクロ・アセンブラー (MASM) を実行するコマンドです。このマクロを再定義して、異なるコマンドを使用することができます。

\$(MAKE)

NMAKE を実行するために使用するコマンド名。このマクロは、NMAKE を再帰的に呼び出すために使用します。このマクロを再定義する場合、NMAKE は警告メッセージを出します。

コマンドの表示オプション (/N) がオンの場合でも、NMAKE は \$(MAKE) が出現するコマンド行を実行します。

\$(MAKEFLAGS)

現在有効な NMAKE オプション。このマクロを再定義することはできません。

特殊マクロ **\$\$\$** および **\$\$@** だけは、2 文字以上のマクロ名は括弧で囲まなければならないという規則の例外です。

このリストの最初の 6 つのマクロには、文字を追加してマクロの意味を変更することができます。ただし、これらのマクロでマクロ置換を使用することはできません。

特殊マクロの例

```
trig.lib : sin.obj cos.obj arctan.obj
!ilib trig.lib $?
```

この例では、マクロ **\$?** は、ターゲット・ファイルと比較して日付の古いすべての従属ファイルの名前を表します。ILIB コマンドの前に感嘆符 (!) を使用すると、NMAKE はリスト内のファイルごとに ILIB コマンドを 1 回ずつ実行します。このように記述すると、ILIB コマンドの発行により、NMAKE はリスト内のファイルごとに ILIB コマンドを 1 回ずつ実行します。このように記述した結果、ILIB コマンドは 3 回まで実行され、実行のたびにモジュールを新しいバージョンに置き換えます。

```
DIR=c:\include
$(DIR)\globals.inc : globals.inc
copy globals.inc $@
$(DIR)\types.inc : types.inc
copy types.inc $@
$(DIR)\macros.inc : macros.inc
copy macros.inc $@
```

この例では、インクルード・ファイルのグループを更新する方法を示しています。ディレクトリー c:\include 内の globals.inc、types.inc、および macros.inc の各ファイルは、現行ディレクトリー内の対応するファイルに従属します。インクルード・ファイルの 1 つの日付が古い場合、NMAKE はそれを現行ディレクトリーの同じ名前のファイルで置き換えます。

特殊マクロ

特殊マクロ `$$@` を使用する次の記述ファイルは等価です。

```
DIR=c:\include
$(DIR)\globals.inc $(DIR)\types.inc $(DIR)\macros.inc : $$(@F)
!copy $? $@
```

特殊マクロ `$$(@F)` は、現行のターゲットのファイル名 (パスは含まない) を示します。

NMAKE は記述ブロックを評価するとき、3 つのターゲットを 1 度に 1 つずつ従属ファイルと比較して評価します。そのため、NMAKE はまず `c:\include\globals.inc` の日付が現行ディレクトリーの `globals.inc` と比較して古いかどうかをチェックします。日付が古い場合、従属ファイル `globals.inc` をターゲットへコピーするコマンドを実行します。NMAKE は、他の 2 つのターゲットについても、この手順を繰り返します。

コマンド行では、マクロ `$$?` はこのターゲットの従属ファイルを参照します。マクロ `$$@` は、ターゲット・ファイルの完全なファイル指定です。

ファイル指定の各部分

完全なファイル指定では、ファイルのベース名、ファイル名拡張子、およびパスを示します。パスは、ディスク・ドライブ ID と、ディスク上でファイルを見付けるために必要な一連のディレクトリーを示します。

例えば、次のようなファイル指定を考えます。

```
c:\source\prog\sort.obj
```

これは以下の部分に分けられます。

Path Name	c:\source\prog
Base File Name	sort
File-Name Extension	.obj

特殊マクロを変更する文字

以下の 6 つのマクロは、解決されるとすべてファイル指定になります (`$$$` と `$$?` は複数のファイル指定になる可能性があります)。

`$$*` `$$@` `$$$` `$$<` `$$?` `$$$@`

これらのマクロには、文字を追加してマクロが戻すファイル名を変更することができます。使用する文字によって、完全なファイル指定のうちのどの部分が戻されるかが異なります。

File Part Returned	Appended Character			
	D	F	B	R
File Path	Yes	No	No	Yes
Base File Name	No	Yes	Yes	Yes
File Name Extension	No	Yes	No	No

変更された特殊マクロの例

マクロ `$$@` が次の値を持つ場合を考えます。

```
c:\source\prog\sort.obj
```

変更されたマクロに対して以下の値が戻されます。

マクロ 値

`$(@D)` `c:%source%prog`

`$(@F)` `sort.obj`

`$(@B)` `sort`

`$(@R)` `c:%source%prog%sort`

変更されたマクロは、常に 2 文字以上となるため、使用するとき括弧で囲む必要があります。

マクロ優先順位規則

同じマクロが 2 箇所以上で定義された場合、最も優先順位の高い定義が使用されます。

優先順位

定義

1 (最高)

コマンド行

2 記述ファイル

3 環境変数

4 TOOLS.INI ファイル

5 (最低)

事前定義マクロ (CC、AS など)

マクロ定義のオーバーライド (/E) オプションを用いて NMAKE を呼び出した場合、環境変数によって定義されたマクロの方が、記述ファイルで定義されたマクロよりも優先します。

推論規則

推論規則は、コマンドが与えられない場合に記述ブロックをどのように処理するかを NMAKE が推論するためのテンプレートです。 .SUFFIXES リストで定義されている拡張子のみに対して、推論規則を使用することができます。拡張子 .c、.obj、.asm、および .exe は、自動的に .SUFFIXES に組み込まれます。

PL/I プログラマー

.SUFFIXES 疑似ターゲットを使用して手動で PL/I ファイル拡張子を追加する必要があります。 335 ページの『.SUFFIXES 疑似ターゲット』を参照してください。

NMAKE が、コマンドを含まない記述ブロックを検出した場合、2 つのファイル拡張子を与えて、従属ファイルからターゲットを作成する方法を指定する推論規則を探します。同様に、従属ファイルが存在しない場合は、NMAKE は同じベース名を持つ別のファイルから従属ファイルを作成する方法を指定する推論規則を探します。

NMAKE は、作成しようとするファイルのベース名が、既に存在するファイルのベース名と一致する場合のみ、推論規則を適用します。

実際、推論規則は、従属ファイルを表す拡張子を持つファイルと、ターゲット・ファイルを表す拡張子を持つファイルの間に 1 対 1 対応がある場合のみに有用です。例えば、ライブラリーに複数のモジュールを挿入する推論規則を定義することはできません。

推論規則を使用することにより、いくつかの記述ブロックに同じコマンドを記述する必要がなくなります。例えば、推論規則を使用して、`pli` コマンドを 1 つだけ指定し、そのコマンドがすべての `PL/I` ソース・ファイル (`.pli` 拡張子を持つ) をオブジェクト・ファイル (`.obj` 拡張子を持つ) に変えるようにすることができます。

次の書式のテキストを記述ファイルまたは `TOOLS.INI` に追加することによって、推論規則を定義します。『特殊機構』を参照してください。

```
.fromext.toext:  
commands  
:
```

推論規則の要素は以下のとおりです。

fromext

ターゲットをビルドするための従属ファイルのファイル名拡張子。

toext ビルドするターゲット・ファイルのファイル名拡張子。

commands

fromext 従属ファイルから *toext* ターゲットをビルドするコマンド。

例えば、`PL/I` ソース・ファイル (`.pli` 拡張子を持つ) を `PL/I` オブジェクト・ファイル (`.obj` 拡張子を持つ) に変換する推論規則は、次のようになります。

```
.pli.obj:  
pli $<
```

特殊マクロ `$<` は、ターゲットと比較して日付の古い従属ファイルの名前を表します。

特殊機構

- NMAKE がターゲット・ファイルと従属ファイルを探すパスを、推論規則内で指定することができます。
- 推論規則は、C プログラムのコンパイルとリンク、およびプログラムのアセンブル用に、事前定義されています。
- NMAKE は、記述ファイル内で推論規則が見つからない場合、`TOOLS.INI` ファイル内で推論規則を探します。
- `.SUFFIXES` リストで定義されている拡張子のみに対して、推論規則を使用することができます。拡張子 `.c`、`.obj`、`.asm`、および `.exe` は、自動的に `.SUFFIXES` に組み込まれます。
- `.SUFFIXES` 疑似ターゲットを使用して手動で `PL/I` ファイル拡張子を追加する必要があります。 335 ページの『`.SUFFIXES` 疑似ターゲット』を参照してください。

推論規則の例

```
.obj.exe:
    ilink $<;

example1.exe: example1.obj

example2.exe: example2.obj
    ilink /co example2,,libv3.lib
```

上記の最初の行で定義されている推論規則は、対応するオブジェクト・ファイルに変更が加えられるたびに **ILINK** コマンドが実行可能ファイルを作成するというものです。推論規則内のファイル名が特殊マクロ `$<` で指定されているため、推論規則は実行可能ファイルの日付の方が古い `.obj` ファイルすべてに適用されます。

NMAKE は、最初の記述ブロック内でコマンドを検出しなかった場合、適用可能な規則があるかどうかを調べ、記述ファイルの最初の 2 行で定義される規則を見つけ出します。**NMAKE** はその規則を適用し、コマンド実行時に `$<` を `example1.obj` に置き換えるため、**ILINK** コマンドは次のようになります。

```
ilink example1.obj;
```

NMAKE は、2 番目の記述ブロックを調べる際には推論規則を検索しません。コマンドが明示的に与えられているためです。

推論規則パス指定

推論規則を定義するときに、**NMAKE** に対してターゲット・ファイルおよび従属ファイルを探す場所を指示することができます。以下の構文を使用します。

```
{frompath}.fromext{topath}.toext
commands
:
```

NMAKE は、*fromext* 拡張子を持つファイルを *frompath* で指定されたディレクトリ内で探します。**NMAKE** は、*topath* によって指定されたディレクトリ内の *toext* 拡張子を持つファイルをビルドするコマンドを実行します。

事前定義推論規則

NMAKE では、いくつかの推論規則が事前定義されています。

表 26. **NMAKE** 事前定義推論規則

推論規則	コマンド・アクション	デフォルト値
<code>.c.obj</code>	<code>\$(CC) \$(CFLAGS) /c \$*.c</code>	<code>icc /c \$*.c</code>
<code>.c.exe</code>	<code>\$(CC) \$(CFLAGS) \$*.c</code>	<code>icc \$*.c</code>
<code>.asm.obj</code>	<code>\$(AS) \$(AFLAGS) \$*;</code>	<code>masm \$*;</code>

- 最初の 2 つの規則は自動的に C プログラムをコンパイルし、リンクします。
- 最後の規則は自動的にプログラムをアセンブルします。
- 上記の規則は、最も多く使用される事前定義推論規則です。事前定義推論規則をすべてリストするには、`makefile` を実行して、`/p` オプションを指定します。すべての使用可能な推論規則が表示されます。

ディレクティブ

ディレクティブを使用すると、バッチ・ファイルに似た記述ファイルを構成することができます。NMAKE では、以下の機能を持つディレクティブを提供します。

- 条件付でコマンドを実行する。
- エラー・メッセージを表示する。
- 他のファイルの内容を組み込む。
- いくつかの NMAKE オプションをオンまたはオフに切り替える。

各ディレクティブは、記述ファイルの最初の列にあり、先頭は感嘆符 (!) から始まっています。感嘆符とディレクティブ・キーワードの間にスペースがあってもかまいません。

以下のリストで、ディレクティブを説明します。

!IF *expression*

expression がゼロ以外の値と評価される場合、!IF キーワードと、次の !ELSE または !ENDIF ディレクティブの間のステートメントを実行します。

!IF ディレクティブと共に使用される *expression* は、整数定数、ストリング定数、またはプログラムによって戻される終了コードで構成することができます。整数定数は、数値符号反転 (-)、1 の補数 (~)、および論理否定 (!) の C の単項演算子を使用することができます。以下にリストする C の 2 項演算子も使用可能です。

演算子 説明

+	加算
-	減算
*	乗算
/	除算
%	モジュラス
&	ビット単位の AND
	ビット単位の OR
^^	ビット単位の XOR
&&	論理 AND
	論理 OR
<<	左シフト
>>	右シフト
==	等しい
!=	等しくない
<	より小
>	より大
<=	より小または等しい

>= より大または等しい

- 括弧を使用して式をグループ化することができます。
- 先頭に 0 を付けて指定されている (8 進) か、0x をつけて指定されている (16 進) 場合を除き、値は 10 進数と見なされます。
- スtringは引用符 (") で囲みます。等号演算子 (==) および非等号演算子 (!=) を使用して 2 つのStringを比較することができます。
- プログラム名を大括弧 ([]) で囲むことによって、式の中でプログラムを呼び出すことができます。プログラムによって戻される終了コードが、式で使用されます。

!ELSE !ELSE ディレクティブの前のステートメントが実行されなかった場合、
!ELSE および !ENDIF ディレクティブの間のステートメントを実行します。

!ENDIF

ステートメントの !IF、!IFDEF、または !IFNDEF ブロックの終わりをマークします。

!IFDEF *macroname*

記述ファイル内で *macroname* が定義されている場合、!IFDEF キーワードと、次の !ELSE または !ENDIF ディレクティブの間のステートメントを実行します。マクロがヌルとして定義された場合でも、定義されているものと見なされます。

!IFNDEF *macroname*

記述ファイル内で *macroname* が定義されていない場合、!IFNDEF キーワードと、次の !ELSE または !ENDIF ディレクティブの間のステートメントを実行します。

!UNDEF *macroname*

前に定義されたマクロの定義を解除します。

!ERROR *text*

テキストを出力して、実行を停止します。

!INCLUDE *filename*

ファイル *filename* を読み取り、評価してから、現行の記述ファイルの処理を再開します。*filename* が不等号括弧 (<>) で囲まれている場合、NMAKE は INCLUDE マクロによって指定されたディレクトリー内でファイルを探します。それ以外の場合は、現行ディレクトリーのみでファイルを探します。INCLUDE マクロは、最初に INCLUDE 環境変数の値に設定されます。

!CMDSWITCHES {+|-}*opt*

/D、/I、/N、および /S の 4 つの NMAKE オプションのいずれかをオンまたはオフに切り替えます。オプションが指定されない場合は、これらのオプションを NMAKE 始動時の値にリセットします。オプションをオンにするには、そのオプションの前に正符号 (+) を付加します。オフにする場合は、前に負符号 (-) を付加します。このディレクティブによって、MAKEFLAGS マクロが更新されます。

326 ページの『特殊マクロ』を参照。

ディレクティブの例

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe:winner.obj
!IFDEF DEBUG
! IF "$ (DEBUG)"=="y"
    ilink /de winner.obj;
! ELSE
    ilink winner.obj;
! ENDIF
!ELSE
! ERROR Macro named DEBUG is not defined.
!ENDIF
```

この例のディレクティブは、以下の処理を行います。

- **!INCLUDE** ディレクティブは、ファイル `infrules.txt` を読み込み、記述ファイルの一部であるかのように評価します。
- **!CMDSWITCHES** ディレクティブは、`/D` オプションをオンにします。これにより、各ファイルがチェックされるときにファイルの日付が表示されます。
- `winner.obj` と比較して `winner.exe` の日付が古い場合、**!IFDEF** ディレクティブはマクロ `DEBUG` が定義されているかどうかを調べます。定義されている場合、**!IF** ディレクティブはそれが `y` に設定されているかどうかを調べます。`y` の場合は、リンカーが `/DE` オプションを用いて呼び出されます。`y` 以外の場合は、`/DE` オプションを指定せずにリンカーが呼び出されます。`DEBUG` マクロが定義されていない場合は、**!ERROR** ディレクティブがメッセージを出力し、**NMAKE** は実行を停止します。

疑似ターゲット

疑似ターゲット は、記述ブロック内の、ファイルでないターゲットです。疑似ターゲットは、ファイルのグループをビルドするか、またはコマンドのグループを実行するための「ハンドル」として機能する名前です。次の例では、`UPDATE` は疑似ターゲットです。

```
UPDATE: *.*
!copy $** a:\product
```

NMAKE は、疑似ターゲットを評価するとき、常にその従属ファイルの日付が古いものと見なします。上記の例では、**NMAKE** は従属ファイルのそれぞれを指定されたドライブおよびディレクトリーにコピーします。

NMAKE では、特別な目的のためにいくつかの疑似ターゲットが事前定義されています。

『事前定義疑似ターゲット』 を参照。

事前定義疑似ターゲット

NMAKE は、記述ファイル内部で特別な規則を規定する疑似ターゲットをいくつか事前定義しています。

.SILENT 疑似ターゲット

構文: `.SILENT : dependents...`

この疑似ターゲットは、単一の記述ブロックについて、実行されたコマンドの表示を抑制します。*/S* オプションを指定すると、すべての記述ブロックで、実行されたコマンドの表示を抑制します。

321 ページの『コマンド表示の抑制 (*/S*)』を参照。

.IGNORE 疑似ターゲット

構文: `.IGNORE : dependents...`

この疑似ターゲットは、単一の記述ブロックについて、プログラムから戻された終了コードを無視します。*/I* オプションを指定すると、すべての記述ブロックで、プログラムから戻された終了コードを無視します。

320 ページの『終了コードの無視 (*/I*)』を参照。

.SUFFIXES 疑似ターゲット

構文: `.SUFFIXES : extensions...`

この疑似ターゲットは、従属ファイルの指定されていないターゲット・ファイルをビルドする必要がある場合に `NMAKE` が処理を試みる対象の、ファイル拡張子を定義します。`NMAKE` は、現行ディレクトリー内で、ターゲット・ファイルと同じ名前前で `<extensions...>` に含まれる拡張子を持つファイルを探します。`NMAKE` がこのようなファイルを見つけ出した場合、推論規則がこのファイルに適用されるのであれば、`NMAKE` はそのファイルをターゲットの従属ファイルとして扱います。

`.SUFFIXES` 疑似ターゲットは、次のように事前定義されています。

`.SUFFIXES : .obj .exe .c .asm`

このリストに拡張子を追加するには、`.SUFFIXES :` の後に新規の拡張子を記述して指定します。例えば次のように記述することによって、`PL/I` ソース・ファイル用の推論規則を作成することができます。

`.SUFFIXES: .pli`

リストをクリアするには、次のように指定します。

`.SUFFIXES:`

`.SUFFIXES` で指定された拡張子に対してのみ、推論規則が適用されます。拡張子が `.SUFFIXES` リスト内で指定されていない場合、`NMAKE` は推論規則を無視します。

.PRECIOUS 疑似ターゲット

構文: `.PRECIOUS : targets...`

この疑似ターゲットは、ターゲットをビルドするコマンドが強制終了または中断した場合でも、ターゲットを削除しないように `NMAKE` に指示します。この疑似ターゲットは、`NMAKE` のデフォルトをオーバーライドします。デフォルトでは、ターゲットが正常にビルドされたことを確認できない場合、`NMAKE` はターゲットを削除します。

次に例を示します。

```
.PRECIOUS : tools.lib
tools.lib : a2z.obj z2a.obj
command
:
```

`tools.lib` をビルドするコマンドが中断され、不完全なファイルが残された場合、`NMAKE` は途中までビルドされた `tools.lib` を削除しません。

疑似ターゲット `.PRECIOUS` は、限られた状況でのみ有用です。ほとんどの商用開発ツールは、独自の割り込みハンドラーを持っており、エラー発生時に「クリーンアップ」を行います。

インライン・ファイル

記述ファイル内で、オペレーティング・システムのコマンド行の制限を超える引数のリストを含むコマンドの発行が必要となる場合があります。`NMAKE` は、コマンド・ファイルの使用をサポートすると同様に、他のプログラムが応答ファイルとして読み込むインライン・ファイルも生成することができます。

インライン・ファイルを生成するためには、記述ブロックで以下の構文を使用します。

```
target : dependents
    command @<<[filename]
inline file text
<< [KEEP | NOKEEP]
```

2 セットの不等号 (<<) の間のテキストはすべて、インライン・ファイルに収められ、ファイル名は *filename* となります。後で *filename* を使用してインライン・ファイルを参照することができます。*filename* が指定されない場合、`TMP` 環境変数が定義されていれば、そこで指定されたディレクトリーで、`NMAKE` がそのファイルに一意の名前を付けます。`TMP` 環境変数が定義されていなければ、`NMAKE` は現行ディレクトリー内で一意のファイル名を作成します。

インライン・ファイルは、一時的にも永続的にもなります。特に指定をしなかった場合、またはキーワード `NOKEEP` を指定した場合、インライン・ファイルは一時的なものになります。ファイルを保持するには `KEEP` を指定してください。

アットマーク (@) は `NMAKE` 構文には含まれませんが、ファイルが応答ファイルであることを示すために、一般的にユーティリティーが使用する文字です。

インライン・ファイルの例

```
math.lib : add.obj sub.obj mul.obj div.obj
    ilib @<<
math.lib
add.obj sub.obj mul.obj div.obj
/L:listing
<<
```

上記の例では、インライン・ファイルを作成し、それを使用してライブラリー・マネージャー (`ILIB`) を呼び出しています。インライン・ファイルは応答ファイルとして `ILIB` によって使用されます。使用するライブラリー、実行するコマンド、および生成するリスト・ファイルを指定します。このインライン・ファイルの内容は、次のようになります。

```
math.lib
add.obj sub.obj mul.obj div.obj
/L:listing
```

ILIB コマンドの後にファイル名がリストされていないため、インライン・ファイルには一意の名前が付けられ、現行ディレクトリー（または TMP 環境変数によって定義されたディレクトリー）内に配置されます。

エスケープ文字

NMAKE は、構文の中で以下の句読文字を使用します。

()	#	\$	^	\
{	}	!	@	-	

これらの文字のいずれかをコマンドで使用しても、NMAKE によって解釈されないようにするためには、その文字の前に脱字記号 (^) を使用します。

次に例を示します。

```
BIG^#.PLI
```

これは、次のように扱われます。

```
BIG#.PLI
```

脱字記号を使用すると、記述ファイル内にリテラル改行文字を記述することができます。この機能は、次の例のようなマクロ定義で役立ちます。

```
XYZ=abc^<ENTER>
def
```

これは、XYZ マクロに C スタイルのストリング abc¥ndef を割り当てたときの効果と同じ効果が得られます。この効果は、行を継続するために円記号 (¥) を使用したときの効果とは異なるため、注意してください。円記号の次に記述された改行文字は、スペースに置き換えられます。

NMAKE は、脱字記号の後に、構文で使用する文字が記述されていない場合、脱字記号を無視します。引用符の中で使用されている脱字記号は、エスケープ文字として扱われません。

エスケープ文字は、従属ブロックのコマンド部分では使用できません。

コマンドを変更する文字

以下の 3 つの文字のいずれかをコマンドの前に付加して、コマンド実行の方法を変更することができます。

- (ダッシュ)

コマンドのエラー・チェックをオフにします。

@ (アットマーク)

コマンドの表示を抑制します。

! (感嘆符)

各従属ファイルごとにコマンドを実行します。

コマンドを変更する文字

変更のための文字とコマンドの間にスペースを挿入してもかまいません。独立した行にあるコマンドは (変更されているかどうかにかかわらず)、1 つ以上のスペースまたはタブによってインデントする必要があります。

1 つのコマンドを変更するため複数の文字を使用することもできます。

エラー・チェックのオフへの切り替え (-)

構文: `-[n] command`

/I オプションは、コマンドのエラー・チェックをグローバルにオフにします。ダッシュ (-) コマンド修飾子は、グローバル設定をオーバーライドして、コマンドのエラー・チェックを個別にオフにします。この修飾子は、以下の 2 通りの方法で使われます。

- ダッシュの後に数字を指定しない場合、すべてのエラー・チェックをオフにします。
- ダッシュの後に数字を指定すると、NMAKE は、コマンドから戻された終了コードがその数よりも大きい場合のみ処理を中止します。

320 ページの『終了コードの無視 (/I)』を参照。

ダッシュ・コマンド修飾子の例

```
light.lst : light.txt
- flash light.txt
```

この例では、flash によって戻される終了コードがどのような値であっても、NMAKE が終了することはありません。

```
light.lst : light.txt
-1 flash light.txt
```

この例では、flash によって戻される終了コードが 1 よりも大きい場合は、NMAKE が終了します。

コマンド表示の抑制 (@)

構文: `@ command`

/S オプションは、NMAKE 実行時のコマンドの表示をグローバルに抑制します。アットマーク (@) 修飾子は、個別のコマンドの表示を抑制します。

/S オプションを指定するか、または @ 修飾子を使用した場合も、コマンドが生成する出力自体は、常に表示されます。

321 ページの『コマンド表示の抑制 (/S)』を参照。

アットマーク (@) コマンド修飾子の例

コマンド表示の抑制 (@)

```
sort.exe:sort.obj
@ echo sorting
```

echo コマンドを呼び出すコマンド行は表示されません。ただし、echo コマンドの出力は表示されます。

従属ファイルに対するコマンド実行 (!)

構文: ! command

感嘆符コマンド修飾子を付加すると、コマンドが特殊マクロ \$? または \$** のいずれかを使用している場合、各従属ファイルごとにコマンドが実行されます。\$? マクロは、ターゲットと比較して日付の古い従属ファイルすべてを参照します。\$** マクロは、記述ブロック内のすべての従属ファイルを参照します。

326 ページの『特殊マクロ』を参照。

感嘆符 (!) コマンド修飾子の例

```
leap.txt : hop.asm skip.c jump.pli
! print $** lpt1:
```

この例では、従属ファイルの変更日にかかわらず、以下の 3 つのコマンドを実行します。

```
print hop.asm lpt1:
print skip.c lpt1:
print jump.pli lpt1:
```

```
leap.txt : hop.asm skip.c jump.pli
! print $? lpt1:
```

この例では、leap.txt ファイルよりも変更日が新しい従属ファイルのみに対して print コマンドを実行します。hop.asm と jump.pli の変更日が leap.txt よりも新しい場合は、以下の 2 つのコマンドが実行されます。

```
print hop.asm lpt1:
print jump.pli lpt1:
```

EXTMAKE 構文

記述ファイルは特殊な構文を使用して、記述ブロック内の最初の従属ファイルのドライブ、パス、ベース名、および拡張子を判別できます。この構文は、*extmake* 構文と呼ばれます。

%s という文字は、最初の従属ファイルの完全なファイル指定を表します。ファイル指定のさまざまな部分が、次の構文を用いて示されます。

%<parts>

<parts>

以下の文字の組み合わせです。

d	ドライブ
p	パス
f	ベース名
e	拡張子

例えば、記述ブロック内の最初の従属ファイルのドライブとパス名を指定するには、次のように記述します。

%<dp>

コマンドを変更する文字

パーセント記号 (%) は、DOS および Windows コマンド行での置き換えを表します。コマンド行引数でパーセント記号を使用するには、パーセントを 2 重に使用します (%%)。

TOOLS.INI のマクロと推論規則

TOOLS.INI ファイル内に、マクロまたは推論規則を記述することができます。NMAKE は、最初に現行ディレクトリーで TOOLS.INI ファイルを探し、次に INIT 環境変数によって指示されるディレクトリーで探します。

NMAKE は TOOLS.INI ファイルを見付けると、次のタグを探します。

```
[nmake]
```

記述ファイルで使用する書式と同じ書式で、このタグの下にマクロと推論規則を記述することができます。

TOOLS.INI ファイルと記述ファイルの両方でマクロまたは推論規則が定義されている場合は、記述ファイルの定義が優先されます。また、/R オプションを使用した場合、TOOLS.INI ファイルは無視されます。

TOOLS.INI の例

```
[nmake]
.SUFFIXES: .pli
COMPILE_OPTS = gonumber source
.pli.obj:
    PLI $*.pli ($(COMPILE_OPTS)
```

TOOLS.INI ファイル内のこれらの行は、次の処理を実行します。

- 推論規則を適用できる拡張子のリストに、.pli ファイル拡張子を追加します。
- COMPILE_OPTS マクロを gonumber source として定義します。
- .pli ソース・ファイルから .obj ファイルをビルドする推論規則を定義します。

第 20 章 パフォーマンスの向上

最適なパフォーマンスのためのコンパイル時オプション

オプションの選択	341
OPTIMIZE	342
IMPRECISE	342
GONUMBER	342
SNAP	342
RULES	343
PREFIX	343
CONVERSION	344
FIXEDOVERFLOW	344
DEFAULT	344
BYADDR または BYVALUE	345
(NON)CONNECTED	346
RETURNS(BYVALUE) または RETURNS(BYADDR)	346
(NO)DESCRIPTOR	346
(RE)ORDER	346
LINKAGE	347
ASCII または EBCDIC	347

IEEE または HEXADEC	347
(NON)NATIVE	347
(NO)INLINE	347
パフォーマンスを向上させるコンパイル時オプションの要約	348
パフォーマンス向上のためのコーディング	348
DATA 指示入出力	349
入力専用パラメーター	349
ストリングの割り当て	350
ループ制御変数	350
PACKAGE 対ネストされた PROCEDURE	351
ネストされたプロシーチャーの例	351
REDUCIBLE 関数	352
DEFINED 対 UNION	352
名前付き定数対静的変数	352
意味のある名前が付けられていない最適なコードの例	353
ライブラリー・ルーチンの呼び出しの回避	354

ユーザーのプログラムの速度を向上することに関する考慮事項の多くは、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しません。ただし、本章では、考慮事項の中でもワークステーション PL/I コンパイラーとそれが生成するコードに特有な考慮事項を取り上げて説明します。

最適なパフォーマンスのためのコンパイル時オプションの選択

選択するコンパイル時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、コンパイル時オプションはコマンド行または環境変数 `IBM.OPTIONS` 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 (デフォルト以外の) コンパイル時オプションを指定する方法があります。

```
PREFIX(NOFLO)  
IMPRECISE  
OPT(2)  
DFT(REORDER)
```

最初の 2 つのオプションは、プログラムのセマンティクスに影響する場合がありますが、通常影響が出るのは異常な状況においてのみです。最初の 2 つのオプションを指定すると、最適化をオフにしてコンパイルした場合でもコードが改善されます。これらのオプションを使用することによって、コンパイラーがエラーを発生させる可能性も減少します。

次のセクションでは、パフォーマンスの向上と、特定のコンパイル時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含まれている 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。この場合に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。

IMPRECISE

このオプションを選択すると、コンパイラーは、浮動小数点演算用にサイズが小さく高速な命令シーケンスを生成します。これにより、浮動小数点の式 (個別でもループでも) を含むプログラムのパフォーマンスに対して大きな効果が見られる場合があります。

ただし、IMPRECISE オプションを使用してプログラムがコンパイルされると、浮動小数点例外が、発生した正確な場所で報告されない場合があります。(これは、特に OPTIMIZE オプションが有効な場合に当てはまります。) さらに、浮動小数点演算において、正確に IEEE に準拠しない結果が生成される可能性もあります。

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。

開発中にデバッグを支援するため、リンカー・オプションのいずれかを使用して、実行可能ファイルにステートメント番号表を組み込むことができます。/DEBUG (/DE) オプションは、これらの表を実行可能ファイルに組み込むようにリンカーに指示するため、ILINK コマンドで /DE を指定しないことにより、実行可能ファイルのサイズを制御しやすくなります。実行可能ファイルのサイズが考慮事項となっている場合、実動モードでは表を除外しておくことができます。

SNAP

SNAP オプションを使用すると、コンパイラーは、すべてのブロックのプロローグ・コードおよびエピローグ・コードで追加の命令を生成します。これらの命令によって、実行時のトレースバック・メッセージ (PLIDUMP と、ON ステートメント上の SNAP オプションによって生成される) 内に、トレースバックが要求されたときにアクティブであったすべてのプロシージャラーが含まれるようにします。

SNAP オプションの使用およびこれらの追加命令の作成のトレードオフは、アプリケーションのパフォーマンスに悪影響を及ぼす可能性がある点です。このことは特に、頻繁に呼び出されるプロシージャに当てはまります。

RULES

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。

```
    dcl (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)** をもちます。つまり、割り算の前にシフトは必要なく、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。

RULES(LAXCTL) オプションを使用すると、次に示すように、固定エクステントを用いて **CONTROLLED** 変数を宣言してから、異なるエクステントを用いてそれを **ALLOCATE** することを、コンパイラーが許可します。

```
    DECLARE X BIT(1) CTL;

    ALLOCATE X BIT(63);
```

ただし、このプログラミング方法では、コンパイラーは、固定エクステントを持つ **CONTROLLED** 変数が存在しないことを前提としなければなりません。したがって、これらの変数が参照されている場合は、生成されるコードの効率が大幅に低下します。

しかし、**CONTROLLED** 変数用の固定エクステントを、常にその長さ (範囲) が変わらない場合にのみ指定するのであれば、オプション **RULES(NOLAXCTL)** を指定すると、パフォーマンスが大幅に向上します。

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。ただし、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特定するのに役立ちます。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を行わずに実行されます。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す。

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。しかし、パーソナル・コンピュータではハードウェアがこの条件を発生させないため、コンパイラーは追加のコードを生成する必要があります。この追加のコードは、ユーザーのプログラムに悪影響を与える可能性があります。プログラムでこの条件の発生を必要としない（または期待しない）場合は、パフォーマンスを向上させるために PREFIX(NOFIXEDOVERFLOW) を指定します。

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。デフォルトのサブオプションは次のとおりです。

IBM
BYADDR
RETURNS(BYVALUE)
NONCONNECTED
DESCRIPTOR
ORDER
ASSIGNABLE LINKAGE(OPTLINK)

ASCII
IEEE
NATIVE
NODIRECTED
NOINLINE

IBM/ANS、ASSIGNABLE/NONASSIGNABLE、および DIRECTED/NODIRECTED サブオプションは、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。

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が変更されると、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 つのパターンで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、コンパイラーは、その変数が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 */ );

    decl parm1 byvalue ...;
    decl parm2 byvalue ...;
    .
    .
    decl 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。

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。

(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。ただし、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のみ、使用されます。

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 宣言で必要なときにだけ 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。

(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降下するブロック) 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。

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（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。

ASCII または EBCDIC

`DEFAULT(ASCII)` オプションは、デフォルトでは、文字データがネイティブの `Intel` スタイルで保持されることを示します。`EBCDIC` サブオプションを指定すると、コンパイラーは、文字変数の入出力を伴うほとんどの操作用に、追加の命令を生成する必要があります。

IEEE または HEXADEC

`DEFAULT(IEEE)` オプションは、デフォルトでは、浮動小数点データがネイティブの `Intel` スタイルで保持されることを示します。`HEXADEC` サブオプションを指定すると、浮動小数点変数が関わるほとんどの操作用に、コンパイラーが実行しなければならない命令が大幅に増えます。

(NON)NATIVE

`DEFAULT(NATIVE)` オプションは、デフォルトでは、固定バイナリー・データ、オフセット・データ、序数データ、および可変ストリングの長さ接頭部がネイティブの `Intel` スタイルで保持されることを示します。`NONNATIVE` を指定すると、これまでにリストされたデータ型が関わる操作用に、追加の命令が生成されます。

(NO)INLINE

サブオプション `NOINLINE` は、プロシージ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など、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ジ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。

- プロシージ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。
- プロシージ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ジャーだけで `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。

パフォーマンスを向上させるコンパイル時オプションの要約

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(2)
IMPRECISE
NOSNAP
PREFIX(NOFIXEDOVERFLOW)
RULES(ANS NOLAXCTL)
次のサブオプションが指定された DEFAULT
(BYVALUE
RETURNS(BYVALUE)
CONNECTED
NODESCRIPTOR
REORDER
ASCII
IEEE
NATIVE
LINKAGE(OPTLINK)
```

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、読みやすさや保守容易性などによって、選択するコーディング・スタイルは変わってきます。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 指示入出力

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

プロシーチャーに、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。プロシーチャー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）。

たとえば、次の宣言では、

dosScanEnv が入力専用の CHAR VARYINGZ ストリングです。

```
dc1 dosScanEnv entry( char(*) varyingz nonasgn byaddr,
                    pointer byaddr )
returns( native fixed bin(31) optional )
options( nodestructor linkage(system) );
```

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、コンパイラーが生成した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。

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もっている。
- 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もてない場合、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もつ 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。

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ジャの例

```
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ジャの例:

```
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 つまたは複数) が変更されない限り、プロシージャーまたは入り口を複数回呼び出す必要がないこと、およびプロシージ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 1 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;
```

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。

```

dcl b1 bit(31);
dcl b2 bit(16) def b1;

dcl
  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、それを認識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。ただし、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、読みやす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 組み込み関数を 2 通りの方法で使って、同一のオブジェクト・コードが生成されています。

```

dcl numeric char value('0123456789');

jx = verify( string, numeric );

jx = verify( string, '0123456789' );

```

次の例は、VALUE 属性を使って、読みやす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

```

dcl 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:

```

dcl ( 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;

dcl 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
- 両方のオペランドが位置合わせ bit(8n) である (n は定数)

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。インラインで生成されたコードを用いて実行される変換を 355 ページの表 27 に示します。

表 27. 変換がインラインで処理される条件

ターゲット	ソース	条件
fixed bin(p1,q1)	fixed bin(p2,q2)	常時
	float(p2)	SIZE が無効の場合
	bit(1)	常時
	bit(n) aligned	n が既知であり
	char(1)	$n \leq 31$ の場合
	pic'(n)9'	CONV が無効の場合 $n \leq 6$ の場合
	pic'(n)Z(m)9'	
		$n + m \leq 6$ の場合
fixed dec(p1,q1)	fixed dec(p2,q2)	特に高速なライブラリー・ルーチンを使用して実行される
float(p1)	fixed bin(p2,q2)	常時
	float(p2)	常時
	bit(1)	常時
	bit(n) aligned	n が既知であり
	char(1)	$n \leq 31$ の場合
	pic'(n)9'	CONV が無効の場合 $n \leq 6$ の場合
	pic'(n)Z(m)9'	
		$n + m \leq 6$ の場合
pictured fixed	pictured fixed	ピクチャーが一致する場合
pictured float	pictured float	ピクチャーが一致する場合
char	char nonvarying	常時
	char varying	常時
	char varyingz	常時
	pictured fixed	常時
	pictured float	常時
	pictured char	常時
pictured char	pictured char	ピクチャーが一致する場合
bit(1) nonvarying	bit(1) nonvarying	常時
bit(n) nonvarying	bit(m) nonvarying	注参照

注: 以下のすべてが適用される場合

- 1) ソースとターゲットはバイト整合
- 2) n と m は既知
- 3) $\text{mod}(m,8)=0$ または $n=m$ またはソースが定数
- 4) $\text{mod}(n,8)=0$ またはターゲットが STATIC 属性、AUTOMATIC 属性、または CONTROLLED 属性を持つスカラー

多くのストリング処理組み込み関数は、ライブラリー・ルーチンへの呼び出しを通じて評価されますが、一部はライブラリーを呼び出すことなく処理されます。356 ページの表 28 では、これらの組み込み関数と、それらがインラインで処理される条件をリストします。

表 28. スtring組み込み関数がインラインで処理される条件

String関数	コメントおよび条件
BOOL	3 番目の引数が定数の場合。また、最初の 2 つの引数は、どちらも bit(1) であるか、またはどちらも aligned bit(n) でなければなりません (n は 8、16、または 32)。関数はまた、ビット単位の 2 項演算に削減することができ、かつ両方の引数 aligned bit である場合には、インラインで処理されます。
COPY	最初の引数の型が character の場合。
EDIT	最初の引数が REAL FIXED BIN の場合、SIZE 条件は無効であり、2 番目の引数はすべて 9 からなる定数Stringです。
HIGH	常時
INDEX	渡される引数が 2 つのみで、それらの型が character の場合。
LENGTH	常時
LOW	常時
MAXLENGTH	常時
SEARCH	渡される引数が 2 つのみで、それらの型が character の場合。
SEARCHR	渡される引数が 2 つのみで、それらの型が character の場合。
SUBSTR	STRINGRANGE が無効の場合。
TRANSLATE	2 番目と 3 番目の引数が定数の場合。
TRIM	渡される引数が 1 つのみで、その型が character の場合。
UNSPEC	常時
VERIFY	渡される引数が 2 つのみで、それらの型が character の場合。
VERIFYR	渡される引数が 2 つのみで、それらの型が character の場合。

第 21 章 ユーザー出口の用法

コンパイラー・ユーザー出口の使用	357	初期化プロシーチャーの作成	361
コンパイラー・ユーザー出口によって実行される プロシーチャー	358	メッセージ・フィルター操作プロシーチャー の作成	362
コンパイラー・ユーザー出口の活動化	358	終了プロシーチャーの作成	364
IBM 提供のコンパイラー出口、IBMUEXIT	359	CICS ランタイム・ユーザー出口の使用	364
コンパイラー・ユーザー出口のカスタマイズ	359	プログラム起動前の動作	364
IBMUEXIT.INF の変更	359	プログラム終了後の操作	365
独自のコンパイラー出口の作成	360	CEEFXITA の変更	365
グローバル制御ブロックの構造	360	データ変換表の使用	365

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。ワークステーション PL/I および PL/I for AIX 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。

提供されるファイルのタイプには、以下のようなものがあります。

- `..%samples` に入っている、拡張子 `PLI` の PL/I ソース・ファイル。
- `..%include` に入っている、拡張子 `CPY` の PL/I インクルード・ファイル。ユーザー出口のコンパイル時には、`CPY` ファイルを検出できるように、`INCLUDE` または `IBM.SYSLIB` 環境変数を必ず設定してください。
- `..%samples` に入っている、拡張子 `DEF` のリンカー定義ファイル。
- `..%samples` に入っている、拡張子 `INF` の制御ファイル (出口に適用可能な場合)。ユーザー出口を使用するときは、`INF` ファイルを含むディレクトリーが、適切な環境変数 (通常 `DPATH`) を用いて指定されていることを確認してください。

コンパイラー・ユーザー出口の使用

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ますし、製品付属の出口を使うこともできます。製品付属の出口は、必要に応じて「そのまま」使用しても、少し変更してもかまいません。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チャー
- 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

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 27 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、メッセージ・フィルター・プロシージャ、および終了プロシージャへ制御を渡します。これらの 3 つのプロシージャは、それぞれ、要求されたプロシージャが完了したならば、制御をコンパイラーへ戻します。

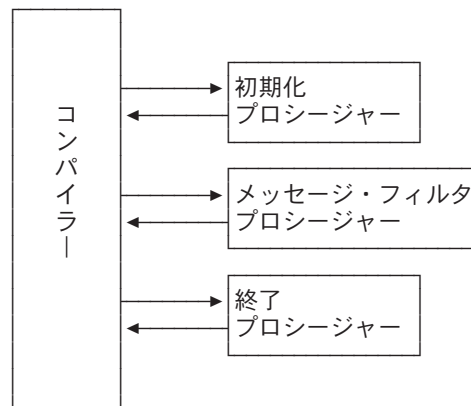


図 27. PL/I コンパイラー・ユーザー出口のプロシージャ

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、360 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、361 ページの『初期化プロシージャの作成』、362 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 364 ページの『終了プロシージャの作成』を参照してください。

コンパイラー・ユーザー出口の活動化

コンパイラー・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、55 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用すると、メッセージ制御ファイルを指定するユーザー・オプション・ストリングを指定できます。ストリングを指定しない場合は、IBMUEXIT.INF が使用されます (359 ページの『IBMUEXIT.INF の変更』を参照) が、そのファイルを探す場所をコンピューターに指示する必要があります。IBMUEXIT.PLI サンプル・プログラムを変更しない場合、デフォルトの動作では、コンパイラーは最初に現行ディレクトリー、次に DPATH で指定されたディレクトリーでIBMUEXIT.INF を探します。

ユーザー・オプション・ストリングは、360 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、360 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 提供のコンパイラー出口、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ー・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。

IBMUEXIT は、以下のいくつかのファイルで構成されています。

IBMUEXIT.PLI

PL/I ソース・コードを含む。

IBMUEXIT.DLL

IBMUEXIT.PLI の実行可能 DLL。このファイルをビルドするには、コマンド行から次のコマンドを発行します。

Windows の場合:

```
pli ibmuexit
ilib /geni ibmuexit.def
ilink /dll ibmuexit.obj ibmuexit.exp
```

IBMUEXIT.DEF

IBMUEXIT.DLL をビルドするために使用される DEF ファイル。

IBMUEXIT.INF

メッセージのフィルター操作を指定する制御ファイル。

PLI ソース・ファイルが提供されており、参考にしたたり、変更して使用することができます。INF 制御ファイルは、モニターすべきメッセージ番号を含んでおり、それらに対して実行する処置を IBMUEXIT に指示します。実行可能モジュールは INF 制御ファイルを読み取り、メッセージを無視するか、またはその重大度を変更します。

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 IBMUEXIT.PLI を変更することもできます。いずれの場合も、コンパイラー・ユーザー出口用の実行可能ファイルの名前は IBMUEXIT.DLL 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 IBMUEXIT.INF 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

IBMUEXIT.INF の変更

まったく新しいコンパイラー・ユーザー出口の作成に時間を費やすのではなく、サンプル・プログラム IBMUEXIT.INF を変更して使用することができます。

INF 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。サンプル IBMUEXIT.INF ファイルを図 28 に示します。

Fac Id	Msg No	Severity	Suppress	Comment
'IBM'	1041	-1	1	Comment spans multiple lines
'IBM'	1044	-1	1	FIXED BIN 7 mapped to 1 byte
'IBM'	1172	0	0	Select without OTHERWISE
'IBM'	1052	-1	1	Nodescriptor with * extent args
'IBM'	1047	12	0	Reorder inhibits optimization
'IBM'	8009	-1	1	Semicolon in string constant
'IBM'	1107	12	0	Undeclared ENTRY
'IBM'	1169	0	1	Precision of result determined by arg

図 28. IBMUEXIT.INF 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。

- 最初の列に単一引用符で囲んだ文字 'IBM' を指定する必要があります。これはメッセージ接頭語です。
- 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 (サンプル・プログラムの 1 つとして製品に付属) を使用できます。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。

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、360 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ーを使うと、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) または「**WARNING**」(重大度コード 4) メッセージの重大度のみです。

プロシージャー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。

プロシージャー特有のメッセージ・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_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。長さは 1 から 15 文字。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。長さは 1 から 15 文字。この値はユーザーによって設定されます。
- **Uex_MFX_Inserts:** メッセージの挿入の回数。範囲は 0 から 6。この値はコンパイラーによって設定されます。
- **Uex_MFX_Inserts_Data:** それぞれの挿入について説明するためのフィールド。この値はコンパイラーによって設定されます。
- **Uex_MFX_Ins_Type:** 挿入の型。以下の挿入の型が使用できます。
 - **Uex_Ins_Type_Xb31:** 整数型に使用。値は 1 です。
 - **Uex_Ins_Type_Char:** 整数型に使用。値は 2 です。
 - **Uex_Ins_Type_Series:** 整数型に使用。値は 3 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_Bin:** 整数型を持つ挿入の整数値。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:** 文字型を持つ挿入の長さ (バイト)。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:** 文字型を持つ挿入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:** 連続型を持つ挿入の各エレメント間で挿入される文字。通常、これはブランク、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:** 連続型を持つ挿入の可変文字ストリングの連続のアドレス。このアドレスは、連結するストリングの数を保持する **FIXED BIN(31)** フィールドを指し示します。このストリングの数の後にストリングのアドレスが続きます。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チャーの完了時に、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、360 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

CICS ランタイム・ユーザー出口の使用

CICS ランタイム出口の CEEFXITA の主要な機能の 1 つとして、PL/I トランザクションが失敗したときに CICS Dynamic Transaction Backout (DTB) が発生するかどうかを制御することができます。CICS ランタイム出口は、CICS の下でのトランザクションの内部で各 PL/I プログラムを起動する直前および直後に駆動されます。出口が呼び出されるたびに、型指定された構造体 CXIT (インクルード・ファイル IBMVCXT.INC に含まれる) が PL/I ランタイムと出口の間の通信に使用されます。

ユーザー出口を見直し、(必要であれば) 修正することを強くお勧めします。

この構造体には、PL/I プログラムに関連する以下のような情報が含まれています。

- 出口を呼び出す理由 (初期化または終了)
- プログラム終了後に呼び出された場合にプログラムがどのように終了したかを表す理由コード
- キー CICS 制御ブロックへのポインター

プログラム起動前の動作

PL/I プログラムの起動前に出口が呼び出された場合、出口はプログラム起動をバイパスするように PL/I ランタイムに指示することができます。この場合、必要に応じて DTB が発生します。

このように出口を呼び出している間、ほかの関数（ランタイム・オプションの問い合わせまたは設定など）は実行できません。

IBM 提供の出口は、PL/I プログラム起動の進行を許可するだけで戻ります。

プログラム終了後の操作

PL/I プログラム起動後に出口が呼び出された場合、出口はプログラム終了の理由を調べ、DTB を要求することができます。終了理由コードは、プログラムが終了した理由を表します。ファイル IBMVCXT.INC に、詳細な情報が含まれています。

この場合、IBM 提供の出口は、次のような場合に DTB を要求します。

- PL/I プログラム戻りコード (PLIRETC で設定) がゼロ以外の場合
- 終了の理由が正常終了以外の場合

CEEFXITA の変更

以下のソース・ファイルが提供されています。

CEEFXITA.PLI

PL/I ソース・コード。

出口を再コンパイルするには、INCDIR コンパイル時オプションを設定して IBMVCXT.INC のディレクトリを組み込みます。コマンド行で次のコマンドを入力してください。

```
pli CEEFXITA
```

IBMVCXT.INC

CXIT 型指定構造体およびその他のインターフェース情報。

CEEFXITA.DLL

実行可能 DLL。

この DLL を再ビルドするには、コマンド行から次のコマンドを発行します。

```
ilink /dll ceefxita.obj ceefxita.def
```

CEEFXITA.DEF

CEEFXITA.DLL をビルドするために使用される DEF ファイル。

データ変換表の使用

コンパイラー、プリプロセッサ、ライブラリー、およびデバッガーが ASCII から EBCDIC またはその逆の変換のために使用するルーチンは、DLL ファイルにあります。

Windows の場合、ルーチンは以下の 2 つのファイルにあります。

- ibmwstb.dll (マルチスレッド化を行わない)
- ibmwmtb.dll (マルチスレッド化を行う)

これらのルーチンのソースは、使用する表を含め、製品に付属しており、必要に応じて異なる表を使用することができます。ファイルが EBCDIC から ASCII に変換される場合、IBM 提供の表とは異なる表を使用してダウンロードするため、表を置き換える必要があります。

データ変換表の使用

変換ルーチンの名前は IBMPBE2A (EBCDIC から ASCII) および IBMPBA2E (ASCII から EBCDIC) です。製品に付属するファイルの名前は変更しないでください。

定義ファイルも製品に付属します。

Windows の場合、以下の定義ファイルがあります。

- ibmwstb.def
- ibmwmtb.def

対応する DLL を作成するときに、これらの定義ファイルを使用する必要があります。

第 22 章 ダイナミック・リンク・ライブラリーの構築

DLL ソース・ファイルの作成	367	DLL の使用	369
DLL ソースのコンパイル	368	DLL を構築するサンプル・プログラム	369
DLL のリンク準備	368	メインプログラムでの FETCH および RELEASE	
Windows におけるエクスポートする名前の指定	368	の使用	371
DLL のリンク	368	DLL からのデータのエクスポート	371

ダイナミック・リンクは、ダイナミック・リンク・ライブラリー (DLL) を使用して外部参照を解決するプロセスです。ダイナミック・リンクの利点には、以下のものがあります。

- メモリー所要量の減少
- アプリケーション変更の単純化
- 柔軟なソフトウェア・サポート
- 関数の透過的な移行
- 複数プログラム言語のサポート
- アプリケーション制御によるメモリーの使用

DLL は、通常、多くのアプリケーションが使用できる共通の関数を提供するために使用されます。DLL を使用するアプリケーションは、ロード時ダイナミック・リンクまたはランタイム・ダイナミック・リンクのいずれかを使用できます。

提供されたランタイム DLL およびユーザー独自の DLL に、動的にリンクできます。本章では、ダイナミック・リンク・ライブラリーを作成し、使用するための以下のステップについて説明します。

- DLL のソース・ファイルの作成
- DLL のモジュール定義ファイル (.DEF) の作成
- ソース・ファイルのコンパイルおよび結果オブジェクト・ファイルのリンクによる、DLL ファイルの構築
- 外部モジュールをリンクするときに使用する、DLL の内容を示すモジュール定義ファイルの作成

各セクションでは、コンパイラと一緒にパッケージされている、サンプル・プログラム SORT.PLI の該当するサンプルを記載します。

DLL ソース・ファイルの作成

DLL を構築するには、まず、DLL に含めたいデータまたはルーチンが入ったソース・ファイルを作成する必要があります。DLL ソース・ファイルには、特別なファイル拡張子は必要ありません。

DLL からエクスポートしたい各ルーチン (つまり、他の実行可能モジュールまたは DLL から呼び出すルーチン) は、デフォルトで、または外部キーワードによって修飾されることで、外部ルーチンでなければなりません。

DLL ソースのコンパイル

ソース・ファイルは、その他のファイルをコンパイルするのと同じ方法 (PLI コマンドを使用) でコンパイルして、DLL を作成できます。ただし、少なくとも 1 つのファイルを DLLINIT オプションを指定してコンパイルしなければならないという例外があります。DLLINIT オプションを指定して DLL 内のすべてのルーチンをコンパイルできます。ただし、DLLINIT を指定してコンパイルしたルーチンを EXE にリンクすることはできません。

オプション XINFO(DEF) を指定してプログラムをコンパイルしたい場合もあります。このオプションは、各プログラムの .DEF ファイルを作成します。これらの .DEF ファイルは、DLL のリンクを準備するときに必要不可欠なものです。

DLL のリンク準備

DLL をリンクする場合は、DLL からエクスポートするものの名前をリンカーに知らせる必要があります。

Windows におけるエクスポートする名前の指定

Windows では、.EXP ファイルを使用して、どの部分をエクスポートするかをリンカーに知らせます。.EXP ファイルはバイナリー・ファイルであり、/GENI オプションを指定して ilib を呼び出し、以下のいずれかを入力として使用して作成します。

- DLL の .DEF ファイル
- DLL によってエクスポートされる名前を含むすべての .OBJ

.DEF ファイルを使用すると、DLL によってエクスポートされるものを正確に制御できるため、このファイルの使用をお勧めします。.OBJ 名を指定すると、指定したオブジェクト・ファイル内のすべての外部名がエクスポートされます。

.EXP ファイルを作成するときに使用できるコマンド例を以下に示します。

```
ilib /geni myliba.def
```

作成される Windows .DEF ファイルは、以下の特性を持ちます。

- Windows バージョンには EXPORTS ステートメントのみが含まれます。
- Windows バージョンには「装飾された」名前が含まれます。

「装飾」という名前は、ルーチンのリンケージによって異なります。コンパイラによって作成された .DEF ファイルを使用する場合は、これを気にする必要はありません。

DLL のリンク

DLL をリンクするには、以下のオプションおよび入力ファイルを使用します。

リンカー・オプション

- /dll、
- /out: 後に DLL の名前を続ける

入力ファイル

- DLL を構成するすべての OBJ

- エクスポートするものを指定する .DEF または .EXP ファイル

例えば、mydll1a.obj および mydll1b.obj を mydll1a.dll にリンクするには、Windows において次のリンク・コマンドを実行します。

```
ilink /dll /out:mydll1a.dll mydll1a.obj mydll1b.obj mydll1a.exp
```

DLL の使用

DLL が構築されると、アプリケーション内のその他のルーチンは、以下のいずれかの方法を使用して、DLL によってエクスポートされる変数およびルーチンにアクセスできます。

- FETCH ステートメント
- インポート・ライブラリーとのリンク

アプリケーションが FETCH ステートメントを使用して DLL のエレメントにアクセスする場合、リンクするときに特別な処置はありません。アプリケーションがその FETCH ステートメントを実行しない場合、DLL は存在する必要もありません。

DLL に静的にリンクしているかのように、アプリケーションが DLL のエレメントにアクセスする場合、リンカーはそのエレメントの名前を解決できる必要があります。

Windows においては、ユーザーが DLL のインポート・ライブラリーにリンクしていれば、リンカーは DLL 内の名前を解決できます。実際に、PL/I ライブラリー・ルーチンの名前はそうように解決されます。例えば、ibmws20i.lib にリンクする場合は、ibmws20.dll のインポート・ライブラリーにリンクします。

Windows においては、DLL のインポート・ライブラリーは、DLL のリンク準備で .EXP ファイルを作成するときに構築されます。

注: ローダーが DLL を検出するには、DLL は、現行作業ディレクトリー内か、Windows の PATH 環境変数にリストされたいずれかのディレクトリー内に存在する必要があります。

DLL を構築するサンプル・プログラム

サンプル・プログラムの SORT.PLI および DRIVER1.PLI は、3 つの異なるソート関数を含む DLL を構築し、使用方法を示しています。これらの関数は、スワップ数を追跡し、ソートに必要な操作を比較します。

サンプル・プログラムのファイルは以下のとおりです。

SORT.PLI

DLL のソース・ファイル

SORT.DEF

DLL のモジュール定義ファイル

DRIVER1.DEF

実行可能モジュールのモジュール定義ファイル

DLL を構築するサンプル・プログラム

EXTDCL.CPY

ユーザーインクルード・ファイル

DRIVER1.PLI

SORT.DLL を使用するメインプログラム

サンプル・プログラムをインストールすると、これらのファイルは、..`¥SAMPLES¥`ディレクトリーに置かれます。

以下の順番にコマンドを実行し、プログラムをコンパイル、リンク、および実行します。

1. `pli sort`
2. `ilib /geni sort.def`
3. `ilink /dll /out:sort.dll sort.obj sort.exp`
4. `pli driver1`
5. `ilink driver1.obj /stack:80000 sort.lib`
6. `driver1`

メインプログラムでの FETCH および RELEASE の使用

SAMPLES ディレクトリーには DRIVER1.PLI の変更バージョンである DRIVER2.PLI も含まれています。このプログラムは、FETCH および RELEASE ステートメントを使用して、ロード時の代わりに実行時に SORT.DLL ルーチンを動的にリンクします。

このバージョンの DRIVER プログラムを使用する主な利点は、ソート・ルーチンをメモリーに入れたり、解放したりする時期を制御できることにあります。ただし、FETCH および RELEASE ステートメントを使用すると、プログラムの実行時間が長くなる可能性があります。

以下の順番にコマンドを実行し、Windows で、このバージョンの DRIVER プログラムをコンパイル、リンク、および実行します。

1. pli sort
2. ilib /geni sort.def
3. ilink /dll /out:sort.dll sort.obj sort.exp
4. pli driver2
5. ilink driver2.obj /stack:80000
6. driver2

DLL からのデータのエクスポート

ここまでは、DLL から外部エントリーをエクスポートする方法について説明しました。DLL から外部データをエクスポートすることもできます。DLL から外部データをエクスポートするには、そのデータが、アプリケーション全体を通して RESERVED として宣言されている必要があります。以下の条件も満たしている必要があります。

- 変数をエクスポートする DLL は、その DLL のパッケージの RESERVES オプションでその変数を指定する必要がある。
- 別の DLL から変数をインポートするすべての DLL および EXE も、その変数を RESERVED(IMPORTED) として宣言する必要がある。

例えば、変数 datatab のみをエクスポートする DLL を作成するには、次のルーチンを使用します。

```
*process dllinit langlv1(saa2);

  edata: package reserves( datatab );

      dcl datatab char(256) reserved external init( .... );
  end;
```

この DLL 外のプロシージャに datatab をインポートするには、次のように宣言します。

```
dcl datatab char(256) reserved(imported) external;
```

第 23 章 Windows における IBM Library Manager の使用

ILIB の実行	373	/REMOVE	381
コマンド行の使用	374	ILIB オプション	381
ILIB 環境変数の使用	375	ILIB オプションの要約	381
コマンド行	375	/?	382
Windows コントロールパネル	375	/BACKUP	382
Windows 98 AUTOEXEC.BAT ファイル	375	/DEF	383
ILIB 応答ファイルの使用	375	/FREEFORMAT	383
ILIB パラメーターの指定例	376	/GENDEF	383
ILIB 入力の制御	377	/GI	383
ILIB 出力の制御	377	/HELP	384
ILIB 出力の制御	378	/LIST	384
ILIB オブジェクト	379	/NOEXT	384
ILIB オブジェクトの要約	379	/OUT	385
Add/Replace	379	/QUIET	385
/EXTRACT	380	/WARN	385

オブジェクト・コード・ライブラリーの作成および管理、インポート・ライブラリーとエクスポート・オブジェクトのペアの作成、およびモジュール定義 (.def) ファイルの生成を行うには、IBM Library Manager (ILIB と呼ばれる) を使用します。ILIB ユーティリティを使用すると、以下のことができます。

- オブジェクト・コレクションからの新規ライブラリーの作成
- ライブラリーの管理
 - オブジェクトの既存ライブラリーへの追加
 - オブジェクトの既存ライブラリーからの削除
 - オブジェクトの既存ライブラリーからのコピー
 - オブジェクトの既存ライブラリーでの置換
- 新規または既存ライブラリーの内容のリスト
- 以下のものからの、インポート・ライブラリーとエクスポート・オブジェクトのペアの作成
 - モジュール定義 (.def) ファイル
 - #pragma export および _Export ステートメントを含むソース・ファイルから生成されたオブジェクト
 - 上記の組み合わせ
- 以下のものからの、モジュール定義 (.def) ファイルの生成
 - 既存 DLL
 - #pragma export および _Export ステートメントを含むソース・ファイルから生成されたオブジェクト
 - 上記の組み合わせ

ILIB の実行

コマンド・プロンプトで `ilib` を入力して ILIB を実行します。

以下の方法でパラメーターを指定できます。

1. 直接、コマンド行に入力します
2. ILIB 環境変数を使用します

3. 応答ファイルと呼ばれるテキスト・ファイルにそれらを記入し、ilib コマンドの後にファイル名を指定します。
4. 上記の組み合わせ

Ctrl+C または Ctrl+Break を押せば、ILIB の実行中にいつでもオペレーティング・システムに戻ることができます。完了前に ILIB を中断すると、バックアップからオリジナル・ライブラリーがリストアされます。

注:

1. ILIB は、開始されると、中断または間違いがあった場合に備えて、オリジナル・ライブラリーのバックアップ・コピーを作成します。オリジナル・ライブラリーおよび修正コピーの両方に、十分なディスク・スペースを確保してください。
2. ライブラリーの最後には .lib 拡張子を付ける必要があります。拡張子を指定しない場合は、デフォルトの拡張子 .lib が付加されます。ハイパフォーマンス・ファイル・システム (HPFS) ファイルがサポートされています。したがって、mylibraryname.new.lib も有効なライブラリーです。

コマンド行の使用

ILIB に必要なすべての入力をコマンド行で指定できます。コマンド行の構文は次のとおりです。

```
ilib [options] [libraries] [@responsefile] [objects]
```

オプション

ILIB の動作に影響するオプション

ライブラリー

作成または変更する入力ライブラリー

応答ファイル

ILIB オプションを含むテキスト・ファイルの名前

オブジェクト

ライブラリー内でのオブジェクト・モジュールの追加、削除、置換、コピー、および移動に使用するコマンド

ILIB コマンド行はフリー・フォーマット・コマンド行です。つまり、入力引数は、任意の回数、任意の順序で指定できます。/FREEFORMAT オプションのみは例外であり、位置の制限があります。詳しくは、383 ページの『/FREEFORMAT』を参照してください。

注: ILIB の OS/2 リリースとの互換性のために、固定フォーマット・コマンド行もサポートされています。固定フォーマット・コマンド行を使用するには、/NOFREEFORMAT オプションを、コマンド行で ilib の直後に指定するか、ILIB 環境変数の最初のパラメーターとして指定する必要があります。デフォルトのコマンド行フォーマットは、フリー・フォーマットです。

本書の目的に合わせて、フリー・フォーマット・コマンド行のみを詳しく説明します。

ILIB 環境変数の使用

ILIB 環境変数を使用して、デフォルトの ILIB オプションを指定できます。ilib コマンドを呼び出すと、コマンド行の前に環境変数が解析されます。

SET コマンドを使用して、ILIB 環境変数に値を指定します。以下の方法でこれを行うことができます。

コマンド行

コマンド行で SET コマンドを使用すると、指定した値はそのセッションでのみ有効になります。前に指定された値はオーバーライドされます。

%variable% を使用して、変数のオリジナル値を付加できます。次の例では、既存オプションの前に /NOFREEFORMAT オプションが指定され、ILIB 環境変数が ILIB 環境変数のオリジナル値に設定されます。

```
SET ILIB=/FREEFORMAT %ILIB%
```

Windows コントロールパネル

Windows の場合は、Windows コントロールパネルを使用して、環境変数を更新し、すぐにそれらを有効にできます (つまり、リブートは必要ありません)。

ILIB 環境変数を設定するには、以下のようになります。

- 「メイン」アイコンをダブルクリックし、「メイン」グループを選択します。
- 「メイン」グループで「システム」アイコンをダブルクリックし、選択します。
- 「変数」フィールドに ILIB を入力します。
- 「値」フィールドに ILIB 環境変数の値を入力します。
- 「設定」を選択します。

Windows 98 AUTOEXEC.BAT ファイル

Windows 98 の場合は、AUTOEXEC.BAT ファイルで環境変数を設定できます。この方式で設定した環境変数は、すべてのユーザー・セッションで有効です。

環境変数を希望する値に設定する行を、AUTOEXEC.BAT ファイルに追加します。次の例を見てください。

```
SET ILIB=/NOBACKUP
```

AUTOEXEC.BAT ファイルで指定した環境変数は、開始するすべてのセッションで有効になるため、ILIB を呼び出すたびにオプションを適用したい場合は、この場所に指定します。ただし、AUTOEXEC.BAT ファイルを変更した場合は、システムをリブートして変更を有効にする必要があります。

ILIB 応答ファイルの使用

応答ファイルを使用して ILIB に入力を提供するには、次のように入力します。

```
ilib @responsefile
```

responsefile は、コマンド行で指定できる情報と同じ情報を含んだファイルの名前です。

なぜ応答ファイルを使用するのか？

以下の場合に応答ファイルを使用します。

- 複雑で長いコマンドを頻繁に入力する
- コマンド・ストリングが、コマンド行長の制限を超える

応答ファイルはコマンド行を拡張し、あらゆるものを応答ファイルに組み込みます。ILIB への入力をコマンド行と応答ファイルに分割するには、コマンド行に入力の一部を指定して、応答ファイルを指定します (応答ファイル名の前にアットマーク (@) を付けます)。アットマークとファイル名の間にはスペースを入れることはできません。

応答ファイル名は、有効な任意の Windows ファイル名にすることができます。ファイル名にスペースまたは @ 記号などの特殊文字を使用するには、ファイル名を引用符で囲む必要があります。

ILIB は、コマンド行で指定した入力と全く同じように、応答ファイルに指定した入力に応答します。引数の間に現れる改行文字は、スペースとして扱われます。したがって、ILIB コマンドを複数行に拡張できます。

注: どのフォーマット・コマンド行を使用するかを指定するオプション (/FREEFORMAT または /NOFREEFORMAT) は、コマンド行の ilib に続く最初のパラメーターとして指定するか、ILIB 環境変数の最初のパラメーターとして指定する必要があります。それらを応答ファイル内で指定することはできません。

ILIB パラメーターの指定例

以下の例では、ILIB にパラメーターを指定する異なる方法を示します。

各例に示す操作によって、既存 mylib.lib ライブラリーから新規ライブラリー newlib.lib およびそのリスト・ファイル newlib.lst が作成されます。mylib.lib は変わりませんが、newlib.lib は以下のように変わります。

- モジュール text が削除されます。
- オブジェクト・ファイル root.obj が、root という名前のオブジェクト・モジュールとして付加されます。
- モジュール table が削除され、root の後に付加される新しい table によって置き換えられます。
- モジュール string が、string.obj という名前のオブジェクト・ファイルにコピーされます。

コマンド行による方法

コマンド行プロンプトで、次のように入力します。

```
ilib /out:newlib.lib /list:newlib.lst mylib.lib /remove:text root table  
/extract:string
```

応答ファイルによる方法

最初に、次の内容の応答ファイルを作成します。

```

/out:newlib.lib
/list:newlib.lst
mylib.lib
/remove:text
root table
/extract:string

```

次に、応答ファイルの名前が `response.fil` の場合は、次のようにして ILIB を呼び出します。

```
ilib @response.fil
```

ILIB 入力の制御

ILIB は、ファイルの内容を調べて入力ファイルのフォーマットを判別します。ほとんどのファイル・フォーマットは、ファイル・ヘッダー情報によって識別できます。入力ファイルのフォーマットが認識できず、ASCII のみが含まれていると思われる場合は、モジュール定義 (`.def`) ファイルとみなされます。

ILIB を使用すると、どのような拡張子でもファイルに指定できます。また、それを正しく処理することもできます。

ILIB 出力の制御

ILIB は、コマンド行に指定されたオプションを調べて、どのような出力が作成されるかを判別します。以下のオプションが ILIB 出力を制御します。

オプション

説明

`/O[UT]:filename`

静的ライブラリーが作成されます。

`/GEND[EF]:filename`

モジュール定義 (`.def`) ファイルが作成されます。短い形式の `/gd` を使用することもできます。

`/GENI[MPLIB]:filename`

インポート・ライブラリーとエクスポート・オブジェクトのペアが作成されます。短い形式の `/gi` を使用することもできます。

`/L[IST]:filename`

リスト・ファイルが作成されます。

上記のどれも指定されない場合、ILIB は、以下のように作成されるものを判別します。

- DEF ファイルが ILIB に対する入力の場合は、インポート・ライブラリーとエクスポート・オブジェクトのペアが作成されます。

注: エクスポートされた記号がない場合、インポート・ライブラリーは作成されません。

- ライブラリーまたはオブジェクト、またはその両方が ILIB に対する入力の場合、それらを結合したライブラリーが作成されます。

ILIB を使用すると、DLL から直接 DEF ファイルを生成できます。ただし、DLL がそのファイルで持つ唯一の情報は装飾されない (エクスポートされた) 名前であるため、記号装飾 (呼び出し規則) およびタイプ情報 (関数またはデータ) は識別できません。ILIB は、DLL からエクスポートされたすべての記号を、_Optlink (デフォルトのリンケージ規則) であるとみなします。ただし、そうでないことを示すオブジェクト・ファイルが提供されない場合に限りです。

DLL で ILIB を使用する最良の方法は、ILIB を使用して、/gd オプションを指定して DEF ファイルを作成することです。必要に応じて DEF ファイルを編集して装飾を変更し、/gi オプションを使用して ILIB によって DEF ファイルを実行し、インポート・ライブラリーとエクスポート・オブジェクトのペアを作成します。

インポート・ライブラリーとエクスポート・オブジェクトのペアを要求したときに、入力として DLL のみを指定すると、ILIB はエラーを生成します。

ILIB 出力の制御

ILIB 出力を制御する方法を示す例を、以下に示します。

ライブラリー

次の例は、text.obj および mylib.lib のオブジェクトからライブラリー newlib.lib を作成します。

```
ilib /out:newlib.lib text.obj mylib.lib
```

注: 入力ファイルとして newlib.lib を指定しない場合は、ライブラリーにその内容は含まれません。出力ファイルがすでに存在しており、それを入力ファイルとして使用しない場合は、そのファイルが置き換わります。

DEF ファイル

この例は、DLL の winner.dll からモジュール定義ファイルの winner.def を作成します。

```
ilib /gd:winner.def winner.dll
```

インポート・ライブラリーとエクスポート・オブジェクトのペア

次の例は、winner.lib という名前のインポート・ライブラリーおよび winner.exp という名前のエクスポート・オブジェクトを作成します。ただし、winner.def にエクスポートされた記号が含まれない場合は、winner.lib は作成されません。

```
ilib /gi winner.def
```

リスト・ファイル

次の例は、ライブラリーの mylib.lib に基づいて、現行ディレクトリーにリスト・ファイルの mylib.lst を生成します。

```
ilib /list:mylib.lst mylib.lib
```

ILIB オブジェクト

ILIB オブジェクトは、ライブラリーでモジュールを操作するときに使用します。ILIB を実行する場合は、複数のオブジェクトを任意の順番で指定できます。

各オブジェクトは、後にコマンドの対象であるオブジェクト・モジュールの名前が続いた ILIB コマンドから構成されています。コマンド行では、スペースまたはタブ文字を使用してオブジェクトを区切ります。

ILIB オブジェクトの要約

Windows における ILIB オブジェクトの要約を以下に示します。

表 29. Windows における ILIB オブジェクト

構文	説明	デフォルト	ページ
<i>filename</i>	ライブラリーにおいて指定されたオブジェクトを追加/置換します。	なし	379
/E[XTRACT]: <i>obj</i>	指定されたオブジェクトを現行ディレクトリーにコピーし、すでに存在する場合は、それを上書きします。	なし	380
/R[EMOVE]: <i>obj</i>	指定されたオブジェクトを、出力ライブラリーに置かれるオブジェクトのリストから除去します。	なし	381

注:

- ILIB オブジェクトには、大/小文字の区別はありません。したがって、小文字、大文字、または大/小文字混合でオブジェクトを指定できます。
- オブジェクトの前のダッシュ (-) とスラッシュ (/) を置換することもできます。例えば、-REMOVE:*filename* と /REMOVE:*filename* は同じことを意味します。
- オブジェクトは、短い形式または長い形式のいずれでも指定できます。例えば、/R:*filename* と /RE:*filename* は /REMOVE:*filename* と同じことを意味します。
- コマンド行を処理するときの操作は、左から右の順番に行われます。
- ILIB は、入力ライブラリーが実行中は、それを変更しません。ライブラリーをコピーして、コピーを変更します。ILIB が中断されると、オリジナル・ライブラリーがリストアされます。

出力ライブラリーが指定されないと、ILIB は出力を生成しません。

Add/Replace

➡—*filename*—➡

デフォルトのアクション (関連オブジェクトなしにコマンド行に *filename* を指定) は、そのファイルをライブラリーに追加します。*filename* がすでにライブラリーに存在する場合は、置き換えられます。

オブジェクト・モジュールのライブラリーへの追加

追加するオブジェクト・ファイルの名前をコマンド行に入力します。`.obj` 拡張子は省略できます。

ILIB は、ライブラリーでは、オブジェクト・ファイルのベース名をオブジェクト・モジュールの名前として使用します。例えば、オブジェクト・ファイル `cursor.obj` がライブラリー・ファイルに追加されると、対応するオブジェクト・モジュールの名前は `cursor` です。

オブジェクト・モジュールは、常にライブラリー・ファイルの最後に追加されます。

オブジェクト・モジュールのライブラリーでの置換

置換するオブジェクト・モジュールの名前をコマンド行に入力します。`.obj` 拡張子は省略できます。

オブジェクト・モジュールがすでにライブラリーに存在する場合、ILIB は、それを新規コピーに置き換えます。

2 つのライブラリーの結合

追加するライブラリー・ファイルの名前を、`.lib` 拡張子を含めて、コマンド行に指定します。そのライブラリーの内容のコピーが、変更するライブラリー・ファイルに追加されます。両方のライブラリーに同じ名前のモジュールが含まれている場合、ILIB は警告メッセージを生成し、同じ名前を持った最初のモジュールのみを使用します。

ILIB は、ライブラリーのモジュールを、変更するライブラリーの最後に追加します。ILIB はモジュールを削除せずにそれらをコピーするため、追加されたライブラリーは、依然として、独立ライブラリーとして存在します。

例

次のコマンドは、ファイル `sample.obj` をライブラリー `mylib.lib` に追加します。`sample.obj` がすでにライブラリー `mylib.lib` に存在する場合、ILIB はそれを置き換えます。

```
ilib /out:mylib.lib mylib.lib sample.obj
```

この例は、ライブラリー `mylib.lib` の内容を、ライブラリー `newlib.lib` に追加します。ライブラリー `mylib.lib` は、このコマンドの実行後も変わりません。

```
ilib /out:newlib.lib newlib.lib mylib.lib
```

/EXTRACT

▶▶—/E[XTRACT]:—obj————▶▶

/EXTRACT は、ライブラリーのモジュールを、同じ名前のオブジェクト・ファイルにコピーするときに使用します。モジュールは、ライブラリー内ではもとのままです。

ILIB は、モジュールをオブジェクト・ファイルにコピーする場合、モジュール名に .obj 拡張子を追加し、そのファイルを現行ディレクトリーに置きます。この名前を持ったファイルがすでに存在する場合、ILIB はそれを上書きします。

例 次のコマンドは、mylib.lib ライブラリーのモジュール sample を、現行ディレクトリーの sample.obj と呼ばれるファイルにコピーします。mylib.lib 内のモジュール sample は変更されません。

```
ilib mylib.lib /extract:sample
```

/REMOVE



/REMOVE は、ライブラリーからオブジェクト・モジュールを削除するときに使用します。/REMOVE の後に、削除するモジュールの名前を指定します。モジュール名は、パス名または拡張子を持ちません。

例 次のコマンドは、モジュール sample を、ライブラリー mylib.lib から削除します。

```
ilib /out:mylib.lib mylib.lib /remove:sample
```

次のコマンドは、mylib.lib ライブラリーの sample.obj を、現行ディレクトリーのオブジェクト・ファイルにコピーし、次に、sample.obj をライブラリーから削除します。

```
ilib /out:mylib.lib mylib.lib /extract:sample /remove:sample
```

ILIB オプション

ILIB オプションは、ILIB の動作に影響を与えます。ILIB を実行する場合は、複数のオプションを任意の順番で指定できます。/FREEFORMAT オプションのみは例外であり、位置の制限があります。

コマンド行では、スペースまたはタブ文字を使用してオプションを区切ります。

ILIB オプションの要約

Windows における ILIB オプションの要約を以下に示します。

表 30. Windows における ILIB オプション

構文	説明	デフォルト	ページ
/?	ヘルプを表示します	なし	382
/BA[CKUP] /NOBA[CKUP]	上書きする前に 出力ファイルをバックアップします (存在する場合)	/BA	382
/DEF:def	エクスポートされた記号および リンカー・パラメーターに関する 情報の取得に使用する .def ファイルの名前を指定します	なし	383

Windows における ILIB の使用

表 30. Windows における ILIB オプション (続き)

構文	説明	デフォルト	ページ
/F[REEFORMAT] /NOF[REEFORMAT]	フリー・フォーマットのコマンド行を 使用します	/F	383
/GEND[EF]:filename	.def ファイルを生成します	なし	383
/GENI[MPLIB]:filename	インポート・ライブラリーを生成します	なし	383
/H[ELP]	ヘルプを表示します	なし	384
/L[IST]:filename	リスト・ファイルを生成します	なし	384
/NOE[XTDICTIONARY] /EXTD[ICTIIONARY]	OMF ライブラリーに拡張ディクショ ナリーを生成しません	/EXTD	384
/O[UT]:filename	出力ライブラリーの名前を 指定します	なし	385
/Q[UIET], /NOL[OGO] /LO[GO], /NOQ[UIET]	始動時にバナーを表示 しません	/LO	385
/W[ARN:msgnum,msgnum[,...]] /NOW[ARN:msgnum,msgnum[,...]]	警告メッセージ番号 <i>msgnum</i> の印刷を使用可能にします	なし	385

注:

1. ILIB オプションには、大/小文字の区別はありません。したがって、小文字、大文字、または大/小文字混合でオプションを指定できます。

オプションの前のダッシュ (-) とスラッシュ (/) を置換することもできます。例えば、-FREEFORMAT と /FREEFORMAT は同じことを意味します。

2. オプションは、短い形式または長い形式のいずれでも指定できます。例えば、/F、/FR、および /FREE は、/FREEFORMAT と同じことを意味します。

各 ILIB オプションの詳細については、下記を参照してください。

/?

▶▶ /? —————▶▶

/? は、有効な ILIB オプションのリストを表示するときに使用します。このオプションは /HELP と同等です。

/BACKUP

▶▶

/BA[CKUP]

/NOBA[CKUP]

 —————▶▶

/BACKUP は、上書きする前に出力ファイル (存在する場合) をバックアップするときに使用します

ILIB は、ライブラリーのベース名をバックアップ・ライブラリーの名前として使用し、.bak 拡張子を付加します。例えば、変更するライブラリーが mylib.lib のときに、バックアップを要求すると、ILIB は、現行ディレクトリーに mylib.bak を作成します。

/DEF

➡—/DEF—:filename—➡

/DEF は、エクスポートされた記号およびリンカー・パラメーターに関する情報の取得に使用する .def ファイルの名前を指定するときに使用します。

.def ファイルがその他の入力ファイルと一緒にコマンド行に指定される場合、ILIB は、内容からそれらのファイルを認識するため、このオプションは必須ではありません。

/FREEFORMAT

➡—[/F[REEFORMAT]]
—[/NOF[REEFORMAT]]—➡

/FREEFORMAT オプションは、フリー・フォーマットのコマンド行を使用することを ILIB に知らせるときに使用します。フリー・フォーマットのコマンド行を使用すると、ILIB 入力引数を、任意の回数、任意の順番で指定できます。

注: このオプションは、コマンド行で ilib の直後に指定するか、ILIB 環境変数の最初の引数として指定する必要があります。/FREEFORMAT または /NOFREEFORMAT のいずれも指定しない場合、ILIB は、フリー・フォーマット・コマンド行にデフォルト設定されます。

/GENDEF

➡—[/GEND[EF]:] filename—➡

/GENDEF オプションは、モジュール定義 (.def) ファイルを作成するときに使用します。

例

次のコマンドは、モジュール定義ファイルの sample.def を DLL の sample.dll から作成します。

```
ilib /gd:sample.def sample.dll
```

/GI

▶▶—/GENI[IMPLIB]—:filename—▶▶

/GENIMPLIB オプションは、インポート・ライブラリーとエクスポート・オブジェクトのペアを作成するときに使用します。

例

次のコマンドは、sample.lib という名前のインポート・ライブラリーおよび sample.exp という名前のエクスポート・オブジェクトを、モジュール定義ファイル sample.def から作成します。ただし、エクスポートされた記号が含まれない場合は、sample.lib は作成されません。

```
ilib /gi sample.def
```

/HELP

▶▶—/H[ELP]—▶▶

/HELP は、有効な ILIB オプションのリストを表示するときに使用します。このオプションは /? と同等です。

/LIST

▶▶—/L[IST]—filename—▶▶

/LIST オプションは、リスト・ファイルを生成するときに使用します。filename を指定しないと、ILIB は、入力ファイル名に拡張子 .lst を追加します。

例

次のコマンドは、mylib.lib の内容のリストを、ファイル mylib.lst に入れるように ILIB に指示します。mylib.lst に対して、パス指定は行いません。作成されるファイルは、デフォルトで、現行ディレクトリーに置かれます。

```
ilib mylib /list:mylib.lst
```

注: /LISTLEVEL オプションは、ILIB の Windows リリースではサポートされません。

/NOEXT

▶▶

/NOE[XTDICTIONARY]
/EXTD[ICTIONARY]

—▶▶

/NOEXTDICTIONARY は、拡張ディクショナリーの生成を使用不可にするときに使用します。

拡張ディクショナリーは、リンク速度を早める、ライブラリーのオプション部分です。ただし、拡張ディクショナリーを使用すると、多くのメモリーが必要になります。

す。拡張ディクショナリーに予約されるスペースは 64K に制限されています。ILIB がメモリー不足 エラーを報告する場合、このオプションを使用することをお勧めします。代わりの方法として、リンクに使用するために、大きなライブラリーを小さなライブラリーに分割することもできます。

/OUT

▶▶—————▶▶

/QUIET

▶▶—————▶▶

/QUIET または /NOLOGO オプションは、ILIB の著作権表示を抑制するときに使用します。

/WARN

▶▶—————▶▶

/WARN オプションは、*msgnum* パラメーターで指定したメッセージ番号の印刷を使用可能にするときに使用します。

第 24 章 呼び出し規則

リンケージについての考慮事項の理解	387	SYSTEM の機能	395
OPTLINK リンケージ	389	SYSTEM リンケージの使用例	396
OPTLINK の機能	389	STDCALL リンケージ (Windows のみ)	398
OPTLINK 使用のヒント	390	STDCALL の機能	398
汎用レジスタの説明	390	STDCALL 規則の使用例	399
パラメーター	390	WinMain の使用 (Windows のみ)	400
パラメーターの受け渡し例	390	CDECL リンケージ	400
規格合致パラメーターのルーチンへの受け渡 し	390	CDECL の機能	401
SYSTEM リンケージ	395	CDECL 規則の使用例	401

この章では、PL/I for Windows によって使用される呼び出し規則について説明します。

OPTLINK
SYSTEM
STDCALL
CDECL

OPTLINK リンケージ規則 (詳細は 389 ページの『OPTLINK リンケージ』を参照) は、VisualAge for C++ (OS/2 および Windows) でもサポートされており、PL/I プロシージャ、C 関数、またはアセンブラ・ルーチンを呼び出す最速の方法です。ただし、OPTLINK は、すべての Windows アプリケーションで標準になっているわけではありません。

Windows では、SYSTEM リンケージは STDCALL リンケージと同じ意味を持ち、STDCALL と同じようにインプリメントされています。ただし、コンパイラは、SYSTEM と STDCALL を別個の名前とみなし、それらが混同されていると、クレームをつけます。STDCALL リンク規則は、398 ページの『STDCALL リンケージ (Windows のみ)』で説明します。

DEFAULT コンパイル時オプションの LINKAGE サブオプションを使用して、プログラム内のすべての関数に対して呼び出し規則を指定できます。OPTIONS 属性の LINKAGE オプションを使用して、個々の関数のリンケージを指定することもできます。

注: 関数のコンパイルに使用したものとは異なる呼び出し規則を使用して、その関数を呼び出すことはできません。例えば、SYSTEM リンケージで関数をコンパイルした場合は、後から OPTLINK リンケージを指定してその関数を呼び出すことはできません。

リンケージについての考慮事項の理解

Windows では、PL/I コンパイラは、OPTLINK、CDECL、および STDCALL という 3 つの主なリンケージをサポートします。Windows では、すべてのシステム・サービスが STDCALL リンケージを使用します。

これらのリンケージは、パラメーター引き渡し規則が異なります。

リンケージについての考慮事項の理解

- OPTLINK リンケージのみが、レジスター内の一部のパラメーターを受け渡します。その他のリンケージは、スタック上のすべてのパラメーターを受け渡します。
- STDCALL リンケージのみが、呼び出し先がスタックをクリーンアップするようにします。その他のリンケージは、呼び出し側がそれを行うようにします。

PL/I for Windows コンパイラーは、SYSTEM リンケージが指定されると、STDCALL リンケージが指定されたかのように解釈します。VisualAge C コンパイラーも同じように解釈します。

Windows では、すべての外部名が装飾されます。外部属性が名前を指定しない場合、名前の装飾はリンケージによって異なります。

- CDECL リンケージを使用したルーチンでは、接頭部として「_」が追加されます。したがって、例えば FUNKY という名前は _FUNKY になります。
- OPTLINK リンケージを使用したルーチンでは、接頭部として「?」が追加されます。したがって、例えば FUNKY という名前は ?FUNKY になります。
- STDCALL リンケージを使用したルーチンでは、接頭部として「_」が追加され、接尾部として、パラメーターによって使用されるバイトが後に付いた「@」が追加されます。例えば、FUNKY という名前が、2 つの byvalue ポインターまたは byaddr パラメーターを持つ場合、_FUNKY@8 という名前になります。

これらの名前装飾によって、ルーチンの呼び出し側が、そのルーチンの誤ったリンケージを指定すると、プログラムはリンクに失敗することになります。

ここまでの名前装飾の説明は、外部属性が名前を指定しなかったルーチンについての説明でした。この説明は、外部属性が、宣言された名前と大/小文字のみが異なる名前を指定した場合についても当てはまります。このような場合は、外部属性の一部として指定された名前が装飾されます。

例えば、次の宣言の場合、リンカーには ?getenv という名前が見えます。

```
dc1 getenv ext('getenv')
  entry( char(*) varz byaddr nonasgn )
  returns( pointer )
  options( nodestructor linkage(optlink) );
```

同じように、次の宣言では (DLL をロードする Windows システム・ルーチンの場合)、外部属性の一部として指定された名前は装飾され、リンカーには _LoadLibraryA@4 という名前が見えます。

```
dc1 loadlibrarya ext('LoadLibraryA')
  entry( char(*) varz byaddr nonasgn )
  returns( pointer byvalue )
  options( linkage(stdcall) nodestructor );
```

ただし、名前が外部属性の一部として指定され、その名前が、大/小文字以外に宣言された名前と異なる場合、名前の装飾は行われません。

例えば、次の宣言の場合、名前の装飾は行われず、リンカーには ?getenv という名前が見えます。

```
dc1 getenv ext('?getenv')
  entry( char(*) varz byaddr nonasgn )
  returns( pointer )
  options( nodestructor linkage(optlink) );
```

この最後の例に示したように、名前装飾をユーザー自身が実行すると、通常、コードは移植しにくいものになります。例えば、Windows および AIX の場合、これまでの例では、`getenv` の最初の宣言のみが有効です。

OPTLINK リンケージ

これはデフォルトの呼び出し規則です。オペレーティング・システムの呼び出しに通常使用される `SYSTEM` リンケージの代替リンケージです。このリンケージの総合パフォーマンスは、`SYSTEM` リンケージのパフォーマンスよりも優れています。

OPTLINK の機能

OPTLINK 規則には以下の機能があります。

- パラメーターは、スタック上で右から左へプッシュされる。
- 呼び出し側は、スタックをクリーンアップする。
- 汎用レジスター `EBX`、`EDI`、および `ESI` は、呼び出しを超えて、保持される。
- 汎用レジスター `EAX`、`ECX`、および `EDX` は、呼び出しを超えて、保持されない。
- 浮動小数点レジスターは、呼び出しを超えて、保持されない。
- 左端の字句である 3 つの規格合致パラメーター (規格合致パラメーターは、すべての `BYADDR` パラメーターのアドレス、および次の `BYVALUE` パラメーターです。ポインター、ハンドル、序数、オフセット、限定エンタリー、実固定バイナリー、`character(1)`、および 1 バイト以下を占有する非変化ビット。) は、3 つの非保持汎用レジスターで受け渡される。
- 最大 4 つの実浮動小数点または 2 つの複合パラメーター (最初の 4 つの字句) は、浮動小数点レジスター・スタックにおいて、拡張精度フォーマット (80 ビット) で受け渡される。
- レジスターで受け渡されないすべての規格合致パラメーター、およびすべての規格合致外パラメーターは、80386 スタック上で受け渡される。
- レジスターにおけるパラメーター用のスペースはスタック上で割り当てられるが、パラメーターはそのスペースにはコピーされない。
- 規格合致戻り値は `EAX` に戻される。
- 実浮動小数点戻り値は、拡張精度フォーマットで、浮動小数点スタックの一番上のレジスターに戻される。
- 複合浮動小数点戻り値は、拡張精度フォーマットで、浮動小数点スタックの一番上の 2 つのレジスターに戻される。
- 外部関数を呼び出す場合、浮動小数点レジスター・スタックには、入り口上の有効なパラメーター・レジスター、および出口上の有効な戻り値のみが含まれる。
- 集合を戻す関数は、呼び出し側によって決まるストレージ域のアドレスを、隠しパラメーターとして受け渡す。この区域は戻された集合になる。この集合のアドレスは `EAX` に戻される。
- 方向フラグは、関数の入り口上、および関数からの出口上で、クリアする必要がある。その他のフラグの状態は、関数への入り口上では無視され、出口上では定義されない。

- コンパイラーは、浮動小数点制御レジスターの内容を変更しない。特定の操作について、制御レジスターの内容を変更したい場合は、変更する前に内容を保管し、操作の後にそれらをリストアします。

OPTLINK 使用のヒント

OPTLINK リンケージを使用するときに下に示したヒントに従うと、アプリケーションのパフォーマンスを改善することができます。

- 最も頻繁に使用される規格合致および浮動小数点パラメーターは、レジスターがそれらを最初に考慮に入れるように、パラメーター・リストにおいては左端の字句にします。それらが互いに隣接している場合は、パラメーター・リストを準備するとより速くなります。
- 関数の最後の方で使用するパラメーターの場合、パラメーター・リストの最後または最後の方に、そのパラメーターを置きます。すべてのパラメーターが関数の最後の方で使用する場合は、SYSTEM リンケージの使用をお勧めします。
- OPTIMIZE を使用してコンパイルします。(74 ページの『OPTIMIZE』 参照を参照してください。)

汎用レジスターの説明

パラメーター

EAX、EDX、および ECX は、最初の 3 つの規格合致パラメーターの字句に使用されます。EAX には最初のパラメーター、EDX には 2 番目のパラメーター、ECX には 3 番目のパラメーターが含まれます。4 バイトのスタック・ストレージが、存在する各レジスター・パラメーターに割り当てられますが、パラメーターは、呼び出し時には、レジスターにのみ存在します。

パラメーターの受け渡し例

以下の例は、わかりやすく例示するためにのみ記載されたものであり、最適化されていません。これらの例では、ユーザーがアセンブラーのプログラミングに習熟していることが前提になっています。各例では、スタックはページの下方向へ伸びていきます。また、ESP は常に、スタックの上部を指します。

規格合致パラメーターのルーチンへの受け渡し

以下の例は、関数 FUNC1 に対する呼び出しのときの、コード・シーケンスおよびスタックの図を示しています。このプログラムは、PREFIX(NOFIXEDOVERFLOW) オプションでコンパイルされることが前提になっています。

```
dc1 func1 entry( char(1),
                fixed bin(15),
                fixed bin(31),
                fixed bin(31) )
returns( fixed bin(31) )
options( byvalue nodescriptor );
```

```
dc1 x fixed bin(15);
dc1 y fixed bin(31);

y = func1('A', x, y+x, y);

caller's Code Up Until Call:
```

```

PUSH    y          ; Push p4 onto the 80386 stack
SUB     ESP, 12     ; Allocate stack space for
                    ; register parameters
MOV     AL, 'A'     ; Put p1 into AL
MOV     DX, x       ; Put p2 into DX
MOVSX   ECX, DX     ; Sign-extend x to long
ADD     ECX, y       ; Calculate p3 and put it into ECX
CALL    FUNC1       ; Make call

```

呼び出し直後のスタック

呼び出し直後のレジスタ設定

ESP →	呼び出し側の Local	EAX	未定義	p1
	p4	EBX	呼び出し側の EBX	
	p3 のブランク・スロット	ECX	p3	
	p2 のブランク・スロット	EDX	未定義	p2
	p1 のブランク・スロット	EDI	呼び出し側の EDI	
	呼び出し側の EIP	ESI	呼び出し側の ESI	

callee's Prolog Code:

```

PUSH    EBP         ; Save caller's EBP
MOV     EBP, ESP    ; Set up callee's EBP
SUB     ESP, callee's local size ; Allocate callee's Local
PUSH    EBX         ; Save preserved registers -
PUSH    EDI         ; will optimize to save
PUSH    ESI         ; only registers callee uses

```

Prolog 後のスタック

Prolog 後のレジスタ設定

ESP →	呼び出し側の Local	EAX	未定義	p1
	p4	EBX	未定義	
	p3 のブランク・スロット	ECX	p3	
	p2 のブランク・スロット	EDX	未定義	p2
	p1 のブランク・スロット	EDI	未定義	
	呼び出し側の EIP	ESI	未定義	
	呼び出し側の EBP			
	呼び出し側の Local			
	保管 EBX			
	保管 EDI			
	保管 ESI			

レジスタ EBX、EDI、および ESI における「未定義」という用語は、FUNC1 のコードで安全に上書きできることを意味します。

callee's Epilog Code:

```

MOV     EAX, RetVal ; Put return value in EAX

```

汎用レジスターの説明

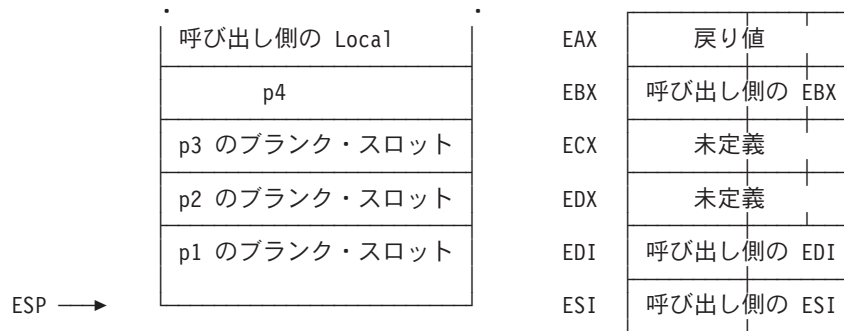
```

POP     ESI      ; Restore preserved registers
POP     EDI
POP     EBX
MOV     ESP, EBP ; Deallocate callee's local
POP     EBP      ; Restore caller's EBP
RET     ; Return to caller

```

Epilog 後のスタック

Epilog 後のレジスター設定



caller's Code Just After Call:

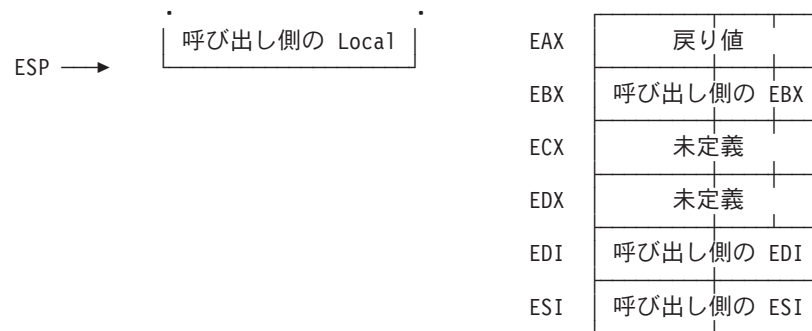
```

ADD     ESP, 16   ; Remove parameters from stack
MOV     y, EAX    ; Use return value.

```

クリーンアップ後のスタック

クリーンアップ後のレジスター設定



浮動小数点パラメーターのルーチンへの受け渡し: 以下の例は、ルーチン FUNC2 に対する呼び出しのときの、コード・シーケンス、80386 のスタック・レイアウト、および浮動小数点レジスター・スタックの状態を示しています。単純にするために、汎用レジスターは示しません。このプログラムは、IMPRECISE オプションでコンパイルされることが前提になっています。

```

dcl func2 entry( float bin(21),
                  float bin(53),
                  float bin(64),
                  float bin(21),
                  float bin(53) )
returns( float bin(53) )
options( byvalue nodescrptor );

dcl (a, b, c) float bin(53);
dcl (d, e) float bin(21);

a = b + func2(a, d, prec(a + c, 53), e, c);

```

caller's Code Up Until Call:

```

PUSH    2ND DWORD OF c      ; Push upper 4 bytes of c onto stack
PUSH    1ST  DWORD OF c      ; Push lower 4 bytes of c onto stack

```

```

FLD    DWORD_PTR e          ; Load e into 80387, promotion
                                ; requires no conversion code
FLD    QWORD_PTR a          ; Load a to calculate p3
FADD   ST(0), QWORD_PTR c    ; Calculate p3, result is float bin(64)
                                ; from nature of 80387 hardware
FLD    QWORD_PTR d          ; Load d, no conversion necessary
FLD    QWORD_PTR a          ; Load a, demotion requires conversion
FSTP   DWORD_PTR [EBP - T1]  ; Store to a temp (T1) to convert to float
FLD    DWORD_PTR [EBP - T1]  ; Load converted value from temp (T1)
SUB     ESP, 32              ; Allocate the stack space for
                                ; parameter list
CALL    FUNC2                ; Make call

```

呼び出し直後のスタック

呼び出し直後の 80387 レジスタ設定

呼び出し側の Local	ST(7)	空
p5 の Upper Dword	ST(6)	空
p5 の Lower Dword	ST(5)	空
p4 の Blank Dword	ST(4)	空
Four	ST(3)	p4 (e)
ブランク	ST(2)	p3 (a + c)
Dwords	ST(1)	p2 (d)
for p3	ST(0)	p1 (a)
2 つのブランク		
p2 の Dwords		
p1 の Blank Dword		
呼び出し側の EIP		

ESP →

callee's Prolog Code:

```

PUSH    EBP                  ; Save caller's EBP
MOV     EBP, ESP             ; Set up callee's EBP
SUB     ESP, callee's local size ; Allocate callee's Local
PUSH    EBX                  ; Save preserved registers -
PUSH    EDI                  ; will optimize to save
PUSH    ESI                  ; only registers callee uses

```

Prolog 後のスタック

Prolog 後の 80387 レジスター設定

呼び出し側の Local	ST(7)	空
p5 の Upper Dword	ST(6)	空
p5 の Lower Dword	ST(5)	空
p4 の Blank Dword	ST(4)	空
Four	ST(3)	p4
ブランク	ST(2)	p3
Dwords	ST(1)	p2
for p3	ST(0)	p1
2 つのブランク		
p2 の Dwords		
p1 の Blank Dword		
呼び出し側の EIP		
呼び出し側の EBP		
呼び出し先の Local		
保管 EBX		
保管 EDI		
保管 ESI		

ESP →

callee's Epilog Code:

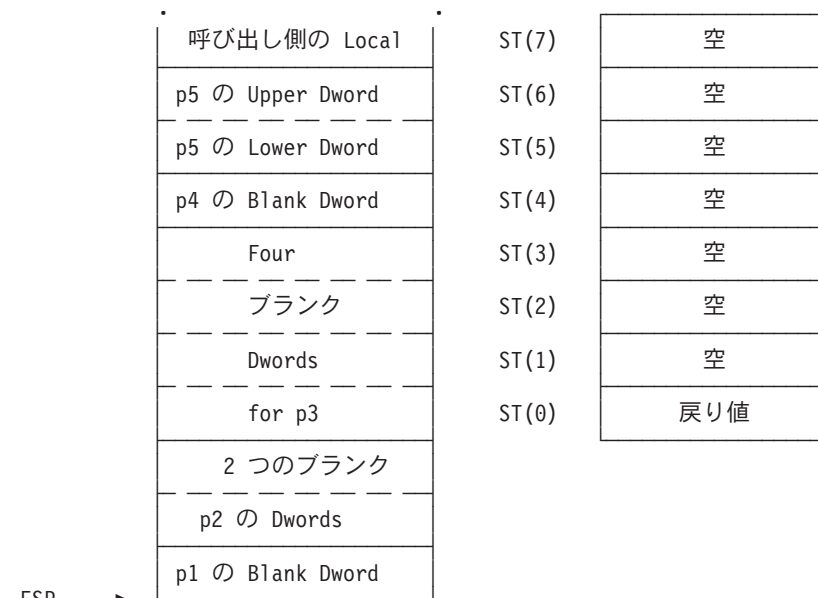
```

FLD    RETVAL    ; Load return value onto floating-point stack
POP    ESI       ; Restore preserved registers
POP    EDI
POP    EBX
MOV    ESP, EBP  ; Deallocate callee's local
POP    EBP       ; Restore caller's EBP
RET                     ; Return to caller

```

Epilog 後のスタック

Epilog 後の 80387 レジスタ設定



caller's Code Just After Call:

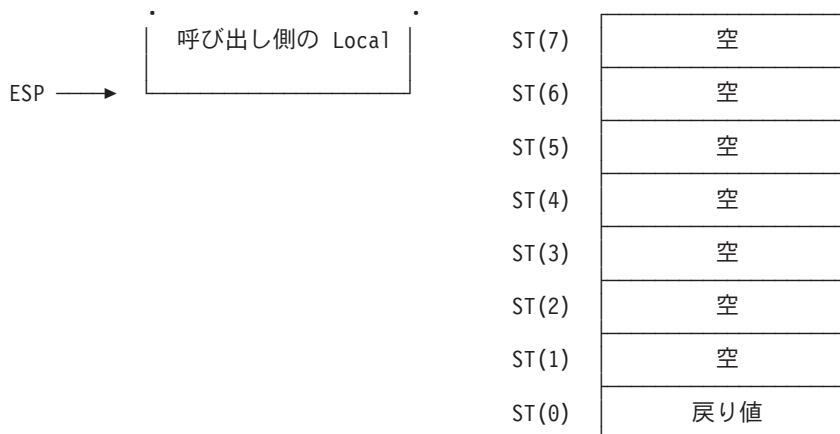
```

ADD     ESP, 40      ; Remove parameters from stack
FADD    QWORD_PTR b  ; Use return value
FSTP    QWORD_PTR a  ; Store expression to variable a

```

クリーンアップ後のスタック

クリーンアップ後の 80387 レジスタ設定



SYSTEM リンケージ

このリンケージ規則を使用するには、関数の宣言において `OPTIONS(LINKAGE(SYSTEM))` 属性を指定するか、`DEFAULT(LINKAGE(SYSTEM))` コンパイル時オプションを指定する必要があります。

SYSTEM の機能

SYSTEM リンケージ規則には、以下の規則が適用されます。

- すべてのパラメーターは、80386 スタック上で受け渡される。
- パラメーターは、右から左の順番でスタック上にプッシュされる。

- 呼び出し関数が、スタックからパラメーターを除去する。
- すべてのパラメーターは、ダブルワード (4 バイト) に調整される。
- 値は OPTLINK リンケージと同じ方法で戻される。
- 方向フラグは、関数の入り口上、および関数からの出口上で、クリアする必要がある。その他のフラグの状態は、関数への入り口上では無視され、出口上では定義されない。
- コンパイラーは、浮動小数点制御レジスターの内容を変更しない。特定の操作について、制御レジスターの内容を変更したい場合は、変更する前に内容を保管し、操作の後にそれらをリストアします。

SYSTEM リンケージの使用例

以下の例は、わかりやすく例示するためにのみ記載されたものであり、最適化されていません。この例では、ユーザーがアセンブラーのプログラミングに習熟していることが前提になっています。この例では、スタックはページの下方向へ伸びていきます。また、ESP は常に、スタックの上部を指します。

以下の例は、2 つのローカル変数 x および y (両方とも fixed bin(31)) を持つ関数 FUNC3 に対する呼び出しのときの、コード・シーケンスおよびスタックの図を示しています。次の呼び出しの場合、

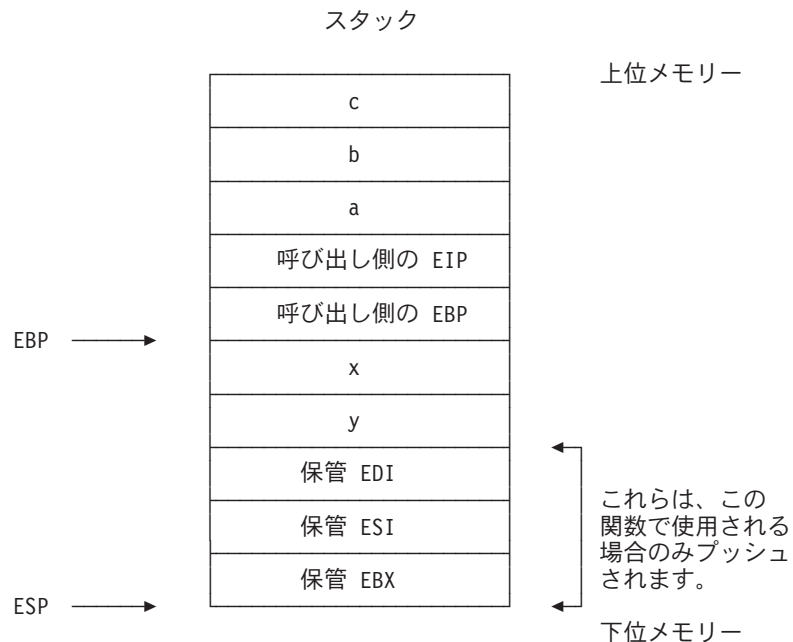
```

dcl func3 entry( fixed bin(31),
                fixed bin(31),
                fixed bin(31) )
returns( fixed bin(31) )
options( byvalue nodestructor linkage(system) );

m = func3(a,b,c);

```

FUNC3 に対する呼び出しのスタックは次のようになります。



スタック上にこの活動化レコードを作成するのに使用される命令は、呼び出し側では次のようになります。

```

PUSH    c
PUSH    b
PUSH    a
MOV     AL, 3H
CALL    func3
.
.
ADD     ESP, 12    ; Cleaning up the parameters
.
.
MOV     m, EAX
.
.

```

呼び出し先では、コードは次のようになります。

```

func3 PROC
PUSH    EBP
MOV     EBP, ESP    ; Allocating 8 bytes of storage
SUB     ESP, 8      ; for two local variables.
PUSH    EDI          ; These would only be
PUSH    ESI          ; pushed if they were used
PUSH    EBX          ; in this function.
.
.
MOV     EAX, [EBP - 8] ; Load y into EAX
MOV     EBX, [EBP + 12] ; Load b into EBX
.
.
XOR     EAX, EAX      ; Zero the return value
POP     EBX          ; Restore the saved registers
POP     ESI
POP     EDI
LEAVE                   ; Equivalent to MOV ESP, EBP
                        ; POP EBP
RET
func3 ENDP

```

保管レジスター・セットは EBX、ESI、および EDI です。その他のレジスター (EAX、ECX、および EDX) は、呼び出し先ルーチンによって変更された内容を持つことができます。

状況によっては、コンパイラーは、EBP を使用せずに自動値またはパラメーター値にアクセスするため、アプリケーションの効率が向上する場合があります。使用されるかされないかに関係なく、EBP は呼び出しを超えては変更されません。

値によって集合を受け渡す場合、コンパイラーは、80386 スタック上に集合をコピーするコードを生成します。集合のサイズが 80386 のページ・サイズ (4K) より大きい場合、コンパイラーは、集合を逆方向にコピーするコードを生成します (つまり、集合の最後のバイトが最初にコピーされます)。

集合はスタック上には戻されません。呼び出し側は、戻される集合が置かれるアドレスを、最初の隠しパラメーター字句としてプッシュします。集合を戻す関数は、すべてのパラメーターが、集合が戻されない場合のパラメーターよりも、EBP から 4 バイト遠くに離れるように注意する必要があります。戻される集合のアドレスは、EAX に戻されます。

STDCALL リンケージ (Windows のみ)

このリンケージ規則を使用するには、関数の宣言において `OPTIONS(LINKAGE(STDCALL))` 属性を指定するか、`DEFAULT(LINKAGE(STDCALL))` コンパイル時オプションを指定する必要があります。

STDCALL の機能

STDCALL 呼び出し規則には、以下の規則が適用されます。

- すべてのパラメーターは、スタック上で受け渡される。
- パラメーターは、字句の右から左の順番でスタック上にプッシュされる。
- 呼び出し先の 関数が、スタックからパラメーターを除去する。
- 浮動小数点値は、浮動小数点レジスター・スタックのトップ・レジスターである `ST(0)` に戻される。集合値を戻す関数は、それらの値を以下のように戻します。

戻される集合値 のサイズ

8 バイト

EAX-EDX ペア

5、6、7 バイト

EAX 戻り値を置くアドレスは、EAX において隠しパラメーターとして受け渡されます。

4 バイト

EAX

3 バイト

EAX 戻り値を置くアドレスは、EAX に隠しパラメーターとして受け渡されます。

2 バイト

AX

1 バイト

AL

サイズが 5、6、7、または 8 バイトより大きい集合を戻す関数の場合、戻り値を置くアドレスは、隠しパラメーターとして受け渡され、アドレスは EAX に戻されます。

- STDCALL には、引数の数が変わるプロトタイプ化されていない STDCALL 関数は動作しない、という制約事項があります。
- 関数名は、下線接頭部、およびパラメーターのバイト数 (10 進数) が後に付いたアットマーク (@) で構成される接尾部によって装飾される。4 バイトより少ないパラメーターは、4 バイトに切り上げられます。構造体サイズも、複数の 4 バイトに切り上げられます。例えば、次のようにプロトタイプ化された関数 `fred` を考えてみます。

```
decl fred ext entry (fixed bin(31) byvalue, fixed bin(31) byvalue,
                    fixed bin(15) byvalue);
```

この関数は、オブジェクト・モジュールでは次のようになります。

```
_FRED@12
```

.DEF ファイルでエクスポート・リストを作成する場合は、装飾されたバージョンの名前を使用する必要があります。DEF ファイルにおいて装飾されていない名前を使用する場合は、ILIB に対し、DEF ファイルと一緒にオブジェクト・ファイルを提供する必要があります。ILIB は、オブジェクト・ファイルを使用して、それぞれの名前が装飾後にどのようなになるかを判別します。

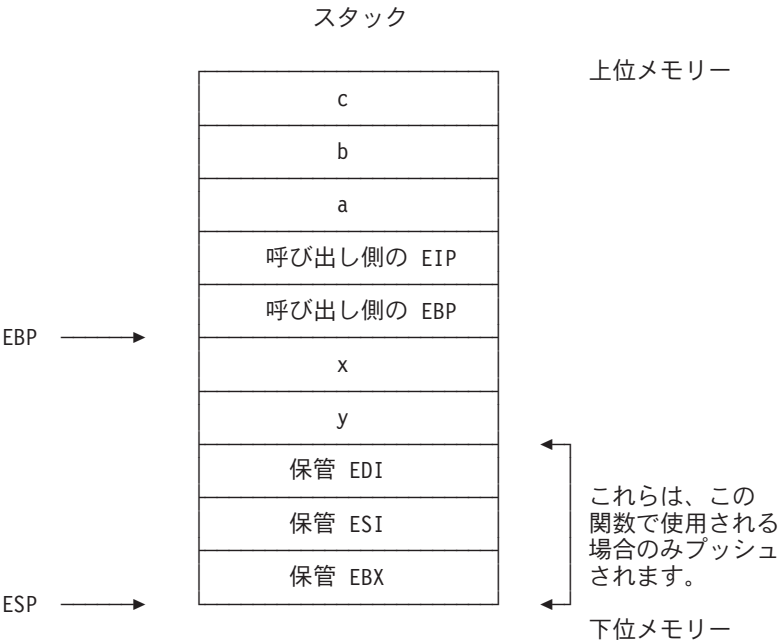
STDCALL 規則の使用例

以下の例は、わかりやすく例示するためにのみ記載されたものです。これらの例では、ユーザーがアセンブラーのプログラミングに習熟していることが前提になっています。これらの例では、スタックはページの下方向へ伸びていきます。また、ESP は常に、スタックの上部を指します。

次の呼び出しでは、a, b, および c は 32 ビットの整数であり、func は 2 つのローカル変数 x および y (両方とも 32 ビットの整数) を持ちます。

```
m = func(a,b,c)
```

FUNC に対する呼び出しのスタックは次のようになります。



スタック上にこの活動化レコードを作成するのに使用される命令は、呼び出し側では次のようになります。

```
PUSH c
PUSH b
PUSH a
CALL _func@12
.
.
MOV m, EAX
.
.
```

呼び出し先では、コードは次のようになります。

```
_func@12 PROC
    PUSH    EBP
    MOV     EBP, ESP           ; Allocating 8 bytes of storage
```

STDCALL リンケージ

```

        SUB     ESP, 8           ; for two local variables.
        PUSH    EDI             ; These would only be
        PUSH    ESI             ; pushed if they were used
        PUSH    EBX             ; in this function.
        .
        MOV     EAX, [EBP - 8]   ; Load y into EAX
        MOV     EBX, [EBP + 12]  ; Load b into EBX
        .
        .
        XOR     EAX, EAX         ; Zero the return value
        POP     EBX             ; Restore the saved registers
        POP     ESI
        POP     EDI
        LEAVE                    ; Equivalent to MOV ESP, EBP
                                   ; POP EBX
        RET     0CH
_func@12 ENDP
```

保管レジスター・セットは EBX、ESI、および EDI です。

構造体はスタック上には戻されません。呼び出し側は、戻される構造体が置かれるアドレスを、最初の隠しパラメーター字句としてプッシュします。構造体に戻す関数は、すべてのパラメーターが、構造体に戻されない場合のパラメーターよりも、EBP から 4 バイト遠くに離れるように注意する必要があります。戻される構造体のアドレスは、EAX に戻されます。

WinMain の使用 (Windows のみ)

プロシージャー・ステートメントで `OPTIONS(WINMAIN)` を指定することにより、WinMain を使用できます (構文については `PL/I 言語解説書` を参照)。これにより、自動的に `LINKAGE(STDCALL)` および `EXT('WinMain')` が指定されたことになります。

WinMain ルーチンには、以下の 4 つのパラメーターが必要です。

- インスタンス・ハンドル
- 前のハンドル
- コマンド行へのポインター
- ShowWindow に受け渡される整数

これらは、C において WinMain が予想する 4 つのパラメーターと同じものです。このルーチン内で行われる呼び出しは、C ルーチンによって予想されるものと同じものです。

実例 `guisamp.pli` はサンプル・ディレクトリーにあります (詳細については `prolog` プログラムを参照)。

CDECL リンケージ

このリンケージ規則を使用するには、関数の宣言において `OPTIONS(LINKAGE (CDECL))` 属性を指定するか、`DEFAULT(LINKAGE(CDECL))` コンパイル時オプションを指定する必要があります。

CDECL の機能

CDECL 呼び出し規則には、以下の規則が適用されます。

- すべてのパラメーターは、スタック上で受け渡される。
- パラメーターは、字句の右から左の順番でスタック上にプッシュされる。
- 呼び出し 関数が、スタックからパラメーターを除去する。
- 浮動小数点値は、ST(0) に戻される。非浮動小数点値を戻すすべての関数は、それらを EAX に戻します。ただし、サイズが 8 バイト以下の集合を戻す特別な場合は除きます。サイズが 4 バイト以下の集合を戻す関数の場合、値は以下のように戻されます。

戻される集合値 のサイズ

8 バイト

EAX-EDX ペア

5、6、7 バイト

EAX 戻り値を置くアドレスは、EAX において隠しパラメーターとして受け渡されます。

4 バイト

EAX

3 バイト

EAX 戻り値を置くアドレスは、EAX に隠しパラメーターとして受け渡されます。

2 バイト

AX

1 バイト

AL

サイズが 5、6、7、または 8 バイトより大きい集合を戻す関数の場合、戻り値を置くアドレスは、隠しパラメーターとして受け渡され、アドレスは EAX に戻されます。

- 関数名は、オブジェクト・モジュールに現れるときは、下線接頭部によって装飾されます。例えば、ソース・プログラムの fred という名前の関数は、オブジェクトでは _fred となります。

.DEF ファイルでエクスポートまたはインポートのリストを作成する場合は、装飾されたバージョンの名前を使用する必要があります。DEF ファイルにおいて装飾されていない名前を使用する場合は、ILIB に対し、DEF ファイルと一緒にオブジェクト・ファイルを提供する必要があります。ILIB は、オブジェクト・ファイルを使用して、それぞれの名前が装飾後にどのようなになるかを判別します。

CDECL 規則の使用例

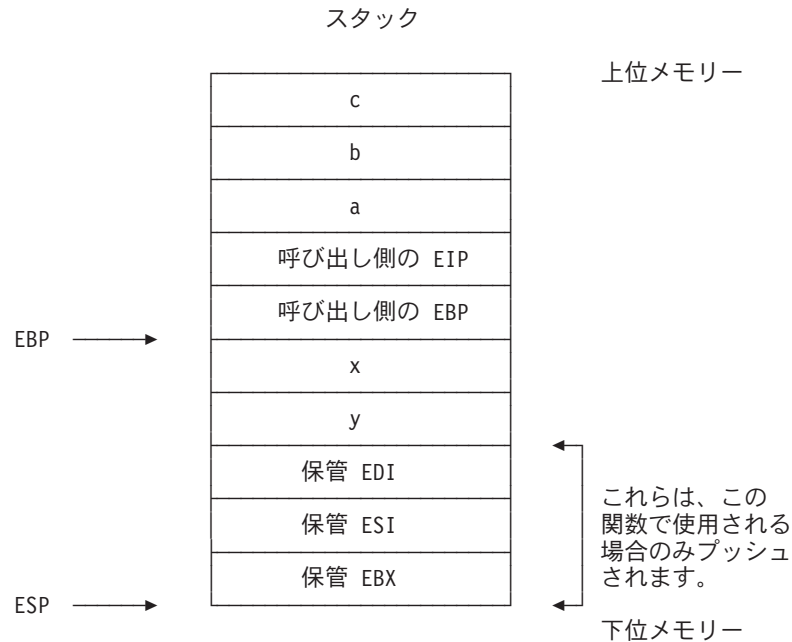
以下の例は、わかりやすく例示するためにのみ記載されたものです。最適化されているわけではありません。これらの例では、ユーザーがアセンブラーのプログラミングに習熟していることが前提になっています。これらの例では、スタックはページの下方向へ伸びていきます。また、ESP は常に、スタックの上部を指します。

次の呼び出しを考えてみます。

```
m = func(a,b,c);
```

変数 a, b, および c は 32 ビットの整数であり、FUNC は 2 つのローカル変数 x および y (両方とも 32 ビットの整数) を持ちます。

FUNC に対する呼び出しのスタックは次のようになります。



スタック上にこの活動化レコードを作成するのに使用される命令は、呼び出し側では次のようになります。

```
PUSH c
PUSH b
PUSH a
CALL _func
:
:
ADD ESP, 12 : cleaning up the parameters
:
:
MOV m, EAX
:
:
```

呼び出し先では、コードは次のようになります。

```
_func PROC
    PUSH    EBP
    MOV     EBP, ESP           ; Allocating 8 bytes of storage
    SUB     ESP, 08H          ; for two local variables.
    PUSH    EDI               ; These would only be
    PUSH    ESI               ; pushed if they were used
    PUSH    EBX               ; in this function.
    :
    :
    MOV     EAX, [EBP - 8]     ; Load y into EAX
    MOV     EBX, [EBP + 12]    ; Load b into EBX
    :
    :
    XOR     EAX, EAX           ; Zero the return value
```

```

        POP     EBX                ; Restore the saved registers
        POP     ESI
        POP     EDI
        LEAVE   ESP               ; Equivalent to  MOV     ESP, EBP
                                   ;                POP     EBP
        RET
_func ENDP

```

保管レジスター・セットは EBX、ESI、および EDI です。構造体が値パラメーターとして受け渡され、構造体のサイズが 5、6、7、または 8 バイトより大きい場合、戻り値を置くアドレスは、隠しパラメーターとして受け渡され、アドレスは EAX に戻されます。

第 25 章 混合言語アプリケーションでの PL/I の使用

データおよびリンケージの一致	405	使用する環境の保守	410
受け渡されるデータの内容	406	PL/I メインからの非 PL/I ルーチンの呼び出し	410
データが受け渡される方法	408	非 PL/I メインからの PL/I ルーチンの呼び出し	411
データが受け渡される場所	409	ON ANYCONDITION の使用	411

ワークステーション環境内で、PL/I を関連言語の 1 つとして使用して、混合言語アプリケーションを開発したい場合があります。例えば、アプリケーションを、C で作成したメインプログラムおよび PL/I で作成したダイナミック・リンク・ライブラリー (DLL) によって構成できます。REXX を使用したアプリケーションを考えることもできます。このアプリケーションは、PL/I DLL にパッケージされた PL/I ルーチンをロードし、呼び出すことができます。

外部ベンダーのソフトウェアを使用して、アプリケーションを構成したい場合もあります。ベンダーのプリパッケージ・プログラムを使用する場合は、PL/I で作成した DLL の形式でユーザー出口を提供できます。

混合言語アプリケーションを作成するのは、通常、難しい作業であり、単一言語でコーディングするときには存在しない多くの要因を考慮する必要があります。通常、さまざまなベンダーの高水準プログラム言語 (例えば、C、C++、COBOL、および PL/I) の場合、特定言語のランタイム・ライブラリーによってインプリメントされた特定のランタイム環境を使用する必要があります。これらの言語が、連携してうまく動作しない領域として次のものがあります。

- データ型のインプリメンテーションおよび使用
- データ・アライメント
- 例外処理機能
- ランタイム環境の初期化と終了
- ユーザー出口ルーチン
- 入出力機能

動作におけるこれらの不整合は、いくつかの混合言語プログラム実行シナリオにおいて、予期しないランタイム動作を引き起こす場合があります。

データおよびリンケージの一致

ルーチンが別のルーチンを正常に呼び出すには、2 つのルーチンは、共用インターフェースに関して一致するビューを持つ必要があります。いずれかのルーチンが PL/I でコーディングされていない場合、これらのインターフェースは、以下のことによって制限されます。

- 受け渡されるデータの内容
- データが受け渡される方法
- データが受け渡される場所

これらについて、以下のセクションで詳しく説明します。混合言語アプリケーションでは、共用インターフェースのビューが一致しないことはよくある問題です。以下の点が重要です。

- 引数およびパラメーターは一致しなければならない。

- 値によって受け取る予定のデータは、値によって受け渡さなければならない。
- 呼び出し先と呼び出し側のルーチンは、同じリンケージを使用する必要がある。

受け渡されるデータの内容

PL/I と C ルーチンは、同等のデータ型のデータを受け渡し、戻すことによって、「通信」します。PL/I と非 PL/I ルーチンは、外部静的変数を使用して通信することはできません。PL/I と C 間で同等のスカラー・データ型を表 31 にリストします。

表 31. C と PL/I 間で同等のデータ型

C データ型	PL/I データ型
signed char	FIXED BIN(7,0)
unsigned char	UNSIGNED FIXED BIN(8,0) または CHAR(1)
signed short	FIXED BIN(15,0)
unsigned short	UNSIGNED FIXED BIN(16,0)
signed (long) int	FIXED BIN(31,0)
unsigned (long) int	UNSIGNED FIXED BIN(31,0)
float	FLOAT BIN(21)FLOAT DEC(6)
double	FLOAT BIN(53)FLOAT DEC(16)
long double	FLOAT BIN(64) FLOAT DEC 18)
enum	ORDINAL
<non-function-type> *	POINTER または HANDLE
<function-type> *	ENTRY LIMITED

表の最後の行に示したように、入出力変数が LIMITED でない場合、C 関数ポインターは PL/I 入出力変数と同等ではありません。この間違いによって引き起こされるエラーは、検出が困難です。

次元数が同じで、下限と上限が同じ場合、同等タイプの配列は同等です。C では、下限を指定することはできません。また、実際の上限は、指定した数より 1 つ少ない数になります。例えば、C で宣言された次の配列を考えてみます。

```
short x [ 6 ];
```

PL/I では、この配列は次のように宣言されます。

```
dcl x(0:5) fixed bin(15);
```

同等タイプの構造体および共用体も、それらのエレメントが同じオフセットにマップされる場合は、同等です。エレメント間に埋め込みがない場合は、オフセットは同じものになります。構造体 (または共用体) のエレメントがすべて UNALIGNED の場合、PL/I は埋め込みを使用しません。ALIGNED のエレメントがある場合は、AGGREGATE リストを調べて、埋め込みがあるかどうかを判別することができます。PL/I はストリングをスカラーとみなしますが、C はそのようにみなしません。したがって、これまでの説明はストリングには適用できません。

以下に示すように、C ビット・フィールドと PL/I ビット・ストリングはあまり似ていません。

- C ビット・フィールドは 32 ビットに制限されるが、PL/I ビット・ストリングは 32767 ビットの長さにすることができる。
- C ビット・フィールドは、必ずしも左から右の順番にマップされるわけではない。Intel C コンパイラーのなかには、次の C 構造体を、PL/I 構造体と同等になるようにマップするものがあります。

C 構造体

```
struct { unsigned byte1 :8;
        unsigned byte2 :8;
        unsigned byte3 :8;
        unsigned byte4 :8;
        } bytes;
```

PL/I 構造体

```
dcl
  1 bytes,
    2 byte1 bit(8),
    2 byte2 bit(8),
    2 byte3 bit(8),
    2 byte4 bit(8);
```

また、他の C コンパイラーのなかには、元の構造体を、次の PL/I 構造体と同等になるように、バイトを逆の順番にしてマップするものもあります。

PL/I 構造体

```
dcl
  1 bytes,
    2 byte4 bit(8),
    2 byte3 bit(8),
    2 byte2 bit(8),
    2 byte1 bit(8);
```

厳密には、C には、文字ストリングはありません。あるのは、文字に対するポインターのみです。ただし、一般的な使用法では、C ストリングは、最後の値が 'X'00' である文字のシーケンスになります。したがって、以下の例では、*address* は、最大 30 個の非ヌル文字を保持できる C 「ストリング」です。

```
char address [ 31 ];
```

次の PL/I 宣言が、C 「ストリング」に一番似ています。

```
dcl address char(30) varyingz;
```

C 関数の宣言では、ストリングは、通常、*char** として宣言されます。例えば、C ライブラリー関数 *strcspn* は、次のように宣言できます。

```
int strcspn( char * string1, char * string2 );
```

同じ関数の PL/I 宣言は次のようになります。

```
dcl strcspn entry( char(*) varyingz,
                  char(*) varyingz )
  returns( fixed bin(31) );
```

上記の例では、C と PL/I の宣言はどちらも不完全です。完全なバージョンについては、この章の最後の方で説明します。

データが受け渡される方法

PL/I および C のいずれも、データのさまざまな受け渡し方法をサポートします。これらの方法を理解するには、以下の用語を知っておく必要があります。

パラメーター

PL/I プロシージャまたは関数の定義で宣言された変数。例えば、次の PL/I 関数定義で、*seed* はパラメーターです。

```
funky:
  proc( seed )
    returns( fixed bin(31) );

    dcl seed fixed bin(31);
    .
    .
    .
  end funky;
```

引数

ルーチンに実際に受け渡される変数または値。関数 *funky* (上記の例) が *rc* = *funky(seed)*; によって呼び出される場合、*seed* は引数です。

By value

引数の値が受け渡されます。呼び出し側ルーチンが *by value* で引数を受け渡す場合、呼び出し先ルーチンは、元の引数を変更できません。

By address

引数のアドレスが受け渡されます。呼び出し側ルーチンが *by address* で引数を受け渡す場合、呼び出し先ルーチンは、呼び出し側の引数を変更できます。

C はすべてのパラメーターを *by value* で受け渡しますが、PL/I (デフォルトで) はパラメーターを *by address* で受け渡します。PL/I は、配列、構造体、共用体、および * として宣言された長さを持つ字符串を除いて、*by value* でのパラメーターの受け渡しもサポートします。

「PL/I 言語解説書」で詳しく説明されているように、BYADDR または BYVALUE 属性で *by address* または *by value* を宣言することによって、パラメーターをそれらのいずれで受け渡すかを指示できます。以下の例では、*modf* への最初のパラメーターは *by value* で受け渡され、2 番目のパラメーターは *by address* で受け渡されます。

```
dcl modf entry( float bin(53) byvalue,
               float bin(53) byaddr )
  returns( float bin(53) );
```

対応する C の宣言は次のとおりです。

```
double modf( double x, double * intptr );
```

宣言で BYADDR または BYVALUE 属性を明示しない場合、そのエントリーのオプション・リストでそれらを指定できます。次の宣言はオプション・リストを使用しており、上記の例と同等なものになっています。

```
dcl modf entry( float bin(53),
               float bin(53) byaddr )
  returns( float bin(53) )
  options( byvalue );
```

パラメーターが `by address` で受け渡される場合でも、その値が、受け取り側ルーチンによって変更されないこともあります。PL/I では、属性 `NONASSIGNABLE` (または `NONASGN`) をそのパラメーターの宣言に追加することによって、このことを指示することができます。次の部分的宣言は、関数 `strcspn` へのいずれの引数も、その関数によって変更されないことを指示します。

```

dcl strcspn entry( nonasgn char(*) varyingz,
                  nonasgn char(*) varyingz )
                  returns( fixed bin(31) );

```

対応する C の宣言は次のとおりです。

```

int strcspn( const char * string1, const char * string2 );

```

ルーチンは、それを呼び出すすべてのルーチンとの間で、データの受け渡しについて同意する必要があります。このような種類のミスマッチを検出するための十分な情報をコンパイラーに提供することによって、問題を回避することができます。例えば、次の宣言は、技術的には、上記のサンプル・コードにおける `modf` の宣言と同等ですが、この宣言を使用すると、すべての引数のアドレスを 2 番目の引数として受け渡すことができます。以前の宣言では、2 番目の引数は正しいタイプを持つ必要があります。

```

dcl modf entry( float bin(53),
               pointer )
               returns( float bin(53) )
               options( byvalue );

```

最後に、PL/I は、いくつかのデータ型 (ストリング、配列、構造体、および共用体) を受け渡す場合、デフォルトで、データ・エクステント (最大のストリングの長さ、配列境界など) を記述する記述子 も受け渡します。C ルーチンは PL/I 記述子を使用できないため、C と PL/I のルーチン間で記述子が受け渡されないようにする必要があります。C エントリーの宣言で `OPTIONS` 属性に `NODESCRIPTOR` オプションを追加することによって、これを行うことができます。例えば、次のようにします。

```

dcl strcspn entry( nonasgn byaddr char(*) varyingz,
                  nonasgn byaddr char(*) varyingz )
                  returns( fixed bin(31) )
                  options( nodestructor );

```

データが受け渡される場所

相互作用するルーチンにとって、受け渡すデータの内容およびデータを受け渡す場所について同意することは、データを受け渡す方法について同意するのと同じように重要です。PL/I および C のいずれの場合も、データを、スタック上、汎用レジスター内、または浮動小数点レジスター内で受け渡すことができます。

PL/I では、`LINKAGE` オプション (プロシーチャー・ステートメントとエントリー宣言の `OPTIONS` オプションでの) によって、データが受け渡される場所が決定します。データ位置のエラーが発生する一般的な場合の 1 つに、ミスマッチのリンケージ・タイプを指定する (または、デフォルトが正しくない場合にリンケージ・タイプを指定しない) 場合があります。

PL/I for Windows は、3 つの 32 ビット・リンケージ・タイプ (`OPTLINK`、`CDECL`、および `STDCALL`) をサポートします。次の PL/I 宣言は、関数 `dosSleep` が `SYSTEM` リンケージを使用することを指示します。

```
dc1 dosSleep entry( fixed bin(31) byvalue )
               returns( fixed bin(31) )
               options( linkage(system) );
```

オプション・リストは、呼び出すすべての C ルーチンによって使用されるリンケージを指定する必要があります。 PL/I および VisualAge for C++ のいずれのコンパイラーも、OPTLINK をデフォルト・リンケージとして使用します。 Windows 上の多くの C ルーチンは STDCALL リンケージを使用します。これらのルーチンの場合、OPTIONS 属性で LINKAGE(STDCALL) を指定する必要があります。例えば、DosSleep の Windows での同等の宣言は以下のようになります。

```
dc1 Sleep      ext('Sleep')
               entry( fixed bin(31) byvalue )
               returns( fixed bin(31) )
               options( linkage(stdcall) );
```

使用する環境の保守

PL/I (および多くのその他の言語) が正常に動作するには、それらの言語が確立しているランタイム環境を損なわないようにする必要があります。言語間呼び出しが関連する場合、このことは次のことを意味します。

- 例外ハンドラーを登録するすべてのルーチンは、PL/I に戻る前にそのハンドラーの登録を解除する必要がある。
- ブロック外への GOTO は、ソースとターゲットのブロックが同じ言語でコード化されており、すべての介入ブロックが同じ言語でコード化されている場合にのみ許可される。

PL/I メインからの非 PL/I ルーチンの呼び出し

メインルーチンが PL/I でコード化されている場合、2 種類の非 PL/I ルーチンを呼び出すことができます。

- システム・ルーチン (DOS および Windows サービスなど)
- C、COBOL、REXX ルーチン

システム・ルーチンは、独自のランタイム環境は必要としません。また、それらは PL/I 実行可能 (.EXE) ファイルまたはダイナミック・リンク・ライブラリー (.DLL) に直接リンクすることができます。 IBM VisualAge C/C++ ルーチンを除いて、その他のすべての非 PL/I ルーチンは、.EXE または .DLL に直接リンクすることはできません。その代わり、それらは、.DLL がロードされたときに、それらに必要なすべてのランタイム環境の初期化を行えるように、.DLL にリンクする必要があります。

IBM VisualAge C/C++ ルーチンは、PL/I とリンクできます。ただし、C ルーチンが PL/I とリンクされており、それらのいずれかが C ライブラリー関数を使用する (または C ライブラリー関数自身である) 場合は、ルーチンを呼び出す前に C ランタイムを初期化する必要があります。C ランタイムは、次のルーチンを呼び出すことで初期化できます。

```
dc1 _CRT_init  ext('_CRT_init')
               entry()
               returns( optional fixed bin(31) )
               options( linkage(optlink) );
```

また、C ランタイムが、オープンしたすべてのファイルをクローズし、獲得した可能性のあるその他すべてのシステム・リソースを戻すようにするには、以下を呼び出すことにより、C ランタイムを終了する必要があります。

```
dc1 _CRT_term  ext('_CRT_term')
               entry()
               returns( optional fixed bin(31) )
               options( linkage(optlink) );
```

非 PL/I メインからの PL/I ルーチンの呼び出し

PL/I ランタイム環境には、以下の機能があります。

- PL/I DLL が非 PL/I メインプログラムから動的にロードされるときに、自分で初期化する。
- 非 PL/I 言語ランタイム環境と、最小の競合で共存する。

非 PL/I ルーチンから直接呼び出される PL/I ルーチンは、OPTIONS オプションで FROMALIEN オプションを持つ必要があり、MAIN オプションを指定することはありません。

非 PL/I ルーチンから呼び出される PL/I ルーチンは、PL/I コードで発生するすべての例外を処理し、最初の PL/I プロシージャで RETURN または END ステートメントを使用して、非 PL/I に戻る必要があります (『ON ANYCONDITION の使用』を参照)。

PL/I ランタイムは、暗黙的に PL/I が獲得したすべてのリソースを解放します。ただし、これが行われるのは、アプリケーションが終了してからです。

また、以下のようなさまざまな PL/I ステートメントを使用して、明示的にリソースを解放できます。

- RELEASE * - すべてのフェッチされたモジュールを解放する
- FLUSH FILE(*) - すべてのファイル・バッファをフラッシュする
- CLOSE FILE(*) - すべてのオープンしたファイルをクローズする

ON ANYCONDITION の使用

すべてのアプリケーションは、その内部で発生するすべての例外を処理し、「通常」制御を、呼び出し側プログラムに戻すことができる必要があります。PL/I 例外処理機能および ANYCONDITION ON ユニットを使用すると、これが可能になります。

非 PL/I ルーチンから呼び出される PL/I ルーチンにおける最初の実行可能ステートメントは、ON ANYCONDITION ステートメントでなければなりません。このステートメントには、他の ON ユニットによって明示的に処理されないすべての状態を処理するコードが含まれている必要があります。処理できない状態が発生する場合は、ルーチンで正常に実行される最後のステートメントを指し示す GOTO ステートメントを使用します。例えば、次のようになります。

```
pliapp:
  proc( p1, ..., pn )
  returns( ... )
  options( fromalien );
```

PL/I ルーチンの呼び出し

```
/* declarations of paramaters, if any */  
/* declarations of other variables */  
  
on anycondition  
begin;  
    /* handle condition if possible */  
  
    /* if unhandled, set return value */  
    goto return_stmt;  
end;  
  
/* mainline code */  
  
return_stmt:  
return( ... );  
  
end_stmt:  
end pliapp;
```

関数でない PL/I ルーチンの場合、GOTO のターゲットは、ルーチンの END ステートメントでなければなりません。

第 26 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。

PL/I から Java とのやり取りを行うには、事前にシステムに Java をインストールしておく必要があります。最新の Java Development Kit (JDK) の無料バージョンをダウンロードする場所は多くあります。

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すでに多数存在し、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、この利点を生かしたい。

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・フレームワークにすで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」

最初に作成するサンプル・プログラムは、よくある「Hello World!」プログラムの一種です。

本書の「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。この例のネイティブ・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、ダイナミック・リンク・ライブラリー (DLL) 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 ダイナミック・リンク・ライブラリー *hiFromPLI.dll* が、PL/I プログラムをコンパイルし、リンクするステップの間に作成されます。

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( 31 ) ) Margins( 1, 100 );
*Process Dllinit xinfo(def);
*Process Default( 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(31)

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。

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。

xinfo(def)

DLL の作成で使用する *.DEF ファイルを作成するように、コンパイラーに指示します。

Default(IEEE);

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。

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・プログラムにある外部 PL/I プロシージャ名

`_Java_callingPLI_callToPLI` を分けて説明します。

_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。

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( NoDescriptor ByVal linkage(stdcall) );
```

JNI インクルード・ファイル

Java インターフェースの PL/I 定義を含む PL/I インクルード・ファイルは、次の 2 つのインクルード・ファイルに含まれています。2 つのファイルは、*jni.cop* の中に *jni_md.cop* が含まれる構造になっています。これらのインクルード・ファイルは、次のステートメントによって組み込まれます。

```
%include jni;
```

jni.cop ファイルの完全なリストについては、*%bimpliw%include* ディレクトリーを参照してください。

完全な PL/I プロシージャー

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( 31 ) ) Margins( 1, 100 ) ;
*Process Dllinit xinfo(def);
*Process Default( IEEE );
PliJava_Demo: Package Exports(*);
```

```
Java_callingPLI_callToPLI:
Proc( JNIEnv , MyJObject )
  External( "Java_callingPLI_callToPLI" )
  Options( NoDescriptor ByValue linkage(stdcall) );
```

```
%include jni;
```

```
Display('Hello from PL/I for Windows!');
```

```
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 hiFromPLI.pli
```

ダイナミック・リンク・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 DLL にリンクします。

```
ilib /nologo /geni hiFromPLI.def
ilink /dll hiFromPLI.obj hiFromPLI.exp java%lib%javai.lib
```

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 PL/I for Windows!
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

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、419 ページの図 29 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* に入っています。最初のサンプル・プログラムで説明したことの多くが、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、419 ページの図 29 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 29. 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

このプログラムの外部 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 , myobject )
external( "_Java_jPassString_pliShowString" )
options( byvalue nodestructor linkage(stdcall) );
```

JNI インクルード・ファイル

Java インターフェースの PL/I 定義を含む PL/I インクルード・ファイルは、次の 2 つのインクルード・ファイルに含まれています。2 つのファイルは、`jni.cop` の中に `jni_md.cop` が含まれる構造になっています。これらのインクルード・ファイルは、次のステートメントによって組み込まれます。

```
%include jni;
```

`jni.cop` ファイルの完全なリストについては、`%bimpliw%include` ディレクトリーを参照してください。

完全な PL/I プロシージャ

完全な PL/I プログラムは、421 ページの図 30 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

ストリングに変換されます。この新しい Java ストリングを使用して、*SetObjectField* JNI 関数によって呼び出し側 Java オブジェクト内のストリング・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 ストリングが Java プログラムによって表示されます。

```
*Process Limits( Extname( 31 ) ) Margins( 1, 100 ) ;
*Process Dllinit xinfo(def);
*Process Default( IEEE );
plijava_demo: package exports(*);

Java_passString_pliShowString:
Proc( JNIEnv , myJObject )
    external( "_Java_jPassString_pliShowString" )
    options( byvalue nodestructor linkage(stdcall) );

%include jni;

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 30. PL/I サンプル・プログラム #2 - ストリングの引き渡し

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 passString.pli
```

ダイナミック・リンク・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 DLL にリンクします。

```
ilib /nologo /geni passString.def  
ilink /dll passString.obj passString.exp javalib%javai.lib
```

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

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、423 ページの図 31 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の多くが、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、図 31 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 31. 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

このプログラムの外部 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 , myobject )
  external( "_Java_jPassInt_pliShowInt" )
  options( byvalue nodestructor linkage(stdcall) );
```

JNI インクルード・ファイル

Java インターフェースの PL/I 定義を含む PL/I インクルード・ファイルは、次の 2 つのインクルード・ファイルに含まれています。2 つのファイルは、`jni.cop` の中に `jni_md.cop` が含まれる構造になっています。これらのインクルード・ファイルは、次のステートメントによって組み込まれます。

```
%include jni;
```

`jni.cop` ファイルの完全なリストについては、`%bimpliw%include` ディレクトリを参照してください。

完全な PL/I プロシージャ

完全な PL/I プログラムは、425 ページの図 32 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( 31 ) ) Margins( 1, 100 ) ;
*Process Dllinit xinfo(def);
*Process Default( IEEE );
plijava_demo: package exports(*);

Java_passNum_pliShowInt:
Proc( JNIEnv , myjobject )
  external( "_Java_jPassInt_pliShowInt" )
  options( byvalue nodestructor linkage(stdcall) );

%include jni;

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 32. 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 passInt.pli
```

ダイナミック・リンク・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 DLL にリンクします。

```
ilib /nologo /geni passInt.def  
ilink /dll passInt.obj passInt.exp java1ib%javai.lib
```

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  
>
```

Java および PL/I の同等なデータ型の判別

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型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 27 章 ソート・ルーチンの使用

S/390 とワークステーションのソート・プログラムの比較	427	例 3	434
ソート・プログラムの使用準備	429	例 4	434
ソート・タイプの選択	429	ソートが成功したかどうかの判別	435
ソート・フィールドの指定	432	ソート・データの入出力	435
例:	433	ソート・データの処理ルーチン	436
ソートするレコードの指定	433	E15 - 入力処理ルーチン (ソート出口 E15)	436
例:	433	E35 - 出力処理ルーチン (ソート出口 E35)	438
ソート・プログラムの呼び出し	434	PLISRTA の呼び出し	441
PLISRT の例	434	PLISRTB の呼び出し	442
例 1	434	PLISRTC の呼び出し	444
例 2	434	PLISRTD の呼び出し、例 1	445
		PLISRTD の呼び出し、例 2	446

PL/I for Windows は、PLISRT x ($x = A, B, C$, または D) 組み込みサブルーチンをサポートします。PLISRT x サブルーチンを使用するには、以下の作業を行う必要があります。

- いずれかのサブルーチン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呼び出しに渡します。この情報には、レコード長、戻りコードとして使用する変数の名前、およびソートを行うのに必要なその他の情報が含まれます。
- 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。

これらのサブルーチンは、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プログラマー作成の PL/I プロシージャである場合とがあります。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。

S/390 とワークステーションのソート・プログラムの比較

既存のメインフレーム・プログラムに CALL PLISRT x が含まれている場合は、それらをダウンロードし、ワークステーション上で実行できます。S/390 で許可されたいくつかのパラメーターは無視され、ある程度、ランタイム動作を変更します。次の表は、OS PL/I によって受け取られたどの引数が、ワークステーション・コンパイラによって無視されるかを示しています。

表 33. ワークステーション PLISRT x

組み込みサブルーチン	引数
PLISRTA	(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
ソート入力: データ・セット	[, データ・セット接頭部, メッセージ・レベル,
ソート出力: データ・セット	ソート手法])

ソート・プログラムの比較

表 33. ワークステーション *PLISRTx* (続き)

組み込みサブルーチン	引数
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])

引数定義:

Sort ステートメント

ソート・フィールドとフォーマットを記述する文字ストリング式。432 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

データの長さとレコード・フォーマットを記述する文字ストリング式。433 ページの『ソートするレコードの指定』を参照。

ストレージ

ワークステーション PL/I によって無視される。

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

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 15 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。ワークステーション PL/I を使用する特定要件については、436 ページの『E15 - 入力処理ルーチン (ソート出口 E15)』を参照してください。

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 35 から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。ワークステーション PL/I を使用する特定要件については、438 ページの『E35 - 出力処理ルーチン (ソート出口 E35)』を参照してください。

データ・セット接頭部

ワークステーション PL/I に無視される。SORTIN および SORTOUT のみを DD 名として処理する。

メッセージ・レベル

ワークステーション PL/I によって無視される。

ソート手法

ワークステーション PL/I によって無視される。

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、およびデータ・レコード長を決める必要があります。

使用する PLISRTx 組み込みサブルーチンを決定するには、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。入力または出力処理のサブルーチンを使用する場合は、436 ページの『ソート・データの処理ルーチン』を参照してください。

ソート組み込みサブルーチン、およびデータのソースおよび宛先は以下のとおりです。

組み込みサブルーチン	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

ソース・データ・セットは SORTIN 環境変数を使用して定義されます。また、宛先データ・セットは SORTOUT を使用して定義されます。あるいは、PUTENV 組み込み関数を使用してそれらの関数を設定することもできます。

使用するサブルーチンを決定したら、以下のように、データ・セットに関するいくつかの事項を決定し、SORT ステートメントに関する情報を指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。

次に、ソートするレコードに関して次の 2 点を決定し、RECORD ステートメントに関する情報を指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 (可変フォーマットの最大長)

PLISRTx への 2 番目の引数である RECORD ステートメント上でこれらを使用します。

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理解する必要があります。PL/I プログラム内で、組み込みサブルーチン 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。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、ソートが成功したか失敗したかを示す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ーの名前です。

ソート・インターフェース・ルーチンは、ソート用の引数リストを、PLISRTx 引数リストが提供する情報から作成します。また、このルーチンは、x として A、B、C、または D のいずれを選択したかによって異なります。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。

ソート操作は、以下の 2 つのいずれかの方法で終了します。

1. 0 から 16 の戻りコードを PL/I 呼び出しプロシージャーに送信して、成功または失敗を伝達する。
2. 特定のエラーが検出され、戻りコードが未定義の場合、エラー条件を発生する。

431 ページの図 33 は、ソート操作を示す単純化されたフローチャートです。

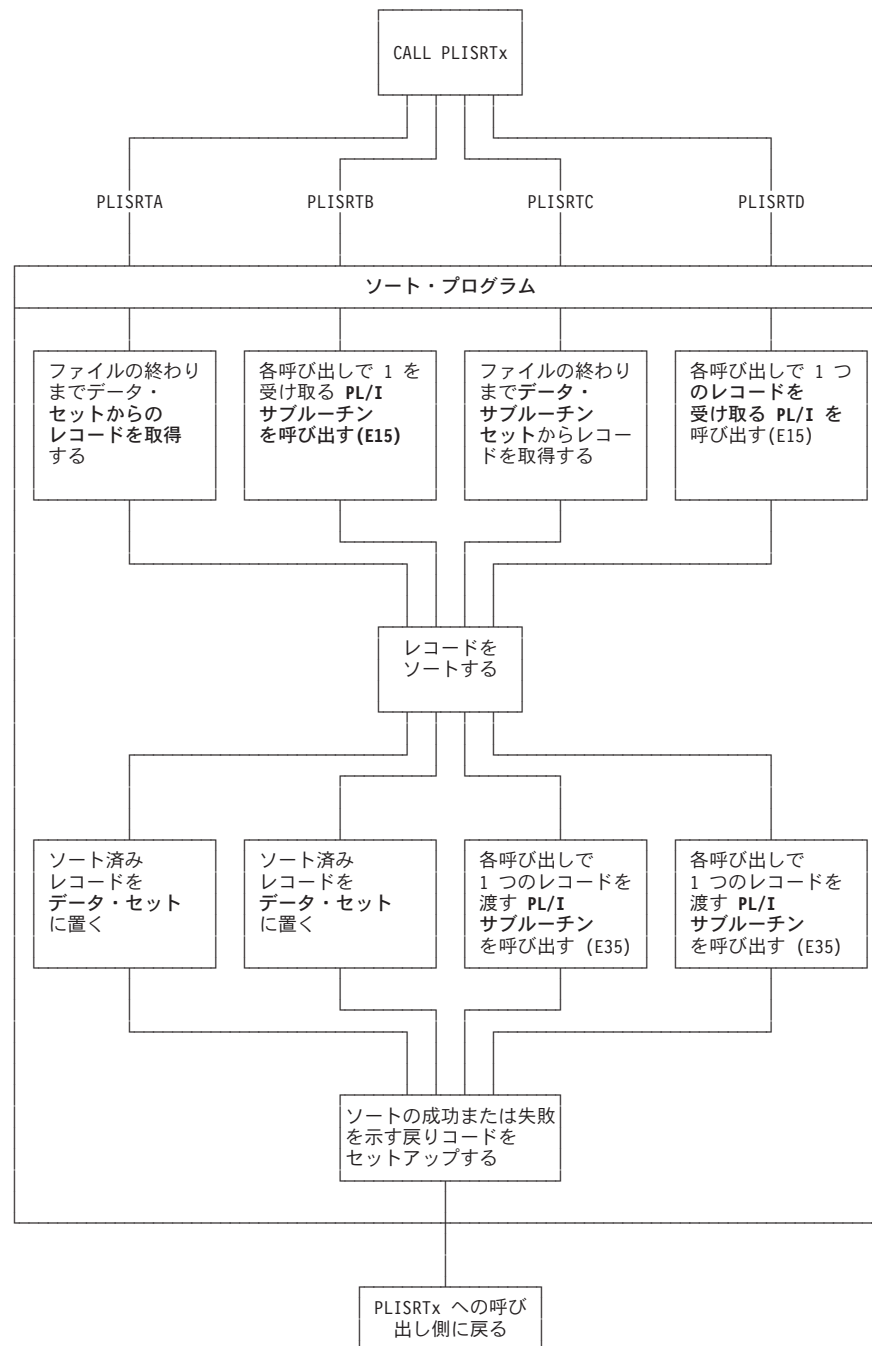


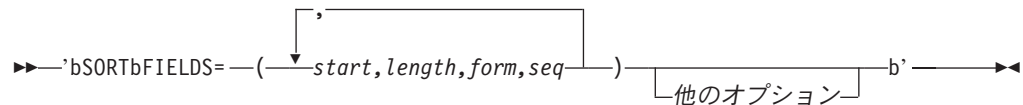
図 33. ソート・プログラムの制御の流れ

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・プログラム内では、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。

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、4 つの組み込みサブルーチンの使用法を示す一連の例があります。

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。



- b** 1 つ以上の空白。ここに示されている空白は必須です。これ以外の空白は認められません。

start,length,form,seq

ソート・フィールド。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、等しい値をもつレコードが 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、等しいレコードの順序は任意になります。ソート・フィールドのオーバーレイはサポートされません。

start

レコード内の開始位置です。バイトで値を指定します。ストリングの最初のバイトは、バイト 1 とみなされます。

length

ソート・フィールドの長さです。バイトで値を指定します。ソート・フィールドの長さには、それぞれのデータ型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。主なデータ型とその長さに対する制約事項を次に示します。

コード データ型と長さ

CH	文字	1 から 256
ZD	ゾーン	10 進数符号付き 1 から 32
PD	パック	10 進数符号付き 1 から 32
FI	固定小数点、符号付き	1 から 256
BI	2 進数、符号なし	1 ビットから 256 バイト

全フィールドの合計長は、256 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)

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

ワークステーション PL/I でサポートされるオプションは、デフォルトの EQUALS のみです。ただし、メインフレームからダウンロードしたソース・コードは、変更する必要はありません。

例:

```
' SORT FIELDS=(1,10,CH,A) '
```

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、評価時に次の構文を受け入れる文字ストリング式でなければなりません。

```
►► 'bRECORDbTYPE=rectype' 'b'
    |,LENGTH=(n)|
```

- b** 1 つ以上のブランク。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

ソート済みデータと未ソート・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・タイプを指定する必要があります。

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・セットから取られるので、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は、32,767 バイトです。可変長のレコードの場合は、2 バイトの接頭部を含める必要があります。

n ソートされるレコードの長さです。

注: 使用できる追加の長さ指定は、ワークステーション PL/I によって無視されます。

例:

```
' RECORD TYPE=F,length=(80) '
```

ソート・プログラムの呼び出し

ソート・フィールドおよびレコード・タイプの指定が決まれば、次に、CALL PLISRTx ステートメントを作成します。

PLISRT の例

以下の例は、PLISRTx に対する呼び出しの一般的に使用される形式を示しています。

例 1

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、および FIXED BINARY (31,0) として宣言された戻りコード RETCODE。

```
call plisrta (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              0,
              retcode);
```

例 2

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。いずれのフィールドも、昇順でソートされます。

```
call plisrta (' SORT FIELD =(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              0,
              retcode);
```

例 3

PLISRTB に対する呼び出し。PL/I ルーチン PUTIN によって入力が入力がソート・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 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) ',
              0,
              retcode,
              putin);
```

例 4

PLISRTD に対する呼び出し。PL/I ルーチン PUTIN によって入力提供され、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 82 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。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=(1,5,CH,A,6,5,CH,D),EQUALS ',
              ' RECORD TYPE=V,LENGTH=(82) ',
              0,
              retcode,
              putin,      /* input routine (sort exit 15) */
              putout);   /* output routine (sort exit 35) */
```

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  Sort successful
16 Sort failed
```

この変数は、FIXED BINARY (31,0) として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;
```

エラーが検出されると、エラー条件が発生します。ソート・プログラムが致命的エラーを検出し、対応するエラー・コードが 16 より大きい場合は、エラー条件が発生します。

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。次の例は、ソート・プログラムから戻された値を使用して、PLIRETC を呼び出す方法を示しています。

```
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力 (E15) および出力 (E35) 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・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 441 ページの図 37 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。

可変長レコードをソートするには、まず、データ・セットを TYPE(VARLS) フォーマットに変換し、次にこの TYPE(VARLS) ファイルをソート・プログラムへの入力として使用する必要があります。TYPE(VARLS) レコードには、先頭に 2 バイトの長さフィールドがあるため、レコード・サイズはそのレコードの長さより実際に

ソート・データの入出力

は 2 バイト小さくなります。つまり、指定するレコード・サイズは、ファイルの最大レコード長より 2 バイト小さくする必要があります。

既存データ・ファイルから読み取り、TYPE(VARLS) として宣言された出力ファイルに書き込む PL/I プログラムを作成することにより、データ・セットを TYPE(VARLS) ファイルに変換できます。

ソート・データの処理ルーチン

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 1 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 - 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。例えば、データを印刷したり (442 ページの図 38 および 445 ページの図 40 を参照)、ソート・フィールドを生成したり、操作したりして、正しい結果を得るのに使用されます。

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・プログラムは、レコードを必要とする場合、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、および渡されたレコードがソートに含まれることを意味する戻りコード 12 を呼び出します。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を呼び出し続けます。つまり、すべてのレコードがすでに渡されており、ソート・プログラムは、ルーチンを再び呼び出さないことを意味します。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。

注: PLISRTB または PLISRTD を呼び出すプログラムは、E15 処理ルーチンのコンパイルに使用したのと同じオプション (ASCII または EBCDIC、NATIVE または NONNATIVE、HEXADEC または IEEE) を指定してコンパイルする必要があります。

E15 ルーチンによって戻されるデータは、固定または可変の文字ストリングでなければなりません。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

する必要があります。しかし、F を指定することも可能です。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。入力ルーチンの例は、442 ページの図 38 および 445 ページの図 40 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、ソート・プログラムの戻りコード 16 (ソート失敗) を PL/I プログラムに設定します。

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てください。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、444 ページの図 39 に示してあります。

```
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 is normally
                               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.      */
        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 of the input procedure      */
```

図 34. 入力プロシージャー用の骨組みコード

さらに、入力ユーザー出口ルーチンをコーディングするには、E15 がプログラム単位にネストされていない場合、PLISRTx を呼び出すプログラム単位に E15 の明示

的属性を指定する必要があります。

```
plisort: proc options(main);

    dcl e15 entry returns(char(2000) varying);

    /* Code to do your processing goes here */

    call plisrtb(' SORT FIELDS=(5,10,CH,A) '
                ' RECORD TYPE=V,LENGTH=(2000) ',
                0,
                retcode,
                e15);

    /* Code to do your processing goes here */

end plisort;

*PROCESS
E15: proc returns (char(2000) varying);
    /* Returns option must be used specifying
       length of data to be sorted, maximum length
       if varying strings are passed to sort. */

    dcl string char(2000) varying;
    /* A character string variable is normally
       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. */

    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 */

        /* 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 35. E15 が、PLISRTx を呼び出すプロシージャーの外部にある場合

E35 - 出力処理ルーチン (ソート出口 E35)

PLISRTC または PLISRTD を呼び出すプログラムは、E35 処理ルーチンのコンパイラに使用したのと同じオプション (ASCII または EBCDIC、NATIVE または NONNATIVE) を指定してコンパイルする必要があります。

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、444 ページの図 39 および 445 ページの図 40 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、それらを 1 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、レコード長をも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(*);
```

典型的な出力処理ルーチンの骨組みコードは、440 ページの図 36 に示してあります。

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てください。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。

```
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 36. 出力処理プロシージャ用の骨組みコード

PLISRTA の呼び出し

```

/*****
/*
/* DESCRIPTION
/*   Sorting from an input data set to an output data set
/*
/* Use the following statements:
/*   set dd:sortin=ex106.dat,type(crlf),lrecl(80)
/*   set dd:sortout=ex106.out,type(crlf),lrecl(80)
/*
/*
/*
/*****/

ex106: proc options(main);
      dcl Return_code fixed bin(31,0);

      call plisrta (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   0,
                   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);
        other   put skip edit (
          'Invalid sort return_code = ', Return_code) (a,f(2));
      end /* Select */;
      /* Set pl/i return code to reflect success of sort */
      call pliretc(Return_code);
end ex106;

```

図 37. PLISRTA-入力データ・セットから出力データ・セットへのソート

図 37 で使用される EX106.DAT の内容

```

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

```

PLISRTB の呼び出し

```

/*****
/*
/* DESCRIPTION
/*   Sorting from an input-handling routine to an output data set
/*
/*
/* Use the following statements:
/*
/*
/*   set dd:sysin=ex107.dat,type(crlf),lrecl(80)
/*   set dd:sortout=ex107.out,type(crlf),lrecl(80)
/*
/*
/*****
ex107:  proc options(main);

        dcl Return_code fixed bin(31,0);

        call plisrtb (' SORT FIELDS=(7,74,CH,A) ',
                     ' RECORD TYPE=F,LENGTH=(80) ',
                     0,
                     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);
          other   put skip edit
                    ('Invalid return_code = ',Return_code)(a,f(2));
        end /* Select */;
        /* Set pl/i return code to reflect success of sort */
        call pliretc(Return_code);

e15x:   /* Input-handling routine gets records from the input
         stream and puts them before they are sorted */
        proc returns (char(80));
          dcl sysin file stream 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 */
            goto ende15;
          end;

          get file (sysin) edit (infield) (1);
          put skip edit (infield)(a(80)); /* Print input */
          call pliretc(12); /* Request sort to include current
                             record and return for more */
          return(Infield);
        ende15:
        end e15x;
      end ex107;

```

図 38. PLISRTB-入力処理ルーチンから出力データ・セットへのソート

442 ページの図 38 で使用される EX107.DAT の内容

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

PLISRTC の呼び出し

```

/*****
/*
/* DESCRIPTION
/*   Sorting from an input data set to an output-handling routine */
/*
/* Use the following statement:
/*
/*   set dd:sortin=ex108.dat,type(crlf),lrecl(80)
/*
/*
*****/

ex108:  proc options(main);

        dcl Return_code fixed bin(31,0);

        call plisrtc (' SORT FIELDS=(7,74,CH,A) ',
                     ' RECORD TYPE=F,LENGTH=(80) ',
                     0,
                     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);
          other   put skip edit
                    ('Invalid return_code = ', Return_code) (a,f(2));
        end /* Select */;
        /* Set pl/i return code to reflect success of sort      */
        call pliretc (return_code);

e35x:   /* Output-handling routine prints sorted records      */
        proc (Inrec);
          dcl inrec char(*);
          put skip edit (inrec) (a);
          call pliretc(4); /* Request next record from sort    */
        end e35x;
end ex108;

```

図 39. PLISRTC-入力データ・セットから出力処理ルーチンへのソート

図 39 で使用される EX108.DAT の内容

```

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

```

PLISRTD の呼び出し、例 1

```

/*****
/*
/* DESCRIPTION
/*   Sorting an input-handling to output-handling routine
/*
/* Use the following statement:
/*
/*   set dd:sysin=ex109.dat,type(crlf),lrecl(80)
/*
/*
*****/

ex109: proc options(main);
      dcl Return_code fixed bin(31,0);
      call plisrtd (' SORT FIELDS=(7,74,CH,A) ',
                   ' RECORD TYPE=F,LENGTH=(80) ',
                   0,
                   Return_code,
                   e15x,
                   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);
        other put skip edit
          ('Invalid return_code = ', Return_code) (a,f(2));
      end /* select */;

      /* Set pl/i return code to reflect success of sort */
      call pliretc(Return_code);

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 */
          goto ende15;
        end;

        get file (sysin) edit (infield) (1);
        put skip edit (infield)(a);
        call pliretc(12); /* Input to sort continues */
        return(infield);
      ende15:
        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 40. PLISRTD-入力処理ルーチンから出力処理ルーチンへのソート

図 40 および 446 ページの図 41 で使用される EX109.DAT および EX110.DAT の内容

```

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

```

PLISRTD の呼び出し、例 2

```

ex110: proc options(main);

    /******
    /*
    /* PLISRTD: sorting from an input-handling rtn to an
    /* output-handling routine. Records are varying-length. */
    /*
    /******

    dcl rc fixed bin(31,0);

    call plisrtd(' SORT FIELDS=(7,4,CH,A) ',
                ' RECORD TYPE=V,LENGTH=(80) ',
                256000,
                rc,
                e15x,
                e35x );

    select( rc );
    when(0) put skip edit
        ('Sort complete return code = 0') (a);
    when(16) put skip edit
        ('Sort failed return code = 16') (a);
    other put skip edit
        ('Invalid return code = ', rc) (a,f(2));
    end;

    call pliretc(rc);

    e15x: proc returns( char(80) varying );

        dcl infield char(80) var;

        on endfile(sysin) begin;
            put skip(3) edit('End of sort program input. ',
                            'Sortout output should follow') (a);
            call pliretc(8);
            goto ende15;
        end;

        get file(sysin) edit(infield) (1);
        put skip edit( infield ) (a);
        call pliretc(12);

        return(infield);
    ende15:
    end e15x;

    e35x: proc ( inrec );

        dcl inrec char(*);

        put skip edit(inrec) (a);
        call pliretc(4);

    end e35x;
end ex110;

```

図 41. PLISRTD-入力処理ルーチンから出力処理ルーチンへのソート

第 28 章 SAX パーサーの使用

コンパイラーは、PLISAXx (x = A または B) というインターフェースを提供しています。このインターフェースは、PL/I に基本的な XML 機能を提供します。このサポートには高速 XML パーサーが含まれています。このパーサーにより、インバウンド XML メッセージを処理したり、適格性を検査したり、メッセージの内容を PL/I データ構造に変換したりするプログラムを作成できます。

XMLCHAR 組み込み関数は、XML 生成をサポートします。

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 文書は適格です。さらに、適格な 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。

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。

\gg -PLISAXA($e,p,x,n_{\underbrace{\quad}_{c}}$) \ll

- | | |
|----------|-------------------------------|
| 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($e, p, x_{\lfloor \cdot, c \rfloor}$) ◀◀

- 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

パッチのもとでは、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 では、入力ファイルを指定する文字ストリングは 'file://filename' というフォームになります。ここで、filename は z/OS UNIX ファイルの名前です。

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。

次に示す各イベントの説明は、図 42 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'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 42. 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」エレメント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、100,001 から 165,535、または 200,001 から 265,535 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、および 200,001 から 265,535 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 );

define alias event_end_of_document
  limited entry( pointer )
  returns( byvalue fixed bin(31) )
  options( byvalue );

define alias event_predefined_ref
  limited entry( pointer, char(1) )
  returns( byvalue fixed bin(31) )
  options( byvalue nodestructor );

define alias event_character_ref
  limited entry( pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue );

define alias event_pi
  limited entry( pointer, pointer, fixed bin(31),
                pointer, fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue );

define alias event_exception
  limited entry( pointer, pointer, fixed bin(31),
                fixed bin(31) )
  returns( byvalue fixed bin(31) )
  options( byvalue );

```

図 43. PLISAXA のコーディング例 - 型宣言

```

saxtest: proc options( main );

dcl
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 44. 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'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),
              addrdata(xmlDocument),
              length(xmlDocument) );

end;

```

図 45. 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 );

    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 );

    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 );

    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 46. 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 );

    decl userToken      pointer;
    decl xmlToken       pointer;
    decl 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 );

    decl userToken      pointer;
    decl xmlToken       pointer;
    decl 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 );

    decl userToken      pointer;

    put skip list( lowercase( procname() ) );

    return(0);
  end;

```

図 46. 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 );

    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 );

    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 );

    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 46. 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 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 );

    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 );

    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 46. 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 );

    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_CDATA:
  proc( userToken, xmlToken, TokenLength )
    returns( byvalue fixed bin(31) )
    options( byvalue );

    dc1 userToken      pointer;
    dc1 xmlToken        pointer;
    dc1 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 );

    dc1 userToken      pointer;
    dc1 xmlToken        pointer;
    dc1 tokenLength     fixed bin(31);

    put skip list( lowercase( procname() )
      || ' <' || substr(xmlToken->chars,1,tokenlength) || '>' );

    return(0);
  end;

```

図 46. 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 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 );

    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 );

    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 46. 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 );

    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 );

    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 );

    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 46. 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 );

    decl userToken      pointer;
    decl xmlToken       pointer;
    decl 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 );

    decl userToken      pointer;
    decl xmlToken       pointer;
    decl 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 );

    decl userToken      pointer;
    decl xmlToken       pointer;
    decl currentOffset  fixed bin(31);
    decl errorID        fixed bin(31);

    put skip list( lowercase( procname() )
      || ' errorid =' || errorid );

    return(0);
  end;
end;

```

図 46. 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 47. 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 34. 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な文字を検出した。	パーサーは content_characters イベントを生成し、XML テキストには (単一の) 無効な文字が入る。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 content_characters イベントを生成し、XML テキストには 2 から 3 文字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 34. 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。

表 34. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
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/小文字を使用した「xml」が入る。
13	16 進文字参照 (形式 <code>&#xddd;</code>) 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 <code>&#ddd;</code>) 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。
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。
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。

表 34. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
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。
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。

表 34. 継続可能な例外 (続き)

番号	説明	継続時のパーサーの処置
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。
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 35. 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。

表 35. 終了例外 (続き)

番号	説明
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先頭文字が、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。
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。

表 35. 終了例外 (続き)

番号	説明
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。
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。

表 35. 終了例外 (続き)

番号	説明
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。
500 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 29 章 アプリケーションにおける PL/I MLE の使用

属性およびオプションの適用	477	日付の比較	482
DATE 属性	477	日付の変換	482
RESPECT コンパイル時オプション	478	日付の減算	482
WINDOW コンパイル時オプション	478	暗黙的な日付計算	483
RULES コンパイル時オプション	479	暗黙的な日付比較	483
日付パターンの理解	479	同種パターンを持った日付の比較	483
パターンおよびウィンドウ操作	480	異なるパターンを持った日付の比較	483
MLE による組み込み関数の使用	481	DATE 属性およびリテラルを伴う比較	483
DAYS	481	DATE 属性および非リテラルを伴う比較	484
DAYSTODATE	482	暗黙的な DATE の代入	484
日付の計算および比較の実行	482	SQL プリプロセッサによる MLE の使用	485
明示的な日付計算	482		

MLE の導入とともに、PL/I for Windows は、多くの追加言語機能のサポートを受け入れています。この章の目的は、新しい属性、コンパイル時オプション、日付パターン、および組み込み機能を理解することです。本章を読み進めるにつれて、既存アプリケーションにこれらをどのように適用すればいいかがわかるようになります。

属性およびオプションの適用

以下のセクションで紹介する言語機能は、他のプログラミング・ガイドや言語リファレンスにもできますが、それらがどのように機能するかをよりよく理解できるように、本章ではそれらの説明を繰り返します。

DATE 属性

2 つのオペランドが DATE 属性を持つ場合は、コンパイラーによって暗黙の日付の比較および変換が行われます。DATE 属性は、変数、引数、または戻り値が、指定されたパターンの日付を持つことを指定します。2000 年言語拡張 は、479 ページの『日付パターンの理解』で説明する多くの日付パターンをサポートします。

パターン

サポートされる日付パターンのうちの 1 つ。パターンを指定しない場合は、YYMMDD がデフォルトになります。

DATE 属性は、次の属性セットのいずれかを持つ変数についてののみ有効です。

- CHAR(n) NONVARYING
- PIC'(n)9' REAL
- FIXED DEC(n,0) REAL

長さまたは精度の *n* は、日付パターンまたはデフォルト・パターンの長さに等しい定数でなければなりません。

RESPECT コンパイル時オプション (この章の後で説明) を指定した場合、DATE 組み込み関数は、属性 DATE('YYMMDD') を持った値を戻します。これにより、エラー・メッセージが生成されることなく、DATE() を、属性 DATE('YYMMDD') を持

った変数に割り当てることができます。ただし、DATE() を、DATE 属性を持たない変数に割り当てるとは、エラー・メッセージが生成されます。

以下に、DATE 属性を使用した例をいくつか示します。

```
dcl gregorian_Date char(6) date;

dcl julian_Date    pic'(5)9' date ('YYDDD');

dcl year           fixed dec(2) date('YY');
```

アプリケーションに 2000 年問題がない場合にも、DATE 属性は有効です。以下の例に示すように、その属性を使用して、異なる日付を取り扱うことができます。

```
dcl gregorian_Date  char(8) date ('YYYYMMDD');

dcl julian_Date     pic'(7)9' date ('YYYYDDD');

if julian_Date > gregorian_Date then ...
```

RESPECT コンパイル時オプション

RESPECT オプションは、コンパイラーがどの属性を認識すべきかを指定するときに使用します。現在、このコンパイル時オプションで選択できるのは、DATE のみです。

デフォルトは RESPECT() であり、この場合、コンパイラーは DATE 属性のすべての指定を無視します。したがって、DATE 属性は、DATE 組み込み関数の結果には適用されません。NORESPECT は RESPECT() の同義語です。

他方、RESPECT(DATE) 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() は、TSO/MVS 上で PLI コマンドを使用してコンパイルするときは受け入れられません。

WINDOW コンパイル時オプション

デフォルトでは、2 桁の年をもったすべての日付は、1950 に始まり、2049 に終わるウィンドウ内に表示されます。WINDOW オプションを使用して、世紀ウィンドウの値を変更できます。

すでに説明したように、このオプションのデフォルトは WINDOW(1950) です。w の値には、以下のいずれかを指定できます。

- 固定世紀ウィンドウの開始を表す、1582 から 9999 (両端を含む) 間の符号なし整数
- 「スライド」世紀ウィンドウを作成する、-1 から -99 (両端を含む) 間の負の整数
- ゼロ。w の値は現在の年であることを示します。

固定ウィンドウを作成するには、WINDOW(1900) と指定することができます。このように指定すると、2 桁のすべての年は、20 世紀に出現するとみなされます。

現在の年が 1998 であり、スライド・ウィンドウを作成したい場合は、WINDOW(-5) と指定できます。生成される世紀ウィンドウは、1993 から 2092 (両端を含む) の年の幅を持ちます。年が 1999 に変わると、ウィンドウも、1 年だけ前方に移動します。

WINDOW コンパイル時オプションを使用して世紀ウィンドウの値を設定する場合、組み込み関数で特に指定のない限り、その値は、それを受け入れる組み込み関数のウィンドウ引数として使用されます。詳細については、481 ページの『MLE による組み込み関数の使用』を参照してください。

RULES コンパイル時オプション

一般に、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不可にすることができ、代替の選択肢があればセマンティクスを選択できます。現在、このオプションで選択できるのは、LAXCOMMENT のみです。

デフォルトは RULES(NOLAXCOMMENT) です。LAXCOM および NOLAXCOM は、サブオプションの受け入れ可能な省略形です。

RULES(LAXCOMMENT) を指定すると、コンパイラーは、特殊文字 /*/ を無視します。したがって、文字セット間にあるものはすべて、コメントではなくて、構文の一部として解釈されます。RULES(NOLAXCOMMENT) を指定すると、コンパイラーは、/*/ をコメントの開始と解釈し、終了の */ のところまでコメントが続くと解釈します。

メインフレームに移植するワークステーション・コードがあり、このコードが、DATE 属性の周りで /*/ を使用している場合は、コンパイラーがその属性を受け付けるように、RULES(LAXCOMMENT) オプションを使用する必要があります。

日付パターンの理解

PL/I MLE は、次の表に示すような、一連の日付パターンをサポートします。

表 36. PL/I MLE によってサポートされる日付パターン

	4 桁の年	例	2 桁の年	例
年が最初にくる	YYYY	1999	YY	99
	YYYYMM	199912	YYMM	9912
	YYYYMMDD	19991225	YYMMDD	991225
	YYYYMMM	1999DEC	YYMMM	99DEC
	YYYYMMDD	1999DEC25	YYMMDD	99DEC25
	YYYYMmm	1999Dec	YYMmm	99Dec
	YYYYMmmDD	1999Dec25	YYMmmDD	99Dec25
	YYYYDDD	1999359	YYDDD	99359
月が最初にくる	MMYYYY	121999	MMYY	1299
	MMDDYYYY	12251999	MMDDYY	122599
	MMYYYY	DEC1999	MMYY	DEC99
	MMDDYYYY	DEC251999	MMDDYY	DEC2599
	MmmYYYY	Dec1999	MmmYY	Dec99
	MmmDDYYYY	Dec251999	MmmDDYY	Dec2599

表 36. PL/I MLE によってサポートされる日付パターン (続き)

	4 桁の年	例	2 桁の年	例
日が最初にくる	DDMMYYYY	25121999	DDMMYY	251299
	DDMMMYYYY	25DEC1999	DDMMMYY	25DEC99
	DDMmmYYYY	25Dec1999	DDMmmYY	25Dec99
	DDDYyyy	3591999	DDDYy	35999

これらのパターンのいずれかから日または月を省略すると、コンパイラーは、それが 1 の値を持つとみなします。

日または月は省略されないが、00/38/11 のように範囲外の場合は、日付が比較を含むときはメッセージが発行されます。規則の例外は、YYMM および YYMMDD のパターンですべての値がゼロの場合です。この場合、このパターンは、1 のユリウス日付、つまり、最小の有効な日付に変換されます。

パターンおよびウィンドウ操作

2 桁の年 (YY) を持った日付をどのように解釈するかを定義するには、WINDOW コンパイル時オプションを使用して、世紀ウィンドウを定義します。前に説明したように、世紀ウィンドウは、2 桁の年が適用される、100 年の幅の開始年を定義します。

PL/I の 2000 年言語拡張 を使用しない場合は、window (w) を使用して、y2 を、2 桁の年から 4 桁の年に変換する、次のような論理をインプリメントする必要があります。

```

dcl y4 pic'9999';
dcl cc pic'99';

cc = w/100;

if y2 < mod(w,100) then
  y4 = (100 * cc) + 100 + y2;
else
  y4 = (100 * cc) + y2;
```

この例を使用すると、WINDOW(1900) を指定する場合は、19 は 1919 年と解釈されます。しかし、WINDOW(1950) を指定する場合は、19 は 2019 年と解釈されます。

逆に、この論理は、4 桁の年から変換する場合は、2 桁の年 (y2) を計算します。

```

dcl y4 pic'9999';

if y4 < w | y4 >= w + 100 then
  signal error;

y2 = mod(y4,100);
```

MLE による組み込み関数の使用

PL/I MLE の日付パターンは、DAYS および DAYSTODATE 組み込み関数によってサポートされます。これらのいずれの組み込み関数も、2 桁年パターンの処理で使用するウィンドウを指定する、オプションの引数 (w) を受け入れます。w を DAYS または DAYSTODATE の一部として指定すると、入力した値は、WINDOW コンパイル時オプションによって定義された値をオーバーライドします。

DAYS

DAYS は、日付 *d* に対応する日数 (リリアン形式) である FIXED BINARY(31,0) 値を戻します。

d 日付を表すストリング式。省略すると、DATETIME() によって戻される値とみなされます。

d の値は、文字タイプを持つ必要があります。そうでない場合、*d* は文字に変換されます。

p 479 ページの表 36 で示した、サポートされる日付パターンのうちの 1 つ。省略すると、コンパイラーは、*p* を、DATETIME 組み込み関数によって戻されるデフォルト・パターン (YYYYMMDDHHMISS999) とみなします。

p は文字タイプを持つ必要があります。そうでない場合、それは文字に変換されます。

w 2 桁年フォーマットの処理に使用される世紀ウィンドウを定義する、整数式。

- 1950 のように値が正の場合、年として扱われます。
- 負またはゼロの場合、その値は、現在のシステム提供年から減算するオフセットを指定します。
- 省略すると、*w* は、WINDOW コンパイル時オプションで指定された値にデフォルト設定されます。

次の例は、DAYS および DAYSTODATE の両方の組み込み関数の使用法を示しています。

```

dcl date_format char(8) static init('MMDDYYYY');
dcl todays_date char(8);
dcl sep2_1993 char(8);
dcl days_of_july4_1993 fixed bin(31);
dcl msg char(100) varying;
dcl date_due char(8);

todays_date = daystodate(days(),date_format);

days_of_july4_1993 = days('07041993','MMDDYYYY');
sep2_1993 = daystodate(days_of_july4_1993 + 60, Date_format);
           /* 09021993 */

date_due = daystodate(days() + 60, date_format);
           /* assuming today is July 4, 1993, this would be Sept. 2, 1993

msg = 'Please pay amount due on or before ' ||
      substr(date_due, 1, 2) || '/' ||
      substr(date_due, 3,2) || '/' ||
      substr(date_due, 5);

```

DAYSTODATE

DAYSTODATE は、 d 日数 (リリアン形式) に対応する p 形式の日付を含む、変換しない文字列を返します。

d 日数 (リリアン形式)。

d は計算タイプを持つ必要があり、必要に応じて、FIXED BINARY(31,0) に変換されます。

p 479 ページの表 36 で示した、サポートされる日付パターンの中の 1 つ。省略すると、コンパイラーは、 p を、DATETIME 組み込み関数によって戻されるデフォルト・パターン (YYYYMMDDHHMISS999) とみなします。

p は文字タイプを持つ必要があります。そうでない場合、それは文字に変換されます。

w 2 桁年フォーマットの処理に使用される世紀ウィンドウを定義する、整数式。

- 1950 のように値が正の場合、年として扱われます。
- 負またはゼロの場合、その値は、現在のシステム提供年から減算するオフセットを指定します。
- 省略すると、 w は、WINDOW コンパイル時オプションで指定された値にデフォルト設定されます。

日付の計算および比較の実行

PL/I 2000 年言語機能を理解し、適切な構文の変更を行えば、MLE を使用して、アプリケーションで計算および比較を実行できます。

明示的な日付計算

DAYS および DAYSTODATE 組み込み関数を使用して、日付の比較および計算を手動で行うことができます。

日付の比較

日付パターン YYMMDD を持つ 2 つの日付 $d1$ および $d2$ を比較するには、次のコードを使用できます。

```
DAYS (d1, 'YYMMDD', w) < DAYS(d2, 'YYMMDD', w)
```

日付の変換

次の代入を使用して、パターン YYMMDD を持った 2 桁の日付 ($d1$) とパターン YYYYMMDD を持った 4 桁の日付 ($d2$) 間の変換を行うことができます。

```
d2 = DAYSTODATE(DAYS(d1,'YYMMDD',w), 'YYYYMMDD');
d1 = DAYSTODATE(DAYS(d2,'YYYYMMDD'), 'YYMMDD', w);
```

日付の減算

2 つの 2 桁年 $y1$ と $y2$ の減算を行うには、次のように、見かけの差を計算する必要があります。

```
DAYSTODATE(DAYS(y1,'YY',w), 'YYYY') -
DAYSTODATE(DAYS(y2,'YY',w), 'YYYY')
```

暗黙的な日付計算

最初に次の 2 つのステップを完了すれば、MLE を使用して、暗黙的な日付の比較および変換を活用できます。

- 2 つのオペランドに DATE 属性を与える
- RESPECT コンパイル時オプションを指定する

暗黙的な日付比較

DATE 属性により、DATE 属性によって宣言された 2 つの変数を比較するとき、暗黙的な共通化が行われます。1 つの変数のみが DATE 属性を持つ比較にはフラグが付けられます。その他の被比較数は、通常、同じ DATE 属性を持つかのよう処理されます。ただし、後から説明するいくつかの例外があります。

暗黙的な共通化とは、コンパイラーが、日付を共通な比較可能表記に変換するコードを生成することを意味します。この処理では、WINDOW コンパイル時オプションで指定したウィンドウを使用して、2 桁年が変換されます。

次のコード断片において、DATE 属性が受け付けられると、2 番目の display ステートメントの比較は「ウィンドウ化」されます。つまり、ウィンドウが 1900 から始まると、比較は false を戻します。しかし、ウィンドウが 1950 から始まると、比較は true を戻します。

```

dcl a   pic'(6)9' date;
dcl b   pic'(6)9' def(a);
dcl c   pic'(6)9' date;
dcl d   pic'(6)9' def(c);

b = '670101';
d = '010101';

display( b || ' < ' || d || ' ? ');
display( a < c );

```

日付の比較は、以下の場所でも行うことができます。

- IF および SELECT ステートメント
- WHILE または UNTIL 文節
- TO 文節によって行われる暗黙的な比較

同種パターンを持った日付の比較

コンパイラーは、以下の条件のもとでは、同一パターンを持った日付を比較する特別なコードを生成しません。

- 比較演算子 = または <= が使用される
- パターンが YYYY、YYYYMM、YYYYDDD、または YYYYMMDD である

異なるパターンを持った日付の比較

異種パターンを持った日付を伴う比較では、コンパイラーは、日付を共通の比較可能表記に変換するコードを生成します。変換が行われた後に、コンパイラーは 2 つの値を比較します。

DATE 属性およびリテラルを伴う比較

1 つの被比較数が DATE 属性を持ち、その他がリテラルを持つ比較を行う場合、コンパイラーは W レベルのメッセージを発行します。以後のコンパイラー・アクションは、以下のように、リテラルの値によって異なります。

暗黙的な日付比較

- リテラルが有効な日付に見える場合、リテラルは、DATE 属性を持った被比較数と同じ日付パターンおよびウィンドウを持つかのように扱われます。
- リテラルが有効な日付に見えない場合、DATE 属性はその他の被比較数で無視されます。

```
decl start_date char(6) date;
if start_date >= '' then /* no windowing */
...
if start_date >= '851003' then /* windowed */
...
```

DATE 属性および非リテラルを伴う比較

1 つの被比較数が DATE 属性を持ち、その他が日付でも、リテラルでもない比較では、コンパイラーは E レベルのメッセージを発行します。日付でない値は、他の被比較数と同じ日付パターンを持ち、同じウィンドウを持つかのように扱われます。

```
decl start_date char(6) date;
decl non_date char(6);

if start_date >= non_date then /* windowed */
...
```

暗黙的な DATE の代入

DATE 属性によって、日付パターンによって宣言された 2 つの変数の代入において、暗黙的な変換も行われます。

- ソースとターゲットが同じ DATE 属性およびデータ属性を持つ場合、代入は、どちらも DATE 属性を持たないかのように行われます。
- ソースとターゲットが異なる DATE 属性を持つ場合、コンパイラーは、代入を行う前に、ソース・データを変換するコードを生成します。
- ソースは DATE 属性を持つが、ターゲットは持たない代入では、コンパイラーは、E レベルのメッセージを発行し、DATE 属性を無視します。
- ターゲットは DATE 属性を持つが、ソースは持たない (ソースはリテラル「ではない」) 代入では、コンパイラーは、E レベルのメッセージを発行し、DATE 属性を無視します。
- ターゲットは DATE 属性を持つが、ソースは持たない (ソースはリテラル「である」) 代入では、コンパイラーは、W レベルのメッセージを発行し、DATE 属性を無視します。

```
decl start_date char(6) date;
start_date = '';
...
```

- ソースが 4 桁の年を持ち、ターゲットが 2 桁の年を持つ場合、ソースは、ターゲット・ウィンドウにない年を持つことができます。この場合、ERROR 条件が発生します。

```
decl x char(6) date;
decl y char(8) date('YYYYMMDD');

y = '20600101';

x = y; /* raises error if window is <= 1960 */
```

- 以下において、DATE 属性は無視されます。
 - デバッガー
 - レコード入出力ステートメントで実行される代入

- ストリーム入出力ステートメント (GET DATA など) で実行される代入および変換

ウィンドウ操作による解決方法を選択しない場合でも、2 桁年と 4 桁年の両方を取り扱う必要のあるコードがあります。このような場合は、次のように、複数の日付パターンを使用できます。

```
dc1 old_date char(6) date('YYMMDD');  
dc1 new_date char(8) date('YYYYMMDD');  
  
new_date = old_date;
```

SQL プリプロセッサによる MLE の使用

SQL プリプロセッサは、DATE 属性を拒否します。ただし、この属性を `/*` と `*/` で囲めば、SQL プリプロセッサはそれを無視します (最初の `/*` から最後の `*/` までの範囲のコメントの一部として)。コンパイラーが、これらの特殊文字間の DATE 属性を受け付けるようにするには、`RULES(LAXCOMMENT)` を指定する必要があります。詳細については、479 ページの『RULES コンパイル時オプション』を参照してください。

第 7 部 付録

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。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ピー・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で提供し、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。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
J74/G4
555 Bailey Avenue
San Jose, CA 95141-1099
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

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。

プログラミング・インターフェース情報

本書には、プログラムを作成するユーザーが IBM PL/I for MVS & VM のサービスを使用するためのプログラミング・インターフェースが記述されています。

ユーザー用のマクロ

IBM PL/I for MVS & VM は、ユーザーのインストール・システムで、IBM PL/I for MVS & VM のサービスを使用するプログラムを作成できるマクロを提供していません。

重要: IBM PL/I for MVS & VM マクロは、プログラミング・インターフェースとして使用しないでください。

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。

AIX	IMS/ESA
CICS	Language Environment
CICS/ESA	OS/2
DFSMS/MVS	OS/390
DFSORT	Proprinter
IBM	Rational
IMS	VisualAge
	WebSphere

Windows 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。

参考文献

Enterprise PL/I 資料

- 「プログラミング・ガイド」、SC88-9123-02
- 「言語解説書」、SC88-9126-02
- 「メッセージおよびコード」、SC88-9127-02
- 「診断ガイド」、GC88-9125
- 「コンパイラーおよびランタイム・プログラム 移行ガイド」、GC88-9124-02

DB2 UDB for OS/390 および z/OS

- 「管理ガイド」、SC88-8761
- 「コマンド解説書」、SC88-8764
- 「SQL 解説書」、SC88-8772
- 「アプリケーション・プログラミングおよび SQL ガイド」、SC88-8763
- 「メッセージおよびコード」、GC88-8768

CICS Transaction Server

- 「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
- 「アプリケーション・プログラミング・リファレンス」、SC88-7690
- 「アプリケーション・プログラミング・ガイド」、SC88-7689

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タリー・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). プロシージャーへのエンタリー・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 つまたはそれ以上の文字、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 つまたは複数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現は国によって異なる場合がある) の拡張英字。

英数字 (alphanumeric 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。

エレメント式 (element expression). 評価されるとエレメント値を生じる式。

エレメント変数 (element variable). エレメントを表す変数。スカラー変数。

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 つまたは複数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 ユニットの処置として指定されている場合) か、または GOTO 文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。

外部記号 (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。

外部記号辞書 (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 の大文字、小文字の英字、\$, @、および #、または NAMES コンバイラー・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えること。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とは、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), 参照で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文脈内で使用されると特定の意味をもつ 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 つまたはそれ以上の数字、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、コロンの、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 つまたは複数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場所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 つまたは複数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 つの文脈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、構造型、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、それが構造型、共用体、または配列のいずれであるかの指定。

修飾名 (qualified name). 構造型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、構造型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 (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). 配列次元の最高限度。

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 つまたは複数の条件名のリスト。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一定義の条件の名前。

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、および定数からなり、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 (string). 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). 一連の実行。

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). 特定文脈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。このような式のアペランドは、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、その実行が終わるまで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か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 つまたは複数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一連のステートメント。

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思なすことができる。

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エレメント式。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 つまたはそれ以上の添え字のリスト。配列のおおのの次元に対して 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 つまたは複数の属性を、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ること。

[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 つまたは複数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。

データ型 (data type). 一連のデータ属性。

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分。

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). 単一のファイル名を参照すれば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・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の参照。

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]

内部プロシージャー (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ー。外部プロシージャー (*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・ロケーター値 (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ー値。これは、現在ロケーター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 つまたは複数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 つまたは複数の次元別にグループ分けされた 1 つまたは複数のデータ・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 つまたは複数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 つまたは複数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). ピクチャー指定で利用できるすべての文字。

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 (\mid)。

ビット・ストリング定数 (bit string constant). 囲まれていて、接尾部 B が直後に付いた一連の 2 進数字。文字定数 (character constant) と対比。単一引用符に囲まれ、後に接尾部 B4 が付いた一連の 16 進数字。

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作業。

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、プロシージャ、またはパッケージのブロック内のすべてのブロック、プロシージャ、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 つまたは複数の名前、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 つまたはそれ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出力名の 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). 入出力定数または入出力変数。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、場合に応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。

ポインター (pointer). ストレージ内の位置を識別するための変数のタイプ。

ポインター値 (pointer value). ポインター・タイプを識別する値。

ポインター変数 (pointer variable). ポインター値が入った 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。文脈中に現れる、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。

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 つまたは複数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一連の主記憶域。

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宛先変更すること。

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確保。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一連の 10 進数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 つまたは複数の 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 桁以下の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とまな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り、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). PLI 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]

アクセス方式
 入出力 200
 DDM 198
アプリケーション・パフォーマンスの向上 341
アプリケーション・プログラム
 SQL のコーディング
 データ宣言 307
移植性
 オペレーティング・システムの相違 11
 環境の相違 19
 国別文字およびその他の記号 12
 組み込み制御文字 11
 言語エレメント 20
 実行可能ファイルの作成 16
 データ表記 16
 マクロ機能の使用 15
 ランタイム動作での変更点 16
 論理エラーの回避 16
インクルード・ファイル
 ODBC での 302
インクルード・プリプロセッサ
 環境変数 102
 構文 102
エラー
 コンパイラまたはライブラリー 193
 システム障害 193
 ソース・プログラムの論理エラー 190
 未初期化の入り口変数の呼び出し 190
 予期しない
 入出力データ 191
 プログラム結果 193
 プログラム終了 192
 予期しないエラー 190
 ランタイム・メッセージ 188
 ループ 190
 ローパフォーマンス 194
 OS PL/I からの発行における相違 21
 PL/I の使用方法が無効 190
エラーおよび条件処理
 一般概念 186
 条件の ON ユニット 189

エラーおよび条件処理 (続き)
 静的派生 186
 通常の戻り 187
 テストおよびデバッグで使用する条件 189
 動的派生 186
 標準システム処置 187
 用語 186
 割り込みと PL/I 条件 186
オプション
 コンパイル時
 DEFAULT 387
 使用
 DD 情報 220
 TITLE 220
 入出力テーブル 222
 ランタイム
 NATLANG 175
DD:ddname 環境変数
 AMTHD 210
 APPEND 211
 ASA 211
 DELAY 212
 DELIMIT 213
 LRECL 213
 LRMSKIP 213
 PROMPT 213
 PUTPAGE 214
 RECCOUNT 214
 RECSIZE 214
 RETRY 215
 SAMELINE 215
 SHARE 216
 SKIPO 216
 TERMLBUF 217
 TYPE 217
FROMALIEN 411
PL/I ENVIRONMENT 属性
 BKWD 203
 BUFSIZE 211
 CONSECUTIVE 203
 CTLASA 204
 GENKEY 204
 GRAPHIC 206
 KEYLENGTH 206
 KEYLOC 207
 ORGANIZATION(CONSECUTIVE) 207
 ORGANIZATION(INDEXED) 207
 ORGANIZATION(RELATIVE) 208
 RECSIZE 208

オプション (続き)
 PL/I ENVIRONMENT 属性 (続き)
 REGIONAL(1) 208
 SCALARVARYING 209
 VSAM 209
 PRINT 属性
 LINE 234
 PAGE 234
 SKIP 234
オフセット
 ステートメント番号の判断 189
 タブ・カウンタ 238
オペレーティング・システム
 データ定義 (DD) 情報 220

[カ行]

改行 (LF)
 定義 217
 論理レコードの区切り 199
 LF ファイル 202
カスタマイズ
 コンパイル時環境変数の設定 30
 ユーザー出口
 グローバル制御ブロックの構造 360
 独自のコンパイラ出口の作成 360
 IBMUEXIT.INF の変更 359
各国語サポート 175
可変長レコード
 ソート 435
 フォーマット 202
画面およびキーボード操作 224
環境の相違、S/390 と AIX 19
環境変数
 インクルード・プリプロセッサ 102
 コンパイル時 30
 マクロ機能 105
 CICS プリプロセッサ 135
 SQL プリプロセッサ 115
管理、ライブラリーの (.LIB ファイル) 373
キー
 開始位置 207
 順次データ・セットへのアクセス 269
 順次レコード値の使用 268
 総称 204
 相対レコード
 埋め込み 269
 短縮形 269

キー (続き)	混合言語アプリケーション 405	コンパイラー・オプション (続き)
相対レコード番号の使用 269	コンソール	LINEDIR 64
ワークステーション VSAM キー順データ・セットの場合の使用 268	出力 242	LIST 64
キー順データ・セット 282	入力 239	LISTVIEW 64
ステートメントとオプション 277	コンパイラー	MACRO 66
タイプと利点 268	オプションの説明 37	MARGINI 66
キーの長さ、検査 206	リスト	MARGINS 66
キーボード	使用されるスタック・ストレージ	MAXGEN 67
画面操作 224	92	MAXMSG 68
疑似変数、制限された 14	コンパイラーの制約事項	MAXNEST 68
記述子域、SQL 116	疑似変数 14	MAXSTMT 69
机上検査 177	組み込み関数 14	MAXTEMP 69
既存のプログラムのワークステーション	構造式 13	MDECK 69
VSAM 向けの修正 271	自動変数のエクステンツ 14	MSG 70
キャラクター型装置 198	配列式 13	NAMES 70
行継続 29	DBCS 15	NATLANG 71
国別文字 12	DEFAULT ステートメント 14	NEST 71
組み込み	iSUB の定義 15	NOT 72
制御文字 11	MACRO プリプロセッサ 15	NUMBER 72
CICS ステートメント 135	RECORD I/O 12	OBJECT 73
SQL ステートメント 117	STREAM I/O 13	OFFSET 73
組み込み SQL	コンパイラー・オプション	ONSNAP 73
利点 298	省略形 37	OPTIMIZE 74, 342
組み込み関数	デフォルト 37	OPTIONS 74
制限付き 14	ADDEXT 40	OR 75
DAYS 481	AGGREGATE 40	PP 75
DAYSTODATE 482	ATTRIBUTES 41	PPCICS 76
グラフィック・データとストリーム入出力	BIFPREC 41	PPINCLUDE 77
229	BLANK 42	PPMACRO 77
グローバル制御ブロック	CHECK 43	PPSQL 78
終了プロシーチャーの作成 364	CMPAT 44	PPTRACE 78
初期化プロシーチャーの作成 361	CODEPAGE 44	PRECTYPE 79
データ入力フィールド 361	COMPILE 45	PREFIX 343
メッセージ・フィルター操作プロシーチャーの作成 362	COPYRIGHT 45	PREFIXE 79
グローバル制御ブロックの構造	CURRENCY 46	PROBE 80
終了プロシーチャーの作成 364	DBCS 46	PROCEED 80
初期化プロシーチャーの作成 361	DEFAULT 46, 344	QUOTE 81
メッセージ・フィルター操作プロシーチャーの作成 362	DLLINIT 54	REDUCE 82
計算、日付を使用した 482	EXIT 55	RESEXP 82
言語間通信 (ILC) 405	EXTRN 55	RESPECT 83, 478
検索規則	FLAG 55	RULES 83, 343, 479
リンカー 152	FLOATINMATH 56	SEMANTIC 90
コーディング	GONUMBER 56, 342	SNAP 90, 342
組み込み制御文字 11	GRAPHIC 57	SOSI 91
パフォーマンスの向上 348	IMPRECISE 57, 342	SOURCE 91
CICS ステートメント 135	INCAFTER 58	STATIC 91
SQL ステートメント 115	INCLUDE 58	STMT 92
コード検査 177	INITAUTO 59	STORAGE 92
構造式の制約事項 13	INITBASED 59	SYNTAX 92
固定長レコード・フォーマット 202	INITCTL 59	SYSPARM 93
コマンド行、ランタイム・オプションの設定 173	INITSTATIC 60	SYSTEM 93
コマンド行パラメーター、ILIB の 374	INSOURCE 60	TERMINAL 94
	LANGLVL 61	TEST 94
	LIBS 61	USAGE 95
	LIMITS 62	WIDECHAR 96
	LINECOUNT 63	WINDOW 96, 478

コンパイラー・オプション (続き)

XINFO 96

XML 99

コンパイラー・ユーザー出口の初期化プロ シージャー 361

コンパイル

環境変数

IBM.DECK 33

IBM.OBJECT 33

IBM.OPTIONS 31

IBM.PRINT 32

IBM.SOURCE 32

IBM.SYSLIB 32

INCLUDE 33

TMP 33

コンパイラーを呼び出す PLI コマンド の使用 33

コンパイル時オプション 37

障害 193

ソース・プログラムの準備 27

ユーザー出口

カスタマイズ 359

活動化 358

プロシージャー 358

IBMUEXIT 359

ワークステーション上でのメインフレ ーム・アプリケーション 11

コンパイル時オプション 12

指定する場所 34

デバッグで使用 179

コンパイル出力

コンパイラー出力 147

コンパイラー・リストの使用 139

コンパイル出力内の相互参照テーブル 144

コンパイルのためのソース・プログラムの 準備

行継続 29

プログラム・ファイルの構造 27

プログラム・ファイルのフォーマット
29

マージン 29

INCLUDE 処理 28

[サ行]

最適コーディング

コーディング・スタイル 348

コンパイル時オプション 341

索引データ・セット 200

サンプル・プログラム 26

サンプル・プログラム、実行 417, 422, 426

システム

エラー処理機能 188

障害 193

システム (続き)

条件に対する標準処置 187

メッセージ 188

実行、プログラムの

ランタイム環境変数の設定 173

ランタイム・オプションの指定 173

集合

長さテーブル、例 145

終了プロシージャー

構文

グローバル 360

特有の 364

コンパイラー・ユーザー出口 364

プロシージャー特有の制御ブロックの
例 364

出荷、ランタイムの 175

出力

コンソールへの

ストリーム・ファイルおよびレコー
ド・ファイル 242

対話式プログラムの例 243

PRINT ファイルのフォーマット
242

ストリーム・ファイル用のデータ・セ
ットの定義 229

SEQUENTIAL 244

順次

アクセス 257

データ・セット

ステートメントとオプション 273

ワークステーション VSAM 267

レコード値

ワークステーション VSAM 順次デ
ータ・セット内の 268

KEYTO を使用した検出 274

使用、2000 年言語拡張 の

言語機能 477

日付パターン 479

条件

インラインでのストリング組み込み関
数の処理 355

インラインの変換処理 354

条件処理

一般概念 186

修飾条件および非修飾条件 189

条件とその属性のリスト 189

有効範囲と派生 186

用語 186

割り込み 186

ON ユニットのコーディング 188

処理、条件の

条件処理の組み込み関数 189

条件のソース 188

PL/I ランタイム・エラー・メッセー
ジ・フォーマット 188

システム・メッセージ 188

処理、条件の (続き)

PL/I ランタイム・エラー・メッセー
ジ・フォーマット (続き)

SNAP メッセージ 188

数値の引数、リンカー用 158

ステートメント

DELETE 223

GET 223

LOCATE 223

READ 223

REWRITE 223

WRITE 223

ステートメント番号、オフセットから判断
する 189

ストリーム指向データ伝送

ストリーム指向データ伝送用

ENVIRONMENT オプション 229

ストリーム入出力によるデータ・セッ
トの作成 229

ストリーム入出力によるデータ・セッ
トへのアクセス

必須情報 233

例 233

ストリーム入出力を用いたファイルの
定義 229

PRINT ファイルの用法 234

SYSIN ファイルと SYSPRINT ファイ
ルの用法 239

ストリーム入出力

データ・セットの作成 229

必須情報 230

例 230

データ・セットへのアクセス 232

ストリーム・ファイルおよびレコード・フ ァイル 242

ストレージ

リスト内のレポート 92

制御ブロック

機能専用 358

グローバル制御 360

制御文字

プリンター 227

CTLASA オプションの ANS 204

生成、DECLARE ステートメントの 307

静的派生 186

静的リンク 149

宣言

ホスト変数、SQL プリプロセッサ
118

ソースの論理エラー 190

ソース・キー 256

ソート出口

E15 436

E35 438

ソート・プログラム

可変長レコード 435

ソート・プログラム (続き)

成功または失敗の伝達 430, 435
ソート・データの入出力 435
ソート・フィールドの指定 432
ソート・プログラムの使用準備 429
ソート・プログラムの呼び出し 434
入力および出力処理ルーチン 436
PLISRTx 427
S/390 とワークステーションの比較 427

ソート・プログラムの使用 427

送信、入出力の 224

装置

キャラクター型 198
標準 198
con 225
std 225

属性と相互参照テーブル 144

[タ行]

ダイナミック・リンク・ライブラリー

構築 367
コンパイル、リンク、実行 370
メインプログラムでの FETCH および
RELEASE の使用 371
DLL ソース・ファイルの作成 367

対話式プログラム、例 243

タブ制御テーブル 237

ダミー

レコード 253

ダンプ

エラー処理 186
オプション・ストリング 182
条件処理 186
タイトル・ストリング 183
定様式 PL/I ダンプ - PLIDUMP 183
デフォルト・オプション 183
トレース情報の SNAP ダンプ 186

端末

会話型入出力 240
出力 242
対話式プログラムの例 243
入力 239

直接アクセス 257

直接データ・セット 286, 295

追加または置換、ライブラリーでのオブジェクトの、ILIB 379

データ

構造体 307
タイプ
Java と PL/I の同等な 426
SQL と PL/I の同等な 119
テスト 178
伝送 203
表記、移植性 16

データ (続き)

ファイル
作成 221
データ・ファイルと OPEN の関連付け 221
PL/I ファイルのクローズ 222
変換 220
変換表 365
リモート・ファイル・アクセス 201
レコード 201

データ指示 I/O

パフォーマンスのためのコーディング 349

DBCS 定数 206

GRAPHIC オプションの指定 206

データ・セット

アクセス
レコード入出力 247
REGIONAL(1) の例 248

アクセス方式 200

キー順アクセス 200

切り離し 222

出力での拡張 211

出力の再作成 211

順次アクセス 198

ストリーム・ファイル 228

タイプ

固定長データ・セット 199
従来型テキスト・ファイルおよび装置 199
ネイティブ・データ・セット 198
領域データ・セット 200

定義と用法 266

デフォルトの識別 219

特性 198

特性の指定 201

特性の設定

データ・セット編成 202

レコード 201

レコード・フォーマット 202

DD:ddname 環境変数 210

PL/I ENVIRONMENT 属性 202

入出力ステートメント、属性、および

オプションの組み合わせ 222

ネイティブ、固定長 199

パスの設定 221

標準入力のリダイレクト

出力、およびエラー装置 225

複数のデータ・セットと 1 つのファイルの関連付け 222

複数ファイルとの関連付け 221

編成

オプション 202

デフォルト 202

領域の 208

DDM および VSAM 210

データ・セット (続き)

領域数 214

領域の 200

領域の最大値 214

レコード入出力アクセス 198

ワークステーション VSAM

キー順データ・セット 277

順次データ・セット 273

直接データ・セット 286

ファイルの定義 267

編成 266

DD:ddname 環境変数 209, 219

DISPLAY ステートメントの入出力 224

PL/I 標準ファイル (SYSPRINT と
SYSIN) 225

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 220

関連付けされていないファイルの使用 221

OPEN ステートメントの TITLE オプションの使用 220

PL/I によるデータ・セットの検索方法 221

PL/I ファイルのオープン 221

REGIONAL(1) 200

VSAM 200

データ・セットへのアクセス

ストリーム入出力 232

レコード入出力 247

REGIONAL(1) 259

REGIONAL(1) の例 248

定義ファイル

作成 367

定様式 PL/I ダンプ 182

テキスト・ファイル

従来型 198

LF 198

テスト・プログラム

コード検査 177

データ・テスト 178

パス・テスト 178

デバッグ・プログラム

一般的な PL/I エラー

コンパイラまたはライブラリー・サブルーチンの障害 193

システム障害 193

ソースの論理エラー 190

未初期化の入出力変数 190

予期しない入出力データ 191

予期しないプログラム結果 193

予期しないプログラム終了 192

ループおよびその他の予期しないエラー 190

ローパフォーマンス 194

デバッグ・プログラム (続き)
一般的な PL/I エラー (続き)
 PL/I の使用方法が無効 190
一般的なデバッグのヒント 179
コンパイル時オプションの使用 179
条件処理 186
ダンプ 182
デバッグでのフットプリントの使用
 DISPLAY 182
 PUT DATA 181
 PUT LIST 181
 PUT SKIP LIST 181
FLAG オプション 179
GONUMBER オプション 180
NOLAXDCL オプション 180
NOLAXIF オプション 180
PREFIX オプション 180
RULES オプション 180
SNAP オプション 180
XREF オプション 180
動的派生 186
特記事項 489
トレース情報 182

[ナ行]

長さ、レコードの
 最大 208
 指定 214
名前付き定数
 対静的変数 352
 定義 352
入出力
 アクセス方式
 BTRIEVE 200, 201
 ISAM 201
 REMOTE 201
 オプション・テーブル 222
 ステートメント・テーブル 222
 ソート・プログラムの使用 435
属性テーブル 222
転送 225
予期しない 191
DDM 200
入力
 コンソールからの制御 239
 コンソールへの
 ストリーム・ファイルおよびレコー
 ド・ファイル 242
 対話式プログラムの例 243
 PRINT ファイルのフォーマット
 242
 ストリーム・ファイル用のデータ・セ
 ットの定義 229
 対話式プログラムの例 243
 SEQUENTIAL 244

入力および出力処理ルーチン、ソート・プ
ログラム 436
ネイティブ・データ・セット
 アクセス 198
 キャラクター型装置 199
 固定長データ・セット 199
 従来型テキスト・ファイル 199
 タイプ 198
 領域データ・セット 200
 DDM データ・セット 200

[ハ行]

バイト反転整数 17
配列式の制約事項 13
パス・テスト 178
パターン、日付の 479
パフォーマンスの向上
 コンパイル時オプションの選択
 DEFAULT 344
 GONUMBER 342
 IMPRECISE 342
 OPTIMIZE 342
 PREFIX 343
 RULES 343
 SNAP 342
 パフォーマンスのためのコーディング
 名前付き定数対静的変数 352
 ライブラリー・ルーチンの呼び出し
 の回避 354
 ループ制御変数 350
 DATA 指示入出力 349
 DEFINED 対 UNION 352
 PACKAGE 対ネストされた
 PROCEDURE 351
 REDUCIBLE 関数 352
パラメーター、ILIB の 373
汎用レジスターの説明
 パラメーター 390
 例
 規格合致パラメーターのルーチンへ
 の受け渡し 390
 浮動小数点パラメーターのルーチン
 への受け渡し 392
比較、日付の
 暗黙的な 483
 異なるパターンによる 483
 同種パターンによる 482, 483
 非リテラルの使用 484
 リテラルの使用 483
日付の減算 482
日付の変換 482
標識変数、SQL 127
標準
 システム処置 187
 装置、ワークステーション 198

ファイル
 オープン 221
 既存のプログラムのワークステーション
 VSAM 向けの修正
 CONSECUTIVE ファイルの使用
 271
 INDEXED ファイルの使用 271
 REGIONAL(1) ファイルの使用
 271
 VSAM ファイルの使用 272
 クローズ 221
定義
 ストリーム入出力 229
 レコード入出力 246
プリンター向け 228
PL/I
 定義 197
 標準 225
REGIONAL(1) データ・セットの宣言
 254
STREAM 属性 228
SYSIN 239
SYSPRINT 239
ファイル属性、サポートされない
 BACKWARDS 13
 EXCLUSIVE 13
 TRANSIENT 13
ファイルの定義
 データ・セットの 269
 REGIONAL(1) データ・セット用 255
ファイル・マーク文字 (*) 242
復帰 - 改行 (CR - LF) 217
フットプリント、デバッグ用 181
不定長レコード・フォーマット 202
浮動小数点データ 18
プラットフォーム
 相違 19
プリプロセッサ
 組み込み 102
 マクロ機能 103
 マクロ・プリプロセッサ 103
CICS オプション 134
PL/I とともに提供される 101
SQL オプション 107
SQL プリプロセッサ 106
プリンター制御文字、ASA 227
プリンター向けファイル 227
印刷する行の長さの制御 235
印刷制御文字 227
タブ制御テーブルの指定変更 237
ファイルの作成の例 237
ANS 印刷制御文字
 リスト 228
 IBM Proprinter 制御文字 228
ASA オプション 227

プログラム
 ファイル・フォーマット
 行継続 29
 コンパイルの準備 27
 説明 29
 正しいフォーマット 27
 マージン 29
 要求 29
 INCLUDE 処理 28
プログラムのリンク
 応答ファイルの使用 153
 検索規則 152
 コマンド行の使用 149
 静的リンク 149
 ディレクトリーの指定 153
 入出力 152
 ファイルの作成
 実行可能ファイル 154
 ダイナミック・リンク・ライブラリー 155
 マップ 156
 戻りコード 156
 リンカーの開始 149
 MAKE ファイルの使用 151
分散データ管理 198
ページ
 PAGELength タブ設定テーブル・フィールド 238
 PAGESIZE タブ設定テーブル・フィールド 238
米国標準規格 (ANS)
 プリンター向けファイル内の 227
 CTLASA オプションの 204
変換表 365
編集指示 I/O 206
変数
 コンパイル時の環境変数 30
編成
 データ・セット 202
 デフォルト 203
 領域データ・セット 208
 VSAM 209
ホスト
 構造体 126
 変数、SQL ステートメント内での使用 118
ホスト変数の使用、SQL プリプロセッサ 118

[マ行]

マージン 29
マイグレーション
 ワークステーション用 OS PL/I ファイル
 CONSECUTIVE ファイル 271

マイグレーション (続き)
 ワークステーション用 OS PL/I ファイル (続き)
 EXCLUSIVE ファイル 271
 INDEXED ファイル 271
 ISAM レコード処理 271
 REGIONAL(1) ファイル 271
 VSAM ファイル 272
 OS PL/I との互換性 11
マクロ機能
 移植性 15
 環境変数 105
 IBM.PPMACRO 31
 マクロ定義 103
マクロ・プリプロセッサ
 マクロ定義 103
マシン割り込み 186
未初期化の入り口変数 190
メインフレーム・アプリケーション
 ワークステーション上での実行 16
メッセージ
 コンパイラー・ユーザー出口での変更 359
 フィルター機能 362
 メッセージのフィルター操作 359
モジュール・テスト 178
戻りコード、リンカー 156

[ヤ行]

ユーザー定義関数、SQL プリプロセッサ 124
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 360
 独自のコンパイラー出口の作成 360
 IBMUEXIT.INF の変更 359
機能 358
コンパイラー 357
CICS ランタイム 364
予期しない
 入出力データ 191
 プログラム終了 192, 193
呼び出し
 コンパイラー 33
呼び出しインターフェースの規則
 ODBC での 302
呼び出し規則 387
汎用レジスターの説明
 パラメーター 390
 パラメーターの受け渡し例 390

[ラ行]

ラージ・オブジェクト (LOB) サポート、SQL プリプロセッサ 121
ライブラリー、コンパイラー・サブルーチンの障害 193
ライブラリー・マネージャー 373
ライブラリー・ルーチンの呼び出しの回避 354
ランタイム
 オプション、指定 173
 動作の相違
 言語エレメント 20
 AREA の INITIAL 属性は無視される 20
 ERROR メッセージの発行 21
 FIXED BIN として宣言された変数の使用 20
 プラットフォーム間の相違 16
 メッセージ
 SNAP 188
 SYSTEM 188
 DLL の出荷 175
ランタイム・オプション
 複数のランタイム・オプションまたはサブオプションの指定 175
 ランタイム・オプションの指定場所 173
 NATLANG 175
ランタイム・オプションの指定 173
リスト指示 I/O
 DBCS 定数 206
 GRAPHIC オプションの指定 206
リモート・アクセス 198
リモート・ファイル・アクセス 198
領域 214
領域データ・セット
 コマンドおよびオプション 254
 説明 253
 必要な情報 256
 ファイル定義
 領域データ・セットでのキーの使用 256
 ENVIRONMENT オプションの指定 256
 REGIONAL(1) データ・セットの使用更新 259
 順次アクセス 259
 ダミー・レコード 257
 直接アクセス 260
 例 260
領域番号 257
リンカー・オプションの設定 157
リンケージ
 OPTLINK
 機能 389

リンケージ (続き)
OPTLINK (続き)
 使用のヒント 390
 例 390
SYSTEM
 説明 395
 例 396
ルーチン、ライブラリー、変換 355
ループ
 使用のヒント 190
 制御変数 350
ON ユニットのコーディング 190
レコード
 長さ 214
 長さの指定 201
 ワークステーション VSAM データ・セットでのアクセス 267
レコード単位入出力
 データ伝送用 ENVIRONMENT オプション 246
 データ・セットの作成 246
 データ・セットへのアクセス 247
 によるデータ・セットの更新 247
 によるファイルの定義 246
 必須情報 248
 連続データ・セットの例 248
レコード・フォーマット 202
連結 30
練習問題 25
 コンパイル時オプションの使用 26
 添付されたサンプル・プログラムの使用 26
HELLO プログラム 25
連続データ・セット
 コンソールからの入力の制御
 大文字と小文字 241
 会話型のファイルの使用 240
 ストリーム・ファイルおよびレコード・ファイル 241
 データのフォーマット 240
 ファイル終わり 242
 コンソールへの出力の制御
 ストリーム・ファイルおよびレコード・ファイル 242
 対話式プログラムの例 243
 PRINT ファイルのフォーマット 242
ストリーム指向データ伝送の用法
 ストリーム入出力によるデータ・セットの作成 229
 ストリーム入出力によるデータ・セットへのアクセス 232
 ストリーム入出力を用いたファイルの定義 229
ENVIRONMENT オプション 229
PRINT ファイルの用法 234

連続データ・セット (続き)
 ストリーム指向データ伝送の用法 (続き)
 SYSIN ファイルと SYSPRINT ファイルの用法 239
説明 227
プリンター向けファイル 227
例 248
レコード単位入出力の使用
 データ伝送用 ENVIRONMENT オプション 246
 データ・セットのアクセスと更新 247
 データ・セットの作成 246
 ファイルの定義 246
PRINT ファイル 242
連絡域、SQL 115
ローパフォーマンス 194

[ワ行]

ワークステーション
 テキスト・ファイル 198
 ネイティブ・データ・セット 198
 レコード・フォーマット 202
ワークステーション VSAM キー順データ・セット
 ロード 280
 DIRECT ファイルを使用したアクセス 283
 SEQUENTIAL ファイルを使用したアクセス 282
ワークステーション VSAM 順次データ・セット
 更新 276
 定義とロード 275
 SEQUENTIAL ファイルからのアクセス 274
 SEQUENTIAL ファイルを使用したアクセス 274
ワークステーション VSAM 直接データ・セット
 ロード 289
 DIRECT ファイルを使用したアクセス 292
 SEQUENTIAL ファイルを使用したアクセス 291
ワークステーション VSAM データ・セット
 キー順
 基本索引ファイル 266
 基本ファイル 266
 順次 267
 タイプと利点 267
 タイプの選択 269
 直接 267

ワークステーション VSAM データ・セット (続き)
 ファイル宣言 269
 ファイルの定義
 既存のプログラムの修正 270
 PL/I ENVIRONMENT 属性のオプションの指定 270
 プログラムの修正
 CONSECUTIVE ファイルの使用 271
 INDEXED ファイルの使用 271
 REGIONAL(1) ファイルの使用 271
 VSAM ファイルの使用 272
 レコードへのアクセス 267
ワークステーション VSAM データ・セット内の相対レコード番号 269
ワークステーション VSAM データ・セットによる入出力
 説明 265
 編成

 キーの用法 268
 作成とアクセス 266
 内のレコードへのアクセス 267
 必要なタイプの判別 267
ワークステーション VSAM キー順データ・セットの使用
 ロード 280
 DIRECT ファイルを使用したアクセス 283
 SEQUENTIAL ファイルを使用したアクセス 282
ワークステーション VSAM 順次データ・セットの使用
 更新 276
 定義とロード 275
 SEQUENTIAL ファイルを使用したアクセス 274
ワークステーション VSAM 直接データ・セットの使用
 ロード 289
 DIRECT ファイルを使用したアクセス 292
 SEQUENTIAL ファイルを使用したアクセス 291
ワークステーション用 OS PL/I ファイルの互換性 271
割り込み 180

[数字]

2000 年言語拡張
PL/I アプリケーションでの使用 477
SQL プリプロセッサによる使用 485

A

ADDBUFF ENVIRONMENT オプション 13
ADDEXT コンパイラー・オプション 40
AGGREGATE コンパイラー・オプション 40
ALIGNED コンパイル時サブオプション 54
AMTHD オプション 210
ANS
印刷制御文字 228
コンパイル時サブオプション 47
制御文字 204, 227
APPEND オプション 211
AREA および INITIAL 属性 20
ASA オプション 211
ASCII
移植性の考慮事項 17
コンパイル時サブオプション
説明 48
パフォーマンスへの影響 347
データ変換表 365
DBCS の移植性 19
ASCII ENVIRONMENT オプション 13
ASSIGNABLE コンパイル時サブオプション 48
ATTRIBUTES コンパイラー・オプション 41

B

BACKUP オプション、ILIB 用 382
BACKWARDS ファイル属性 13
BIFPREC コンパイラー・オプション 41
BIN1ARG コンパイラー・サブオプション 51
BKWD オプション 203
BLANK コンパイラー・オプション 42
BTRIEVE アクセス方式 201
BUFFERS ENVIRONMENT オプション 13
BUFND ENVIRONMENT オプション 13
BUFNI ENVIRONMENT オプション 13
BUFOFF ENVIRONMENT オプション 13
BUFSIZE オプション 211
BYADDR
説明 345
パフォーマンスへの影響 345
DEFAULT オプションでの用法 48
BYVALUE
説明 345
パフォーマンスへの影響 346
DEFAULT オプションでの用法 48

C

CEE.OPTIONS 環境変数 174
CHECK コンパイラー・オプション 43
CICS
環境変数 135
IBM.PPCICS 32
サポート 132
プリプロセッサのオプション 134
ランタイム・ユーザー出口 364
CMPAT コンパイラー・オプション 44
CODEPAGE コンパイラー・オプション 44
COMPILE コンパイラー・オプション 45
CONNECT TO ステートメント 128
CONNECTED コンパイル時サブオプション
説明 48
パフォーマンスへの影響 346
CONSECUTIVE
ファイル 271
option
ストリーム入出力 229
定義 203
COPYRIGHT コンパイラー・オプション 45
CTLASA オプション 204
CURRENCY コンパイラー・オプション 46
CURRENCY コンパイル時オプション
移植性 12

D

DATE 属性
定義および構文 477
無視される場合 484
DAYS 組み込み関数 481
DAYSTODATE 組み込み関数 482
DBCS (2 バイト文字セット)
および GRAPHIC オプション 206
テーブル名 307
DBCS コンパイラー・オプション 46
DBCS の制約事項 15
DCLGEN 307
DD 情報
レコード・フォーマット 202
TITLE ステートメント 220
DDM アクセス方式 198, 200
DDM データ・セット
レコード・フォーマット 202
AMTHD の値 210
DD:ddname 環境変数 209
代替 ddname 220
特性の指定 209
AMTHD 210

DD:ddname 環境変数 (続き)

APPEND 211
ASA 211
DELAY 212
DELIMIT 213
LRECL 213
LRMSKIP 213
PROMPT 213
PUTPAGE 214
RECCOUNT 214
RECSIZE 214
RETRY 215
SAMELINE 215
SHARE 216
SKIP0 216
TERMLBUF 217
TYPE 217
DECLARE
STATEMENT 定義 129
TABLE ステートメント 128
DEF オプション、ILIB 用 383
DEF ファイル
作成 367
DEFAULT コンパイラー・オプション 46
サブオプション
ALIGNED 54
ASCII または EBCDIC 48
ASSIGNABLE または
NONASSIGNABLE 48
BIN1ARG または
NOBIN1ARG 51
BYADDR または BYVALUE 48
CONNECTED または
NONCONNECTED 48
DESLIST または
DESCLOCATOR 52
DESCRIPTOR または
NODESCRIPTOR 48
E 54
EVENDEC または
NOEVENDEC 50
IBM または ANS 47
IEEE または HEXADEC 50
INITFILL または NOINITFILL 51
INLINE または NOINLINE 49
LINKAGE 50
NATIVE または NONNATIVE 49
NATIVEADDR または
NONNATIVEADDR 49
NULLSTRADDR または
NONNULLSTRADDR 52
NULLSTRPTR 52
NULLSYS または NULL370 52
ORDINAL 50

DEFAULT コンパイラー・オプション (続き)

サブオプション (続き)

OVERLAP または

NOOVERLAP 50

RECURSVIE または

NONRECURSIVE 52

REORDER または ORDER 50

RETCODE 53

RETURNS 53

SHORT 53

デフォルトのサブオプションの使用
344

DEFAULT コンパイル時オプション

サブオプション

DUMMY 53

LOWERINC または

UPPERINC 51

DEFAULT ステートメントの制約事項
14

DEFINED

対 UNION 352

DELAY オプション

説明および構文 212

DELETE ステートメント 223

DELIMIT オプション

説明および構文 213

DESLIST コンパイル時サブオプション
52

DESCLOCATOR コンパイル時サブオプション
52

DESCRIPTOR コンパイル時オプション

パフォーマンスへの影響 346

DESCRIPTOR コンパイル時サブオプション

説明 48

DIRECT ファイル

使用してワークステーション VSAM

直接データ・セットにアクセス 292

ワークステーション VSAM キー順データ・

セットにアクセスするための

使用 283

DISPLAY 182

DLL 367

DLL からのデータのエクスポート 371

DLLINIT コンパイラー・オプション 54

DPATH ランタイム環境変数 173

DRIVER サンプル・プログラム

コンパイル、リンク、および実行 371

実行時に DLL を FETCH する例

371

DRIVER1.DEF ファイル

DLL を構築するサンプル・プログラム

369

DRIVER1.PLI ファイル

DLL を使用するサンプル・プログラム

370

DSNTIAR.PLI サンプル・プログラム

129

DUMMY コンパイル時サブオプション

53

E

E コンパイル時サブオプション 54

EBCDIC

移植性の考慮事項 17

コンパイル時サブオプション 48

データ変換表 365

パフォーマンスへの影響 347

DBCS の移植性 19

ENVIRONEMENT オプション、サポート
されない 13

ENVIRONMENT オプション

ストリーム指向データ伝送 229

レコード単位データ伝送用

CONSECUTIVE 246

CTLASA 246

ORGANIZATION(CONSECUTIVE)
246

RECSIZE 246

SCALARVARYING 246

ENVIRONMENT 属性

オプション

CTLASA 227

オプションの指定

ストリーム入出力 229

レコード入出力用の 246

ワークステーション VSAM データ・

セット用の 270

特性の指定 203

BKWD 203

BUFSIZE 211

CONSECUTIVE 203

CTLASA 204

GENKEY 204

GRAPHIC 206

KEYLENGTH 206

KEYLOC 207

ORGANIZATION 207

RECSIZE 208

REGIONAL(1) 208

SCALARVARYING 209

VSAM 209

CLOSE ステートメントの (REREAD)
領域データ・セット 255

ERROR

ON ユニット 21

EVENDEC コンパイル時サブオプション

50

EXCLUSIVE ファイル属性 13

EXEC SQL ステートメント 106

EXIT コンパイラー・オプション 55

EXTDICTIONARY オプション、ILIB 用
384

EXTRACT オブジェクト、ILIB 用 380

EXTRN コンパイラー・オプション 55

F

FETCH ステートメント

メインプログラムでの使用 371

filespec 210

FILLERS 238

FIXED

BINARY、マッピングおよび移植性
20

TYPE オプション 218

FLAG コンパイラー・オプション 55

FLAG コンパイル時オプション

デバッグ時に使用 179

FLOATINMATH コンパイラー・オプション
56

FREEFORMAT オプション、ILIB 用
383

FROMALIEN コンパイル時サブオプション
411

G

GENDEF (/gd) オプション、ILIB 用 383

GENIMPLIB (/gi) オプション、ILIB 用
383

GENKEY オプション 203, 204

GET ステートメント 223

コンソールからの入力の制御 240

GRAPHIC オプション 206

GONUMBER コンパイラー・オプション
56, 342

GONUMBER コンパイル時オプション
デバッグ時に使用 180

GRAPHIC

ENVIRONMENT オプション 206,
229

GRAPHIC コンパイラー・オプション 57

H

HELLO プログラム 25

HELP オプション、ILIB 用 384

HEXADECIMAL

移植性の考慮事項 18

コンパイル時サブオプション 50

I

IBM コンパイル時サブオプション 47
IBMUEXIT コンパイラ出口 359
IEEE

移植性の考慮事項 18
コンパイル時サブオプション 50

ILIB

応答ファイルの使用 375
オブジェクト 379
オプション 381
概要 373
出力 377
入力 377
パラメーターの指定 373
呼び出し 373

ILINK 環境変数 158

ilink 構文 149

IMPRECISE コンパイラ・オプション
57

パフォーマンスの向上 342

INCAFTER コンパイラ・オプション
58

INCLUDE

環境変数 33
処理 28
ステートメント、DCLGEN の使用
311

INCLUDE コンパイラ・オプション 58

INDEXAREA ENVIRONMENT オプショ
ン 13

INDEXED ファイル、プログラムの修正
271

INITAUTO コンパイラ・オプション
59

INITBASED コンパイラ・オプション
59

INITCTL コンパイラ・オプション 59

INITFILL コンパイル時サブオプション
51

INITIAL 属性 20

INITSTATIC コンパイラ・オプション
60

INLINE コンパイル時サブオプション 49

INSOURCE コンパイラ・オプション
60

ISAM アクセス方式 201

iSUB 定義の制約事項 15

J

Java 413, 414, 415, 417, 418, 419, 420,
421, 422, 424, 425, 426

Java コード、コンパイル 415, 419, 424

Java コード、作成 414, 418, 422

jni

JNI サンプル・プログラム 414, 418,
422

K

KEY

順次レコード値 269
相対レコード番号 269
ワークステーション VSAM キー順デ
ータ・セットのキー 268
READ ステートメントのオプション
203

KEYFROM 268

KEYFROM、相対レコード番号 269

KEYLENGTH オプション 206

KEYLOC オプション 207

KEYTO

順次レコード値 269
相対レコード番号 269
ワークステーション VSAM キー順デ
ータ・セットのキー 268
ワークステーション VSAM 順次デー
タ・セットにアクセスするための順
次ファイル 274

L

LANGLVL コンパイラ・オプション
61

LEAVE ENVIRONMENT オプション 13

LIBS コンパイラ・オプション 61

LIMITS コンパイラ・オプション 62

LINE オプション
コンソールへの出力の制御における
242

PRINT ファイル使用時 234

PRINT ファイルでの用法 234

PUT ステートメントの 227

LINECOUNT コンパイラ・オプション
63

LINEDIR コンパイラ・オプション 64

LINESIZE オプション
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 230
タブ設定テーブル・フィールド 238
定義 235

OPEN ステートメント 230

LINKAGE コンパイル時サブオプション
構文 50

パフォーマンスへの影響 347

呼び出し規則 387

LIST オプション、ILIB 用 384

LIST コンパイラ・オプション 64

LISTVIEW コンパイラ・オプション
64

LOCATE ステートメント 223

LOWERINC コンパイル時サブオプション
51

LRECL オプション 213

LRMSKIP オプション 213

M

MACRO コンパイラ・オプション 66

make ファイル・ユーティリティ
(NMAKE) 315

MARGINI コンパイラ・オプション 66

MARGINS コンパイラ・オプション
66

MAXGEN コンパイラ・オプション 67

MAXMSG コンパイラ・オプション 68

MAXNEST コンパイラ・オプション
68

MAXSTMT コンパイラ・オプション
69

MAXTEMP コンパイラ・オプション
69

MDECK コンパイラ・オプション 69

MSG コンパイラ・オプション 70

N

NAMES コンパイラ・オプション 70

NATIVE コンパイル時サブオプション
移植性の考慮事項 17

説明 49

パフォーマンスへの影響 347

NATIVEADDR コンパイル時サブオプシ
ョン 49

NATLANG

ランタイム・オプション 175

NATLANG コンパイラ・オプション
71

NCP ENVIRONMENT オプション 13

NEST コンパイラ・オプション 71

NMAKE ユーティリティ

インライン・ファイル 336

オプション 318

概要 315

構文 316

コマンド行の使用 316

コマンドを変更する文字 337

コマンド・ファイルの使用 317

推論規則 329

説明 321

ディレクティブ 332

特殊マクロ 326

NMAKE ユーティリティ (続き)
マクロの使用 323
make ファイル・ユーティリティ
(NMAKE) 316
TOOLS.INI ファイル 340
NOBACKUP オプション、ILIB 用 382
NOBINARG コンパイラー・サブオプション 51
NODESCRIPTOR コンパイル時サブオプション 48
NOEVENDEC コンパイル時サブオプション 50
NOEXTDICTIONARY オプション、ILIB 用 384
NOFREEFORMAT オプション、ILIB 用 383
NOINITFILL コンパイル時サブオプション 51
NOINLINE コンパイル時サブオプション 49
NOLAXDCL コンパイル時オプション 180
NOLAXIF コンパイル時オプション 180
NONASSIGNABLE コンパイル時サブオプション 48
NONCONNECTED コンパイル時サブオプション 48
NONNATIVE コンパイル時サブオプション 49
NONNATIVEADDR コンパイル時サブオプション 49
NONRECURSIVE コンパイル時サブオプション 52
NONULLSTRADDR コンパイラー・サブオプション 52
NOOVERLAP コンパイル時サブオプション
説明 50
NOT コンパイラー・オプション 72
NOT コンパイル時オプション
移植性 12
NOWARN オプション、ILIB 用 385
NOWRITE ENVIRONMENT オプション 13
NULL370 コンパイル時サブオプション 52
NULLSTRADDR コンパイラー・サブオプション 52
NULLSTRPTR コンパイラー・サブオプション 52
NULLSYS コンパイル時サブオプション 52
NUMBER コンパイラー・オプション 72
NUMBER コンパイル時オプション 144

O

OBJECT コンパイラー・オプション 73
ODBC 297
オンライン・ヘルプ 299
環境固有の情報 299
組み込み SQL 298
接続中 299
提供されるインクルード・ファイル 302
ドライバ・マネージャー 298
背景 297
利点 298
C データ型のマッピング 304
CALL インターフェースの規則 302
PL/I からの API の使用 301
OFFSET コンパイラー・オプション 73
ONSNAP コンパイラー・オプション 73
Open Database Connectivity (ODBC 参照) 297
OPEN ステートメント
使用
LINESIZE オプション 230
TITLE オプション 201, 221
ファイルのオープン 221
レコード長の指定 201
OPTIMIZE コンパイラー・オプション 74, 342
OPTIONS コンパイラー・オプション 74
OR コンパイラー・オプション 75
OR コンパイル時オプション
移植性 12
ORDER コンパイル時サブオプション
説明 50
パフォーマンスへの影響 346
ORDINAL コンパイル時サブオプション 50
ORGANIZATION オプション 207
OVERLAP コンパイル時サブオプション
説明 50

P

PACKAGE 対ネストされた
PROCEDURE 351
PAGE オプション
PRINT ファイルでの用法 234
PUT ステートメントの 227
PATH ランタイム環境変数 173
PLI コマンド
コンパイラーの呼び出し 33
コンパイル時オプションの指定 34
PLIDUMP
取得
ファイル情報 182
TCA 情報 182

PLIDUMP (続き)

推奨されるコーディング 183
説明 182
定様式 PL/I ダンプの読み取り 185
PLISRTx
使用するサブルーチンの決定 429
成功または失敗の伝達 430, 435
ソート・データの入出力 435
ソート・フィールドの指定 432
ソート・プログラムの呼び出し 434
入力および出力処理ルーチン 436
パラメーター 427
PLITABS 238
PL/I
コンパイラー
言語の使用方法が無効 190
ユーザー出口のプロシーチャー 358
標準ファイル 225
ファイル
構造体 27
コンパイルの準備 27
データ・セットとの関連付け 219
定義 197
ENVIRONMENT 属性
他の SAA インプリメンテーション
に移植可能なオプション 202
OPEN ステートメント 201
PL/I コード、コンパイル 417, 421, 425
PL/I コード、作成 415, 420, 424
PL/I コード、リンク 417, 421, 425
PP コンパイラー・オプション 75
PPCICS コンパイラー・オプション 76
PPINCLUDE コンパイラー・オプション 77
PPMACRO コンパイラー・オプション 77
PPSQL コンパイラー・オプション 78
PPTRACE コンパイラー・オプション 78
PPTRACE コンパイル時オプション 78
PRECTYPE コンパイラー・オプション 79
PREFIX コンパイラー・オプション 79, 343
デフォルトのサブオプションの使用 344
PREFIX コンパイル時オプション
デバッグ時に使用 180
PRINT ファイル
印刷する行の長さの制御 235
タブ制御テーブルの指定変更 237
端末でのフォーマット 242
ANS 印刷制御文字の挿入 234
PRINT 属性の適用 234
PROBE コンパイラー・オプション 80

PROCEED コンパイラー・オプション
80
PROMPT オプション 213
Proprinter、IBM、制御文字 204
PUT
ステートメント
コンソールからの入力の制御 240
属性およびオプション 222
FILE オプションを使用しないで
239
GRAPHIC オプション 206
DATA 181
LIST 181
SKIP LIST 181
PUT ステートメント
PAGE、SKIP、および LINE オプシ
ョン 227
PUTPAGE オプション 214

Q

QUOTE コンパイラー・オプション 81

R

READ ステートメント、属性およびオプ
ション 223
RECCOUNT オプション 214
RECORD I/O
制限 12
RECORD OUTPUT ファイル
連続データ・セット関連 204
CTLASA の使用 204
RECORD 条件
ワークステーション VSAM 用の
CONSECUTIVE ファイルの修正
271
ワークステーション VSAM 用の
INDEXED ファイルの修正 271
RECORD ファイル 207
RECSIZE オプション
ストリーム入出力用の 229
説明および構文 214
レコード長の指定 201
PL/I ENVIRONMENT 属性 208
RECURSIVE コンパイル時サブオプシ
ョン 52
REDUCE コンパイラー・オプション 82
REDUCIBLE 関数 352
REGIONAL ENVIRONMENT オプション
13
REGIONAL(1)
データ・セット
アクセスと更新 259
作成 257

REGIONAL(1) (続き)
データ・セット (続き)
順次アクセスの使用 259
説明 253
直接アクセスの使用 260
例 257
ファイル 271
ENVIRONMENT オプション 208
RELEASE ステートメント、例 371
REMOTE アクセス方式 201
REMOVE オブジェクト、ILIB 用 381
REORDER コンパイル時サブオプション
説明 50
パフォーマンスへの影響 346
REREAD ENVIRONMENT オプション
13
RESEXP コンパイラー・オプション 82
RESPECT コンパイラー・オプション
83, 478
RETCODE コンパイル時サブオプション
53
RETRY オプション 215
RETURNS コンパイル時サブオプション
53, 346
REWRITE ステートメント 223
REWRITE ステートメント、属性およびオ
プション 224
RULES コンパイラー・オプション 83,
479
パフォーマンスへの影響 343
RULES コンパイル時オプション
デバッグ時に使用 180

S

SAA サブオプション、LANGVLV の 61
SAA2 サブオプション、LANGVLV の
61
SAMELINE オプション 215
SCALARVARYING オプション 209
SEMANTIC コンパイラー・オプション
90
SEQUENTIAL
INPUT 203
OUTPUT 245
UPDATE 203
SEQUENTIAL ファイル
使用してワークステーション VSAM
直接データ・セットにアクセス 291
ワークステーション VSAM キー順デ
ータ・セットにアクセスするための
使用 282
ワークステーション VSAM 順次デー
タ・セットにアクセスするための使
用 274

SET コマンド
ランタイム環境変数 173
SHARE オプション 216
SHORT コンパイル時サブオプション 53
SIS ENVIRONMENT オプション 13
SKIP ENVIRONMENT オプション 13
SKIP オプション
コンソールからの入力の制御 241
PRINT ファイルでの用法 234
PUT ステートメントの 227
SKIP0 オプション 216
SMARTdata 公共機関 198
SNAP コンパイラー・オプション 90
パフォーマンスへの影響 342
SNAP コンパイル時オプション
ダンプ 180
デバッグ時に使用 180
メッセージ 188
SORT.DEF ファイル 369
SORT.PLI ファイル 369
SOSI コンパイラー・オプション 91
SOURCE コンパイラー・オプション 91
SQL ステートメント
INCLUDE 311
SQL プリプロセッサ 485
エラー戻りコード、処理 129
オプション 107
環境変数 31, 115
記述子域 116
標識変数の使用 127
ホスト構造体の使用 126
ホスト変数の使用 118
ユーザー定義関数 124
ラージ・オブジェクト・サポート 121
連絡域 115
EXEC SQL ステートメント 106
SQLCA 115
SQLDA 116
STATIC コンパイラー・オプション 91
STMT コンパイラー・オプション 92
STORAGE コンパイラー・オプション
92
STREAM I/O
制限 13
STREAM 属性
説明 228
データ・セット 229
SYNTAX コンパイラー・オプション 92
SYSIN ファイル
属性 239
標準入力のリダイレクト 225
SYSPARM コンパイラー・オプション
93
SYSPRINT ファイル
属性 239
標準出力のリダイレクト 225

SYSTEM

メッセージ 188

リンケージ、呼び出し規則 395

SYSTEM コンパイラー・オプション 93

T

TERMINAL コンパイラー・オプション
94

TERMLBUF オプション 217

TEST コンパイラー・オプション 94

TITLE オプション

使用

RECSIZE オプション 230

SYSPRINT および SYSIN ファイ
ル 225

説明 220

データ・セットに関連付けられていな
いファイルの使用 221

ファイルのオープンとクローズ 221

レコード長の指定 201

TMP 環境変数 33

TOTAL ENVIRONMENT オプション 13

TP ENVIRONMENT オプション 13

TRANSIENT ファイル属性 13

TRKOFL ENVIRONMENT オプション
13

TYPE オプション 217

レコード・フォーマットの指定 202

U

U フォーマット・レコード 202

UNDEFINEDFILE 条件

データ・セットに関連付けられていな
いファイルの使用 219

ファイル・オープン時に発生 219

UNLOCK ステートメント 12

UPPERINC コンパイル時サブオプション
51

USAGE コンパイラー・オプション 95

V

VSAM

ファイル、プログラムの修正 272

option 209

VSAM キー順データ・セットの基本ファ
イル 266

W

WARN オプション、ILIB 用 385

WIDECHAR コンパイラー・オプション
96

WINDOW コンパイラー・オプション
96, 478

WINDOW コンパイル時オプション 96

WRITE ステートメント、属性およびオブ
ション 223

X

XINFO コンパイラー・オプション 96

XML コンパイラー・オプション 99

XREF コンパイル時オプション

デバッグ時に使用 180

リスト内の出力 144

[特殊文字]

*PROCESS ステートメント 27

? オプション、ILIB 用 382

%INCLUDE ステートメント 28

%LINE ディレクティブ 29

%OPTION ディレクティブ 28

%PROCESS ステートメント

および PROCEDURE ステートメント
27

コンパイル時オプションの指定 35



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・プログラム 移行ガイド

GC88-9125

診断ガイド

SC88-9126

言語解説書

SC88-9127

コンパイル時メッセージおよびコード

SC88-4280-02



日本アイ・ビー・エム株式会社

〒103-8510 東京都中央区日本橋箱崎町19-21