

REXX for CICS Transaction Server
バージョン 1 リリース 1

ユーザーズ・ガイドおよびリファレンス



注記

本書および本書で紹介する製品をご使用になる前に、[479 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、REXX Development System for CICS® バージョン 1 リリース 1 (プログラム番号 5655-086)、REXX Runtime Facility for CICS バージョン 1 リリース 1 (プログラム番号 5655-087)、および新しい版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。

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：

REXX for CICS Transaction Server
Version 1 Release 1
User Guide and Reference

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

© Copyright International Business Machines Corporation 1974, 2020.

目次

この PDF について	xiii
第 1 部 REXX アプリケーションの開発	1
第 1 章 REXX のフィーチャーとコンポーネント	3
第 2 章 REXX アプリケーションの作成と実行	5
REXX 命令の構文	5
REXX 命令のフォーマット	6
REXX 命令の大文字と小文字	6
REXX 節のタイプ	8
2 バイト文字セットの名前を使用するプログラム	10
プログラムの入力	11
プログラムの実行	11
エラー・メッセージの解釈	12
大文字への変換の防止	13
プログラムへの情報の受け渡し	14
プログラム・スタックまたは端末入力装置からの情報の取得	14
プログラムを呼び出す際の値の指定	15
大文字への入力変換の防止	16
演習: ARG 命令の使用	16
引数の受け渡し	17
第 3 章変数および式を使用した変数データの処理	19
VARIABLES	19
式	21
算術演算子	21
比較演算子	24
論理 (ブール) 演算子	27
連結演算子	28
演算子の優先順位	29
第 4 章プログラム内のフローの制御	33
条件付き命令	33
IF...THEN...ELSE 命令	33
ネストされた IF...THEN...ELSE 命令	35
SELECT WHEN...OTHERWISE...END 命令	38
ループ命令	41
反復ループ	41
条件付きループ	44
複合ループ	48
割り込み命令	50
EXIT 命令	50
CALL 命令と RETURN 命令	51
SIGNAL 命令	54
第 5 章関数	57
組み込み関数	58
サブルーチンと関数	62
サブルーチンおよび関数の作成	64

内部または外部のサブルーチンまたは関数の使用の選択.....	71
情報の受け渡し.....	71
サブルーチンまたは関数からの情報の受信.....	78
第 6 章データの操作.....	81
複合変数およびステムの使用.....	81
複合変数とは.....	81
ステムの使用.....	82
演習: 複合変数およびステムの使用.....	82
データの解析.....	83
構文解析命令.....	83
ワードへの解析についての詳細.....	85
パターンによる解析.....	86
引数としての複数のストリングの構文解析.....	88
第 7 章プログラムからのコマンドの使用.....	91
コマンドでの引用符の使用.....	91
コマンドでの変数の使用.....	91
コマンドとしての別の REXX プログラムの呼び出し.....	92
プログラムからのコマンドの発行.....	92
コマンドがホスト環境に渡される仕組み.....	93
ホスト・コマンド環境の変更.....	93
第 8 章プログラム内の問題の診断.....	95
TRACE 命令による式のトレース.....	95
TRACE 命令によるコマンドのトレース.....	97
REXX の特殊変数 RC および SIGL の使用.....	97
対話式デバッグ機能を使用したトレース.....	98
対話式 TRACE 出力の保管.....	99
第 9 章 REXX/CICS ヘルプ・ユーティリティーの使用.....	101
第 10 章プログラミング・スタイルおよび手法.....	103
自己診断テスト.....	104
お楽しみ.....	104
プログラムの設計.....	106
ループの設計方式.....	106
結果.....	106
これまでの成果.....	106
段階的詳細化: 例.....	107
データの再検討.....	108
プログラムの修正.....	108
プログラムの変更.....	108
プログラムのトレース.....	108
コーディング・スタイル.....	109
第 2 部 REXX の構成.....	113
第 11 章 REXX サポートの構成.....	115
RFS ファイル・プールの作成.....	115
リソース定義の作成.....	115
LSRPOOL 定義の確認.....	116
CICSTART メンバーの更新.....	116
CICS 初期設定 JCL の変更.....	117
RFS ファイル・プールの形式設定.....	118
インストールの検証.....	119
ヘルプ・ファイルの作成.....	120

REXX Db2 インターフェースの構成.....	121
第 12 章 REXX/CICS システムの定義と管理.....	123
REXX/CICS 許可コマンドと許可コマンド・オプション.....	123
システム・プロファイル exec.....	123
MVS PDS REXX 許可ライブラリー.....	123
許可ユーザーの定義.....	124
システム・オプションの設定.....	124
REXX ファイル・システム (RFS) ファイル・プールの定義.....	124
CICSTART のための PLT エントリーの作成.....	124
セキュリティー出口.....	124
CICSECX1.....	124
CICSECX2.....	125
第 13 章パフォーマンスの考慮事項.....	127
第 14 章セキュリティー.....	129
REXX/CICS による複数トランザクション ID のサポート.....	129
REXX/CICS ファイル・セキュリティー.....	129
REXX/CICS コマンド・レベル・セキュリティー.....	129
REXX/CICS 許可コマンド・サポート.....	130
セキュリティー定義.....	130
第 3 部 REXX for CICS Transaction Server: リファレンス.....	133
第 15 章製品機能の概要.....	135
第 16 章構文図の読み方.....	139
第 17 章 REXX の一般的な概念.....	141
構造および一般的な構文.....	141
文字.....	142
コメント.....	142
トークン.....	143
暗黙指定されるセミコロンの.....	146
継続.....	147
式および演算子.....	147
式.....	147
演算子.....	148
括弧および演算子の優先順位.....	151
文節および命令.....	152
割り当ておよび記号.....	153
定数記号.....	154
単純記号.....	154
複合記号.....	154
語幹.....	155
外部環境に対するコマンド.....	156
CICS で実行される REXX の基本構造.....	157
標準 REXX 機能のサポート.....	160
REXX コマンド環境のサポート.....	161
標準 CICS 機能のサポート.....	161
他のプログラミング言語へのインターフェース.....	161
第 18 章キーワード命令.....	163
ADDRESS.....	163
ARG.....	164
CALL.....	165

DO	168
DROP	172
EXIT.....	172
IF	173
INTERPRET	174
ITERATE	175
LEAVE	175
NOP	176
NUMERIC.....	176
OPTIONS.....	177
PARSE	179
PROCEDURE	181
PULL	182
PUSH	183
QUEUE.....	184
RETURN	184
SAY	185
SELECT.....	185
SIGNAL.....	186
TRACE.....	187
UPPER.....	191
第 19 章関数.....	193
構文.....	193
関数およびサブルーチン.....	193
組み込み関数.....	196
ABBREV (省略形).....	196
ABS (絶対値).....	197
ADDRESS.....	197
ARG (引数).....	197
BITAND (ビットごとの AND).....	198
BITOR (ビットごとの OR).....	198
BITXOR (ビットごとの排他 OR).....	199
B2X (2 進数から 16 進数へ).....	199
CENTER/CENTRE.....	200
COMPARE.....	200
CONDITION.....	200
COPIES	201
C2D (文字から 10 進数へ).....	201
C2X (文字から 16 進数へ).....	202
DATATYPE.....	202
DATE.....	203
DBCS (2 バイト文字セット関数).....	204
DELSTR (ストリングの削除).....	205
DELWORD (ワードの削除).....	205
DIGITS	205
D2C (10 進数から文字へ).....	206
D2X (10 進数から 16 進数へ).....	206
ERRORTEXT	207
EXTERNALS	207
FIND	207
FORM	207
FORMAT.....	208
FUZZ	209
INDEX.....	209
INSERT.....	209
JUSTIFY.....	210
LASTPOS (最後の位置).....	210

LEFT	210
LENGTH.....	211
LINESIZE	211
MAX (最大).....	211
MIN (最小).....	211
OVERLAY	212
POS (位置).....	212
QUEUED	212
RANDOM	213
REVERSE	213
RIGHT	214
SIGN.....	214
SOURCELINE	214
SPACE	214
STORAGE.....	215
STRIP	215
SUBSTR (サブストリング)	215
SUBWORD	216
SYMBOL	216
TIME.....	216
TRACE.....	218
TRANSLATE	218
TRUNC (切り捨て).....	219
USERID.....	219
VALUE.....	219
VERIFY.....	220
WORD	221
WORDINDEX	221
WORDLENGTH	221
WORDPOS (ワード位置).....	222
WORDS	222
XRANGE (16 進数範囲).....	222
X2B (16 進数から 2 進数へ).....	222
X2C (16 進数から文字へ).....	223
X2D (16 進数から 10 進数へ).....	223
REXX/CICS で提供される外部関数.....	224
STORAGE.....	224
SYSSBA.....	224
第 20 章構文解析.....	227
いくつかのワードに構文解析するための単純形式のテンプレート.....	227
ストリング・パターンを含むテンプレート.....	229
定位置 (数値) パターンを含むテンプレート.....	230
変数パターンによる解析.....	233
UPPER の使用.....	234
解析命令の要約.....	234
解析命令の例.....	235
高度な解析に関する情報.....	236
複数のストリングの構文解析.....	236
ストリング・パターンと定位置パターンの組み合わせ: 特殊なケース.....	236
DBCS 文字の解析.....	237
解析ステップの詳細.....	237
第 21 章数値と算術演算.....	241
概要: 数値.....	241
算術機能の定義.....	242
数字	242
Precision (精度).....	242

算術演算子.....	243
算術演算規則: 基本演算子.....	243
算術演算規則: 加算演算子.....	244
数値の比較.....	245
指数表記.....	246
数値情報.....	248
整数.....	248
REXX によって直接使用される数値.....	248
エラー.....	248
第 22 章条件および条件トラップ.....	249
条件がトラップされないときのアクション.....	250
条件がトラップされたときのアクション.....	250
条件情報.....	252
特殊変数.....	252
第 23 章 REXX/CICS テキスト・エディター.....	255
コマンド行コマンド.....	258
ARBCHAR.....	258
ARGS.....	258
BACKWARD.....	259
BOTTOM.....	259
CANCEL.....	260
CASE	260
CHANGE.....	261
CMDLINE.....	262
CTLCHAR.....	262
CURLINE.....	263
DISPLAY.....	263
DOWN.....	264
EDIT.....	264
EXEC.....	265
FILE.....	266
FIND	267
FORWARD.....	268
GET.....	268
GETPDS.....	269
INPUT.....	269
JOIN.....	270
LEFT	270
LINEADD.....	271
LPREFIX.....	271
MACRO	271
MSGLINE.....	272
NULLS.....	273
NUMBERS.....	273
PFKEY.....	274
PFKLINE.....	274
QQUIT.....	275
QUERY.....	276
QUIT	277
RESERVED	277
RESET.....	278
RIGHT	278
SAVE.....	279
SORT.....	279
SPLIT.....	280
STRIP	280

SYNONYM	281
TOP.....	281
TRUNC	282
UP.....	282
第 24 章 REXX/CICS ファイル・システム.....	283
ファイル・プール、ディレクトリー、ファイル.....	283
現行ディレクトリーとパス.....	284
セキュリティ.....	285
RFS コマンド.....	285
AUTH	286
CKDIR.....	286
CKFILE.....	287
COPY.....	287
DELETE.....	288
DISKR	288
DISKW	289
GETDIR.....	289
MKDIR.....	290
RDIR.....	290
RENAME.....	290
REXX/CICS ファイル・リスト・ユーティリティー	291
呼び出し.....	291
REXX/CICS ファイル・リスト・ユーティリティーでのマクロ	292
FLST コマンド.....	292
FLST 戻りコード.....	298
FLST からの EXEC およびトランザクションの実行.....	298
第 25 章 REXX/CICS List System.....	299
ディレクトリーおよびリスト.....	299
現行ディレクトリーとパス.....	300
セキュリティ.....	300
RLS コマンド.....	300
CKDIR.....	300
DELETE.....	301
LPULL.....	301
LPUSH.....	302
LQUEUE.....	302
MKDIR.....	303
READ.....	303
VARDROP.....	304
VARGET.....	304
VARPUT.....	305
WRITE.....	305
第 26 章 REXX/CICS コマンド定義.....	307
背景.....	307
コマンドの定義.....	307
REXX プログラムに渡されるコマンド引数.....	308
アセンブラー・プログラムに渡されるコマンド引数.....	308
CICPARMS 制御ブロック.....	309
REXX 以外の言語インターフェース	310
CICGETV: REXX 変数を取得、設定、またはドロップするための呼び出し.....	310
第 27 章 REXX/CICS Db2 インターフェース.....	313
プログラミング上の考慮事項.....	313
SQL ステートメントの組み込み.....	313
結果の受信.....	315

SQL 連絡域の使用.....	316
SQL ステートメントの使用例.....	316
Db2 コマンドの埋め込み.....	317
結果の受信.....	318
Db2 コマンドの使用例.....	319
第 28 章 REXX/CICS ハイレベル・クライアント/サーバー・サポート.....	321
ハイレベルで、最適な透過的 REXX クライアント・インターフェース.....	321
REXX ベース・アプリケーションのクライアントおよびサーバーのサポート.....	321
クライアント/サーバー・コンピューティングでの REXX の値.....	321
REXX/CICS クライアント EXEC の例.....	322
REXX/CICS サーバー EXEC の例.....	322
第 29 章 REXX/CICS パネル機能.....	323
パネルの定義.....	324
.DEFINE verb を使用したフィールド制御文字の定義.....	324
.DEFINE.....	325
.PANEL verb を使用した実際の PANEL レイアウトの定義.....	328
.PANEL.....	329
パネル生成とパネル入出力.....	330
PANEL RUNTIME.....	331
PANEL 変数.....	334
パネル機能戻りコード情報.....	335
状態コードと入力コード.....	338
ロケーション・コード.....	341
サンプル・パネルの例.....	341
REXX パネル・プログラムの例.....	343
第 30 章 REXX/CICS コマンド.....	349
ALLOC.....	350
AUTHUSER.....	350
CD.....	351
CEDA.....	352
CEMT.....	353
CLD.....	367
CONVTMAP.....	368
COPYR2S.....	369
COPYS2R.....	371
C2S.....	373
DEFCMD.....	374
DEFSCMD.....	376
DEFTRNID.....	378
DIR.....	379
EDIT.....	380
EXEC.....	381
EXECDROP.....	382
EXECIO.....	383
EXECLOAD.....	384
EXECMAP.....	385
EXPORT.....	386
FILEPOOL.....	387
FLST.....	389
FREE.....	390
GETVERS.....	390
HELP.....	391
IMPORT.....	391
LISTCMD.....	393
LISTPOOL.....	393

LISTTRNID.....	394
PATH.....	394
PSEUDO.....	396
RFS.....	397
RLS.....	400
SCRNINFO.....	402
SET.....	403
SETSYS.....	405
S2C.....	407
TERMID.....	407
WAITREAD.....	408
WAITREQ.....	408
第 31 章エラー番号とメッセージ.....	411
CICREXnnn エラー・メッセージ.....	412
第 32 章戻りコード.....	423
パネル機能の戻りコード.....	423
SQL 戻りコード.....	423
Db2 戻りコード.....	423
RFS および FLST.....	424
EDITOR および EDIT.....	425
DIR.....	426
SET.....	426
CD.....	426
PATH.....	427
RLS.....	427
LISTCMD.....	428
CLD.....	428
DEFCMD.....	429
DEFSCMD.....	429
DEFTRNID.....	429
EXECDROP	430
EXECLOAD.....	430
EXECMAP	430
ALLOC および FREE.....	431
EXPORT および IMPORT.....	431
FILEPOOL.....	432
GETVERS.....	433
COPYR2S.....	433
COPYS2R.....	433
LISTPOOL.....	434
LISTTRNID.....	434
C2S.....	434
PSEUDO.....	434
AUTHUSER.....	434
SETSYS.....	435
S2C.....	435
TERMID.....	435
WAITREAD.....	435
WAITREQ.....	436
コマンド固有でない戻りコード.....	436
EXEC.....	436
CEDA および CEMT.....	437
EXECIO	450
CONVTMAP.....	450
SCRNINFO.....	450
CICS.....	450

第 33 章 2 バイト文字セット (DBCS) サポート.....	453
DBCS: 概要.....	453
DBCS データ操作および記号の使用可能化.....	454
シンボルとストリング.....	454
命令と DBCS.....	455
DBCS 関数の処理.....	457
組み込み関数の例.....	458
DBCS 処理関数.....	462
DBADJUST	462
DBBRACKET	462
DBCENTER	463
DBCJUSTIFY	463
DBLEFT	464
DBRIGHT	464
DBRLEFT	465
DBRRIGHT	465
DBTODBCS	465
DBTOSBCS	466
DBUNBRACKET	466
DBVALIDATE	466
DBWIDTH	467
第 34 章 予約キーワードおよび特殊変数.....	469
予約キーワード.....	469
特殊変数.....	469
第 35 章 デバッグ補助機能.....	471
プログラムの対話式デバッグ.....	471
実行への割り込みおよびトレースの制御.....	472
第 36 章 基本マッピング・サポートの例.....	473
第 37 章 参考文献.....	477
特記事項.....	479
索引.....	483

この PDF について

この PDF では、REXX/CICS または REXX for CICS Transaction Server について説明します。この IBM® プログラム・プロダクトでは、REXX/CICS に、ネイティブの REXX ベースのアプリケーション開発、カスタマイズ、プロトタイピング、およびプロシージャー型言語環境を、関連するランタイム機能とともに提供します。

REXX for CICS Transaction Server (REXX for CICS) は、REXX for CICS/ESA の新しい名前です。

- REXX Development System for CICS バージョン 1 リリース 1 は、REXX Development System for CICS/ESA バージョン 1 リリース 1 の新しい名前です。
- REXX Runtime Facility for CICS バージョン 1 リリース 1 は、REXX Runtime Facility for CICS/ESA バージョン 1 リリース 1 の新しい名前です。

バージョン、リリース、および製品番号に変更はありません。この資料における REXX for CICS に関する言及には、CICS との利用がサポートされるこの製品の以前のバージョンがすべて含まれ、依然として REXX for CICS/ESA と呼ばれます。

この PDF の対象読者

この PDF は、REXX for CICS Transaction Server の命令および関数を参照する必要があるユーザー、ならびに構文解析などの REXX 言語の項目についてさらに詳しく学習する必要があるユーザーを対象としています。また、REXX プログラムの記述方法を学習したい方すべてを対象としています。ユーザーのタイプには、アプリケーション・プログラマー、システム・プログラマー、エンド・ユーザー、管理者、開発者、テスター、およびサポート担当者が含まれます。

この PDF の概要

この PDF には、ユーザー・ガイドと参照資料の両方が含まれています。REXX アプリケーションの開発のセクションと REXX サポートの構成のセクションは、REXX for CICS Transaction Server に精通するために役立ちます。参照セクションには、REXX の命令、関数、およびコマンドが含まれています。命令、関数、およびコマンドは、それぞれのセクションでアルファベット順にリストされています。また、REXX でプログラムを作成するために認識しておく必要がある一般概念についても、詳しく説明しています。

本書で説明するプログラミング言語は、REstructured eXtended eXecutor (再構造化拡張実行プログラム) 言語 (一般に REXX) と呼ばれています。また、本書では、CICS Transaction Server の REXX 言語処理プログラム (ここ以降は、略して「言語処理プログラム」) が、再構造化拡張実行プログラム言語をどのように処理するのか、つまり解釈するのかについても説明しています。

この PDF の日付

この PDF は、2019 年 1 月 31 日に作成されました。

第 1 部 REXX アプリケーションの開発

このセクションでは、REXX プログラムとその構文について紹介し、REXX プログラムを作成および実行する際に必要な手順と、よくある問題を回避するために理解しておかなければならない概念について説明します。

REXX プログラムは、REXX インタープリターで直接解釈する REXX 言語で書かれた命令で構成されています。プログラムには、ホスト環境で実行されるコマンド (CICS コマンドなど) を含めることもできます ([91 ページの『第 7 章 プログラムからのコマンドの使用』を参照](#))。

第 1 章 REXX のフィーチャーとコンポーネント

REXX は、汎用プログラミング言語です。環境をホストするコマンドと混用でき、強力な機能を提供し、幅広い数学的機能を備えています。

REXX プログラムは、CICS の下で多くのタスクを実行できます。これには、CEDA (オンライン・リソース定義トランザクション) および CEMT (マスター端末トランザクション) ユーティリティーへのコマンドだけでなく、EXEC CICS コマンド、SQL ステートメントの発行も含まれます。

REXX のフィーチャー

REXX のフィーチャーのいくつかを以下に示します。

使いやすさ

REXX 言語は、多くの命令が意味のある英語のワードであるため、読み書きが容易です。REXX 命令は、SAY、PULL、IF...THEN...ELSE...、DO...END、EXIT などの一般的なワードです。

フリー・フォーマット

REXX フォーマットについての規則は、少数しかありません。命令を特定の桁から開始する必要はありません。行の中でスペースをスキップしたり、行全体をスキップしたりすることができます。多くの行にわたる 1 つの命令を書くことも、1 行に複数の命令を書くこともできます。変数を事前定義する必要はありません。命令は、大文字、小文字、または大/小文字混合で入力できます。[REXX 命令の構文](#)を参照してください。

組み込み関数

REXX は、さまざまな処理、検索、および比較演算をテキストと数値の両方に対して実行する組み込み関数を提供します。その他の組み込み関数では、フォーマット設定機能および算術計算を提供します。

デバッグ機能

REXX/CICS で実行されている REXX プログラムでエラーが発生した場合、REXX はエラーを示すメッセージを書き込みます。プログラム・エラーを見つけるために、REXX TRACE 命令や対話式デバッグ機能を使用することもできます。

インタープリター型言語

REXX/CICS 製品には、REXX/CICS インタープリターが組み込まれています。REXX プログラムが実行されると、インタープリターは各行を直接処理します。プログラムを実行する前に、プログラムをコンパイルしたり、リンク・エディットしたりする必要はありません。

幅広い構文解析機能

REXX には文字操作のための幅広い構文解析機能が組み込まれているため、文字、数値、および混合入力を区切るパターンをセットアップできます。

REXX のコンポーネント

REXX は、以下のコンポーネントから構成されており、プログラマーにとって強力なツールとなります。

節

節は、命令、ヌル節、またはラベルにすることができます。命令は以下のとおりです。

- キーワード命令
- 割り当て
- コマンド (REXX/CICS コマンド、CICS コマンド、および SQL)。

言語処理プログラムは、キーワード命令および割り当てを処理します。

組み込み関数

これらの関数は言語処理プログラムに組み込まれており、便利な処理オプションが提供されます。

外部関数

REXX/CICS では、REXX の特定のタスクを実行するためにシステムと対話するこれらの関数が提供されます。

データ・スタック関数

データ・スタックは、入出力および他のタイプの処理のためにデータを保管できます。

前提条件

REXX/CICS は、サポートされるすべてのリリースの CICS Transaction Server で実行されます。CICS TS で必要な前提条件以外の前提条件はありません。

ランタイム機能

REXX/CICS MVS™ ランタイム機能は、REXX/CICS MVS exec のランタイム環境を提供するので、REXX/CICS MVS 開発システムがインストールされていない CICS 領域で、REXX/CICS exec を実行できます。

これは、ある企業が複数の CICS 領域を使用しており、そのうちの特定の領域のみ REXX ベースの開発に使用できる場合に、これらの exec を実行する機能が必要な場合、特に役立ちます。また、完全な REXX/CICS MVS 開発システムのライセンスを購入しなくても、ランタイム機能を使用して、他の領域で REXX 前提条件を持つ製品を実行することもできます。

第 2 章 REXX アプリケーションの作成と実行

REXX 言語の利点の 1 つとしては、日常使用している英語に類似していることが挙げられます。この類似性により、以下の例で示すように、REXX プログラムの読み書きが容易になります。

単純なプログラムの例

1 行の出力を書き込むには、REXX 命令の SAY を使用し、出力するテキストを続けます。

```
/* Sample REXX Program */  
SAY 'Hello world!'
```

図 1. 例: *Hello World* プログラム

このプログラムは、これが REXX プログラムであることを識別するコメント行から始まっています。コメントは /* で開始され、*/ で終了します。

プログラムを実行すると、SAY 命令は以下の出力を端末装置に送信します。

```
Hello world!
```

より長いプログラムの例

これよりも長いプログラムでも、命令は日常使用している英語に類似しているため、容易に理解できます。以下の例で、2 つの数値を加算するプログラム ADDTWO を呼び出します。

1. CICS 端末で、画面をクリアして、以下のコマンドを入力します。

```
REXX addtwo
```

2. 2 つの数値を入力します。

以下に ADDTWO プログラムを示します。プログラム・コード内のコメントは、入力された最初の数値が 42 で、2 番目の数値が 21 であると想定しています。

```
/****** REXX *****/  
/* This program adds two numbers and produces their sum. */  
/****** REXX *****/  
say 'Enter first number.'  
PULL number1 /* Assigns: number1=42 */  
say 'Enter second number.'  
PULL number2 /* Assigns: number2=21 */  
sum = number1 + number2  
SAY 'The sum of the two numbers is' sum.'
```

図 2. 例: *ADDTWO* プログラム

上記のサンプル・プログラムを実行すると、最初の PULL 命令が変数 *number1* に値 42 を割り当てます。2 番目の PULL 命令が変数 *number2* に値 21 を割り当てます。次の行には、代入式が含まれています。言語処理プログラムは、*number1* と *number2* の値を加算し、結果 63 を *sum* に割り当てます。そして最後に、SAY 命令が以下の出力行を表示します。

```
The sum of the two numbers is 63.
```

REXX 命令の構文

一部のプログラミング言語には、各行に文字を入力する方法や場所に関して厳密な規則があります。例えば、アセンブラー言語では、ステートメントを特定の列で開始しなければなりません。一方、REXX の構文規則は単純です。大文字、小文字、または大/小文字混合を使用できます。REXX には、入力可能な列についての制約事項はありません。

命令は、任意の行の任意の列で開始できます。以下の命令はいずれも有効です。

```
SAY 'You can type in any column'
      SAY 'You can type in any column'
        SAY 'You can type in any column'
```

上記の命令は、以下のように端末出力装置に送信されます。

```
You can type in any column
You can type in any column
You can type in any column
```

REXX 命令のフォーマット

REXX 言語はフリー・フォーマットを使用します。つまり、ワードとワードの間に追加のスペースを挿入できるということです。

例えば、以下に示す式はすべて同じ意味を持ちます。

```
total=num1+num2
total =num1+num2
total = num1+num2
total = num1 + num2
```

また、プログラム全体にわたってブランク行を挿入してもエラーは発生しません。

REXX 命令の大文字と小文字

REXX 命令は、小文字、大文字、または大/小文字混合で入力できます。英字を単一引用符または二重引用符で囲まない限り、言語処理プログラムは英字を大文字に変換します。

例えば、SAY、Say、say はすべて同じ意味です。

命令での引用符の使用

引用符のペアで囲まれた一連の文字は、リテラル・ストリングになります。以下の例には、リテラル・ストリングが含まれています。

```
SAY 'This is a REXX literal string.' /* Using single quotation marks */
SAY "This is a REXX literal string." /* Using double quotation marks */
```

リテラル・ストリングを 2 種類の異なる引用符で囲むことはできません。以下は、正しくない例です。

```
SAY 'This is a REXX literal string.'" /* Using mismatched quotation marks */
```

SAY 命令に含まれるリテラル・ストリングを引用符で囲まないと、通常、言語処理プログラムはそのステートメントを大文字に変換します。その例を次に示します。

```
SAY This is a REXX string.
```

以下の結果になります。

```
THIS IS A REXX STRING.
```

(この例では、いずれのワードも、既に値が割り当てられている変数の名前ではないことを前提としています。REXX では、変数のデフォルト値は、その変数名の大文字の名前になります)。

アポストロフィが含まれるリテラル・ストリングは、二重引用符で囲むことができます。

```
SAY "This isn't difficult!"
```

また、単一引用符のペアは 1 つとして処理されるため、アポストロフィの代わりに 2 つの単一引用符を使用することもできます。

```
SAY 'This isn''t difficult!'
```

いずれの方法でも、同じ結果になります。

```
This isn't difficult!
```

命令の終了

行にセミコロン (;) が含まれているか、行がコンマ (,) で終わっていない限り、通常、1つの行には1つの命令が含まれます。

行の終わり、またはセミコロンは、命令の終了を意味します。1行に1つの命令を含める場合、行の終わりは命令の終わりを示します。1行に複数の命令を含める場合、隣接する命令の間をセミコロンで区切る必要があります。

```
SAY 'Hi!'; say 'Hi again!'; say 'Hi for the last time!'
```

この例の結果は、3行になります。

```
Hi!  
Hi again!  
Hi for the last time!
```

命令の継続

コンマは継続文字です。これは、命令が次の行に続くことを意味します。このようにコンマを使用すると、行を連結する際にスペースも追加されます。以下に、リテラル・ストリングを次の行に続ける場合に、継続文字としてのコンマがどのように機能するかを示します。

```
SAY 'This is an extended',  
  'REXX literal string.'
```

最初の行の終わりにあるコンマは、2つの行を連結して出力する際にスペースを (extended と REXX の間に) 追加します。その結果、以下の単一の行になります。

```
This is an extended REXX literal string.
```

以下の2つの命令はまったく同じもので、同じ結果になります。

```
SAY 'This is',  
  'a string.'
```

```
SAY 'This is' 'a string.'
```

2つの個別のストリングの間にあるスペースは保持されます。

```
This is a string.
```

スペースを追加しないリテラル・ストリングの継続

命令を2行以上の複数の行に続ける一方、REXXが行にスペースを追加しないようにする必要がある場合は、連結オペランド (2つの OR の縦棒、||) を使用します。

```
SAY 'This is an extended literal string that is bro'||,  
  'ken in an awkward place.'
```

上記の例は、ワード「broken」にスペースが含まれることなく、1行になります。

```
This is an extended literal string that is broken in an  
awkward place.
```

以下の2つの命令もまったく同じもので、同じ結果になることに注意してください。

```
SAY 'This is' ||,  
  'a string.'  
  
SAY 'This is' || 'a string.'
```

これらの例は、以下の結果になります。

```
This isa string.
```

どちらの例でも、連結演算子によって 2 つのストリングの間のスペースが削除されます。

以下の例に、REXX のフリー・フォーマットを示します。

```
/****** REXX ******/
SAY 'This is a REXX literal string.'
SAY      'This is a REXX literal string.'
  SAY 'This is a REXX literal string.'
SAY,
'This',
'is',
'a',
'REXX',
'literal',
'string.'

SAY 'This is a REXX literal string. '; SAY 'This is a REXX literal string.'
SAY '      This is a REXX literal string.'
```

図 3. フリー・フォーマットの例

この例を実行すると、同じ 6 つの行が出力され、その後に字下げされた 1 行が続きます。

```
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
      This is a REXX literal string.
```

命令は行の任意の場所で開始できます。また、空白行を挿入することも、命令に含まれるワードとワードの間に追加スペースを挿入することもできます。言語処理プログラムは、空白行も、複数のスペースも無視します。このようなフォーマットの柔軟性により、空白行やスペースを挿入してプログラムを読みやすくすることができます。

空白とスペースは、構文解析時にのみ意味を持ちます。[83 ページの『データの解析』](#)を参照してください。

REXX 節のタイプ

REXX 節は、命令、ヌル節、およびラベルにすることができます。命令には、キーワード命令、代入命令、またはコマンドを使用できます。

以下に、3 つのタイプの文節を使用したサンプル・プログラムを示します。このサンプル・プログラムに続いて、文節の各タイプについて説明します。

```
/* QUOTA REXX program. Two car dealerships are competing to */
/* sell the most cars in 30 days. Who will win?                */

store_a=0; store_b=0
DO 30
  CALL sub
END
IF store_a>store_b THEN SAY "Store_a wins!"
ELSE IF store_b>store_a THEN SAY "Store_b wins!"
ELSE SAY "It's a tie!"
EXIT

sub:
store_a=store_a+RANDOM(0,20) /* RANDOM returns a random number in */
store_b=store_b+RANDOM(0,20) /* in specified range, here 0 to 20 */
RETURN
```

キーワード命令

キーワード命令は、言語処理プログラムに何らかの処理を行うよう指示します。キーワード命令は、言語処理プログラムに行わせる処理を識別する REXX キーワードで始まります。例えば、キーワード DO を使用すると、複数の命令をグループ化して、それらの命令を繰り返し実行できます。キーワード IF は、条件が満たされているかどうかをテストします。キーワード SAY は、現行の端末出力装置に書き込みます。

IF、THEN、および ELSE の 3 つのキーワードは、1 つの命令の中で連動します。それぞれのキーワードが文節の形を取り、命令のサブセットとなります。IF キーワードの後に続く式が真であれば、THEN キーワードの後に続く命令が処理されます。そうでなければ、ELSE キーワードの後に続く命令が処理されます。(ELSE 節を THEN と同じ行に配置する場合は、ELSE の前にセミコロンが必要であることに注意してください)。THEN または ELSE の後に複数の命令を続ける場合は、それらの命令のグループの前と後にそれぞれ DO と END を使用します。IF 命令について詳しくは、[33 ページの『条件付き命令』](#)を参照してください。

EXIT キーワードは、言語処理プログラムに対し、プログラムの終了を指示します。上記のサンプル・プログラムでは、EXIT を使用する必要があります。そうしないと、言語処理プログラムがラベル sub: の後に続くサブルーチン内のコードを実行してしまうためです。EXIT が必要にならないプログラムもありますが (例えば、サブルーチンを使用しないプログラムなど)、このキーワードを含めることが適切なプログラミング手法と言えます。EXIT について詳しくは、[50 ページの『EXIT 命令』](#)を参照してください。

割り当て

代入命令は、値を変数に割り当てるか、変数の現行値を変更します。以下に、単純な代入命令を示します。

```
number = 4
```

上記のサンプル・プログラムでは、単純な代入命令 store_a=0 が使用されています。代入命令の左側 (等号の前) は、右側 (等号の後) の値を受け取る変数の名前です。右側は実際の値 (4 など) または式にできます。式とは、評価する必要があるものを指します (演算式など)。式には数値、変数、またはその両方を含めることができます。

```
number = 4 + 4
```

```
number = number + 4
```

最初の例では、number の値は 8 になります。プログラムの最初の例のすぐ後に続く 2 番目の例では、number の値は 12 になります。式について詳しくは、[21 ページの『式』](#)を参照してください。

ラベル

ラベルは、sub: のように、シンボル名の後にコロンが続く形になります。ラベルには、1 バイト文字、2 バイト文字、または 1 バイト文字と 2 バイト文字の組み合わせを含めることができます。(2 バイト文字は、OPTIONS ETMODE がプログラムの最初の命令である場合にのみ有効です)。ラベルはプログラムの一部を識別するもので、一般にサブルーチンおよび関数の中で、SIGNAL 命令とともに使用されます。(制御をメインプログラムに戻すために、サブルーチンの最後に RETURN 命令を組み込む必要があることに注意してください)。ラベルの使用について詳しくは、[62 ページの『サブルーチンと関数』](#)および [54 ページの『SIGNAL 命令』](#)を参照してください。

ヌル節

ヌル節は、ブランクのみ、またはコメント、あるいはその両方で構成されます。言語処理プログラムはヌル節を無視しますが、ヌル節を使用するとプログラムが読みやすいものになります。

コメント

コメントは /* で開始され、*/ で終了します。コメントは 1 行以上にするこも、行の一部にするこもできます。コメントには、REXX 命令を読む人にとって、明らかではない可能性がある情報を含めることができます。プログラムの先頭のコメントで、プログラム全体の目的を説明することや、場合によっては特殊な考慮事項をリストすることもできます。個々の命令の隣にコメントを含めて、その命令の目的を明確にできます。

注: REXX/CICS では、REXX プログラムがコメントで始まることを要件としていません。ただし、移植性の理由から、各 REXX プログラムをコメントで開始して、そのコメントに REXX という語を含めるこ

とを推奨します。すべての言語処理プログラムに、このプログラム ID が必要になるわけではありません。ただし、MVS TSO と CICS で同じ EXEC を実行する場合は、`/* REXX */` プログラム ID を含めて TSO 要件を満たす必要があります。

ブランク行

ブランク行は、命令のグループを分離して、プログラムを読みやすくします。プログラムが読みやすければ、プログラムの理解や保守もそれだけ容易になります。

コマンド

コマンドとは、1つの式のみからなる1つの文節です。コマンドは、事前に定義された環境に送信されて処理されます。(式の中に評価すべきではない部分がある場合、その部分を引用符で囲んでください)。前のサンプル・プログラムには、コマンドが1つも含まれていませんでした。以下の例では、ADDRESS 命令にコマンドが含まれています。

```
/* REXX program including a command */
ADDRESS REXXCICS 'DIR'
```

ADDRESS は、キーワード命令です。ADDRESS 命令で環境およびコマンドを指定すると、指定した環境に単一のコマンドが送信されます。この場合、環境は REXXCICS です。コマンドは、環境の後に続く次の式です。

```
'DIR'
```

DIR コマンドは、現行ファイル・システム・ディレクトリー内のファイルをリストします。ホスト・コマンド環境の変更方法について詳しくは、93 ページの『[ホスト・コマンド環境の変更](#)』を参照してください。コマンドの発行について詳しくは、91 ページの『[第7章 プログラムからのコマンドの使用](#)』を参照してください。

2 バイト文字セットの名前を使用するプログラム

REXX プログラムでは、リテラル・ストリング、記号、およびコメントに、2 バイト文字セット (DBCS) の名前を使用できます。このような文字ストリングは、1 バイト、2 バイト、またはこの両方の組み合わせにすることができます。DBCS の名前を使用するには、プログラム内の最初の命令を `OPTIONS ETMODE` にする必要があります。これにより、DBCS 文字が含まれるストリングの妥当性を言語処理プログラムによって検査することを指定します。

DBCS 文字は、シフトアウト (SO) およびシフトイン (SI) 区切り文字で囲む必要があります。(SO 文字は `X'OE'`、SI 文字は `X'OF'` です。) SO 文字および SI 文字は印刷できません。以下の例では、より小さい (<) 記号とより大きい (>) 記号がシフトアウト (SO) とシフトイン (SI) をそれぞれ表しています。例えば、以下の例の `<.S.Y.M.D>` および `<.D.B.C.S.R.T.N>` は、DBCS 記号を表しています。

例

以下に、DBCS 変数名および DBCS サブルーチン・ラベルを使用するプログラムの例を示します。

```
/* REXX */
OPTIONS 'ETMODE'          /* ETMODE to enable DBCS variable names */
<.S.Y.M.D> = 10           /* Variable with DBCS characters between */
                           /* shift-out (<) and shift-in (>) */
y.<.S.Y.M.D> = JUNK
CALL <.D.B.C.S.R.T.N>     /* Call subroutine with DBCS name */
EXIT
<.D.B.C.S.R.T.N>:        /* Subroutine with DBCS name */
DO i = 1 TO 10
  IF y.i = JUNK THEN      /* Does y.i match the DBCS variable's value? */
    SAY 'Value of the DBCS variable is : ' <.S.Y.M.D>
END
RETURN
```


プログラムの入力

このタスクについて

以下のプログラムを入力する場合、REXX ファイル・システム (RFS) に保管されているファイルについては REXX/CICS エディターを使用してください。MVS 区分データ・セット (PDS メンバー) にファイルを保管できる任意のエディターを使用することもできます。ここでは、REXX/CICS エディターを使用することを想定しています。

このプログラムの名前は HELLO EXEC です (今は、ファイル・タイプを exec と仮定します)。

1. CESN と入力し、要求に応じてユーザー ID とパスワードを入力することによって、REXX for CICS 端末にサインオンします。
2. 画面をクリアします。
3. 以下を入力します。

```
edit hello.exec
```

4. プログラムを、 `/* REXX HELLO EXEC */` で始めて、[11 ページの図 4](#) で示されるように正確に入力します。次に、EDIT コマンドを使用して、プログラムをファイルします。

```
====> file
```

これでプログラムを実行する準備ができました。

```
/* REXX HELLO EXEC */  
  
/* A conversation */  
say "Hello! What is your name?"  
pull who  
if who = "" then say "Hello stranger!"  
else say "Hello" who
```

図 4. HELLO EXEC

プログラムの実行

このタスクについて

exec を実行する前に、画面をクリアします。EXEC のファイル・タイプのプログラムを実行したい場合は、REXX の後に、そのファイル名を入力します。この場合、`rexx hello` をコマンド行に入力して Enter (キー) を押します。試してみましょう。

あなたの名前は Sam だとします。sam と入力して、Enter キーを押します。Hello SAM が表示されます。

```
rexx hello  
Hello! What is your name?  
sam  
Hello SAM
```

以下で具体的に説明します。

1. SAY 命令は Hello! What is your name? を表示します。
2. PULL 命令は、応答を待つためにプログラムを一時停止します。
3. ユーザーは sam をコマンド行に入力してから Enter (キー) を押します。
4. PULL 命令は、SAM というワードを who と呼ばれる変数 (コンピューターのストレージ内の場所) に入れます。

5. IF 命令は、who が空に等しいか確認します。

```
who = ""
```

これは、「who に保管された値は空に等しいかどうか」という意味です。明らかにするために、REXX は変数名の代わりに保管された値を使用します。したがって、現在の質問は、「SAM は空に等しいか」です。

```
"SAM" = ""
```

6. いいえ、真ではありません。then の後の命令は処理されません。代わりに、REXX は else の後の命令を処理します。
7. SAY 命令は、"Hello" who と表示します。これは、以下のように評価されます。

```
Hello SAM
```

ここで、最初に応答を入力しないで Enter (キー) を押したらどうなるでしょうか。

```
hello
Hello! What is your name?

Hello stranger!
```

入力を期待される反面、多分、ユーザーは自分の名前を入力する必要があるとわからないでしょう。(おそらくプログラムは、名前の部分をクリアする必要があります。)いずれにせよ、名前を入力する代わりに Enter (キー) を押すだけの場合は以下ようになります。

1. PULL 命令は、"" (空) を who という名前のコンピューターのストレージ内の場所に入れます。
2. 再度、IF 命令は変数を検査します。

```
who = ""
```

意味: who の値は空に等しいかどうか。who の値が置換された場合、この検査は、次の内容になります。

```
"" = ""
```

今回は、検査が真になります。

3. したがって、then の後の命令が実行され、else の後の命令は実行されません。

エラー・メッセージの解釈

エラーが含まれるプログラムを実行すると、多くの場合はエラー・メッセージにエラーが発生した行が示され、そのエラーについての説明が提供されます。エラー・メッセージが出力される原因としては、構文エラーや計算誤差が考えられます。

例

以下のプログラムには構文エラーがあります。

```
/****** REXX *****/
/* This REXX program contains a deliberate error of not closing */
/* a comment. Without the error, it would pull input to produce */
/* a greeting. */
/*******/

PULL who                                /* Get the person's name.
IF who = '' THEN
    SAY 'Hello, stranger'
ELSE
    SAY 'Hello,' who
```

図 5. 構文エラーがあるプログラムの例

上記のプログラムが実行されると、言語処理プログラムは以下の出力行を送信します。

```
7 +++ PULL who /* Get the person's name. IF who =  
' ' THEN SAY 'Hello, stranger' ELSE SAY 'Hello,' who  
CICREX453E Error 6 running HELLO EXEC, line 7: Unmatched "/" or quote
```

コメントの終わりに `*/` が欠落しているエラーを言語処理プログラムが検出するまで、プログラムは実行されます。PULL 命令の行には構文エラーが含まれているため、PULL 命令はデータ・スタックまたは端末からのデータを使用しません。プログラムが終了し、言語処理プログラムがエラー・メッセージを送信します。

最初のエラー・メッセージは、言語処理プログラムがエラーを検出したステートメントの行番号で始まります。その後、3つの正符号(+++)とステートメントの内容が続きます。

```
7 +++ PULL who /* Get the person's name. IF who =  
' ' THEN SAY 'Hello, stranger' ELSE SAY 'Hello,' who
```

2番目のエラー・メッセージは、メッセージ番号で始まります。その後、プログラム名、言語処理プログラムがエラーを検出した行、エラーの説明を含むメッセージが続きます。

```
CICREX453E Error 6 running HELLO EXEC, line 7: Unmatched  
"/" or quote
```

このプログラムの構文エラーを修正するには、行7のコメントの最後に `*/` を追加します。

```
PULL who /* Get the person's name. */
```

大文字への変換の防止

通常、言語処理プログラムは英字を大文字に変換してから処理します。大文字への変換を防止できます。

プログラム内の文字

プログラム内の英字が大文字に変換されないようにするには、単にその文字を単一引用符または二重引用符で囲みます。言語処理プログラムは、引用符で囲まれているかどうかに関わらず数字と特殊文字については変更しません。英字、数字、特殊文字が混在する句を指定した SAY 命令を使用する場合、言語処理プログラムは英字のみを変更します。

```
SAY The bill for lunch comes to $123.51!
```

以下の結果になります。

```
THE BILL FOR LUNCH COMES TO $123.51!
```

この例では、いずれのワードも、他の値が割り当て済みの変数の名前ではないことを前提としています。

引用符によって、プログラム内の情報が入力されたとおりに処理されるようにすることができます。この点は、以下の場合に重要となります。

- 出力が小文字または大/小文字混合でなければならない場合。
- コマンドが正しく処理されることを確実にする場合。例えば、プログラム内の変数名がコマンド名と同じである場合、そのコマンドが発行されると、プログラムがエラーで終了する可能性があります。コマンドと同じ名前の変数名を使用しないこと、およびすべてのコマンドを引用符で囲むことが適切なプログラミング手法といえます。

プログラムに入力される文字

別のプログラムから入力を読み取ったり、入力を渡したりする場合、言語処理プログラムは、それらを処理する前に英字を大文字に変更します。大文字に変換されないようにするには、PARSE 命令を使用します。

例えば、以下のプログラムは端末から入力を読み取り、その情報を端末出力装置に送信します。

```
/****** REXX *****/
/* This REXX program gets the name of an animal from the input */
/* stream and sends it to the terminal. */
/******/

PULL animal          /* Get the animal name.*/
SAY animal
```

例えば、入力が `tyrannosaurus` の場合、言語処理プログラムは以下の結果を出力します。

```
TYRANNOSAURUS
```

言語処理プログラムが、入力されたとおりに読み取るようにするには、PULL 命令の代わりに PARSE PULL 命令を使用します。

```
PARSE PULL animal
```

入力が `TyRann0sauRus` の場合は、以下の出力になります。

```
TyRann0sauRus
```

演習 - サンプル・プログラムの実行と変更

上記の例を作成して実行することができます。PULL 命令を PARSE PULL 命令に変更して、結果の違いを確認してください。

注: 文字 `@ # $ ¢` は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料で使用されている表記規則および用語](#)も参照してください。

プログラムへの情報の受け渡し

プログラムを実行する際に、プログラムに情報を渡すには、いくつかの方法があります。

このタスクについて

- PULL を使用して、プログラム・スタックまたは端末入力装置から情報を取得し、それをプログラムに渡すことができます。
- プログラムの呼び出し時に入力を指定できます。

プログラム・スタックまたは端末入力装置からの情報の取得

PULL 命令は、プログラムが入力を受け取るための方法の 1 つです。

このタスクについて

入力の行を区切ることで、PULL 命令は端末から複数の値を同時に取り出すことができます。

例

以下の ADDTWO プログラムでは、PULL を使用して、2 つの入力数値を受け取ります。

```
/****** REXX *****/
/* This program adds two numbers and produces their sum. */
/******/
PULL number1
PULL number2
sum = number1 + number2
SAY 'The sum of the two numbers is' sum'.'
```

図 6. PULL を使用するプログラムの例

以下の例は、PULL 命令で入力の行を区切って使用することによって、端末から複数の値を同時に取り出す方法を示すバリエーションになっています。

```
/****** REXX ******/
/* This program adds two numbers and says their sum */
/****** REXX ******/
PULL number1 number2
sum = number1 + number2
SAY 'The sum of the two numbers is' sum'.'
```

図 7. PULL を使用する例のバリエーション

プログラムを呼び出すには、CICS 端末から以下のコマンドを発行します。

```
REXX addtwo 42 21
```

上記の PULL 命令は、端末から数値 42 と 21 を取り出します。

プログラムを呼び出す際の値の指定

プログラムで入力を受け取るには、プログラムを呼び出す際に、値を指定するという方法もあります。

例えば、42 と 21 の 2 つの数値を ADD という名前のプログラムに渡すには、以下の CICS コマンドを使用できます。

```
REXX add 42 21
```

以下の例に示すように、ADD プログラムは、ARG 命令を使用して入力を変数に割り当てます。

```
/****** REXX ******/
/* This program receives two numbers as input, adds them, and */
/* produces their sum. */
/****** REXX ******/
ARG number1 number2
sum = number1 + number2
SAY 'The sum of the two numbers is' sum'.'
```

図 8. ARG 命令を使用するプログラムの例

ARG は最初の数値 42 を number1 に割り当て、2 番目の数値 21 を number2 に割り当てます。

値の数が ARG または PULL の後に続く変数名の数より少ない場合、またはそれより多い場合は、エラーが発生する可能性があります。これについて、以降のセクションで説明します。

指定する値の数が少なすぎる場合

指定する値の数が、PULL または ARG の後に続く変数の数より少ないと、余分な変数はヌル・ストリングに設定されます。以下の例では、プログラムに 1 つの数値のみを渡します。

```
REXX add 42
```

言語処理プログラムは、値 42 を ARG に続く最初の変数 number1 に割り当てます。2 番目の変数 number2 にはヌル・ストリングを割り当てます。この状態では、プログラムは、2 つの変数を追加しようとしたときにエラーで終了します。その他の状態では、プログラムがエラー終了しない場合があります。

指定する値の数が多すぎる場合

指定する値の数が、PULL または ARG の後に続く変数の数より多いと、最後の変数に残りの値が取り込まれます。例えば、以下のようにプログラム ADD に 3 つの数値を渡します。

```
REXX add 42 21 10
```

言語処理プログラムは、値 42 を ARG に続く最初の変数 `number1` に割り当てます。値 21 10 を 2 番目の変数 `number2` に割り当てます。この状態では、プログラムは、2 つの変数を追加しようとしたときにエラーで終了します。その他の状態では、プログラムがエラー終了しない場合があります。

最後の変数に残りの値が取り込まれないようにするには、PULL または ARG 命令の最後にピリオド (.) を使用します。

```
ARG number1 number2 .
```

ピリオドは、不要な余剰情報を収集するダミー変数として機能します。この場合、`number1` は 42 を受け取り、`number2` は 21 を受け取り、ピリオドによって 10 は破棄されます。余剰情報がない場合、ピリオドは無視されます。PULL または ARG 命令内で、以下のようにピリオドをプレースホルダーとして使用することもできます。

```
ARG . number1 number2
```

この例の場合、最初の値 42 は破棄され、`number1` と `number2` が次の 2 つの値 21 と 10 を受け取ります。

大文字への入力変換の防止

PULL 命令と同様に、ARG 命令は英字を大文字に変換します。大文字に変換されないようにするには、PARSE ARG 命令を使用します。

例

以下の例は、PARSE ARG を使用して大文字に変換されないようにする方法を示しています。

```
/****** REXX ******/
/* This program receives the last name, first name, and score of */
/* a student and reports the name and score.                      */
/*******/
PARSE ARG lastname firstname score
SAY firstname lastname 'received a score of' score'.'
```

図 9. PARSE ARG を使用するプログラムの例

演習: ARG 命令の使用

このタスクについて

左側の列に、プログラムに送信された入力値を表示します。右側の列は、その入力を受信する、プログラム内の ARG 命令です。各変数が受け取る値は何ですか。

	入力	入力を受け取る変数
1	115 -23 66 5.8	ARG first second third
2	.2 0 569 2E6	ARG first second third fourth
3	13 13 13 13	ARG first second third fourth fifth
4	Weber Joe 91	ARG lastname firstname score
5	Baker Amanda Marie 95	PARSE ARG lastname firstname score
6	Callahan Eunice 88 62	PARSE ARG lastname firstname score .

応答

1. first = 115、second = -23、third = 66 5.8
2. first = .2、second = 0、third = 569、fourth = 2E6
3. first = 13、second = 13、third = 13、fourth = 13、fifth = null

- 4. lastname = WEBER、firstname = JOE、score = 91
- 5. lastname = Baker、firstname = Amanda、score = Marie 95
- 6. lastname = Callahan、firstname = Eunice、score = 88

引数の受け渡し

一般に、プログラムに渡される値は、引数と呼ばれます。引数は、1つのワードまたは複数のワードからなるストリングで構成することができます。引数を複数のワードで構成する場合は、ワードとワードの間を空白で区切ります。渡される引数の数は、プログラムを呼び出す方法によって異なります。

このタスクについて

CALL 命令または REXX 関数呼び出しのいずれかを使用して REXX プログラムを呼び出す場合、プログラムに最大 20 個の引数を渡すことができます。各引数はコンマで区切ります。

関数およびサブルーチンについて詳しくは、[62 ページの『サブルーチンと関数』](#)を参照してください。
引数について詳しくは、[88 ページの『引数としての複数のストリングの構文解析』](#)を参照してください。

第 3 章 変数および式を使用した変数データの処理

コンピューター・プログラミングの最も強力な側面の 1 つは、結果を得るために変数データを処理できることです。

このタスクについて

プロセスの複雑さに関係なく、データが不明な場合や変化する場合は、そのデータを記号で置き換えます。記号は、その値が変化する場合に変数と呼ばれます。計算して解決しなければならない記号または数値のグループは、式と呼ばれます。

VARIABLES

変数とは、値を表す文字または文字のグループのことです。変数には 1 バイト文字または 2 バイト文字、あるいはその両方を含めることができます。

2 バイト文字は、OPTIONS ETMODE がプログラムの最初の命令である場合にのみ有効です。以下の変数 *big* は、値 1,000,000 を表します。

```
big = 1000000
```

変数は、場合に応じて異なる値を参照できます。*big* に別の値を割り当てると、値が再び変更されるまでは、その新しく割り当てられた値を取得します。

```
big = 999999999
```

プログラムを作成する時点では不明な値を、変数で表すこともできます。以下の例では、ユーザーの名前が不明であるため、変数 *who* でユーザー名を表しています。

```
PARSE PULL who      /* Gets name from current input stream */  
                    /* and puts it in variable "who" */
```

変数名

変数名は、変数を表す部分です。変数名は常に代入ステートメントの左側に位置し、右側にその変数の値自体が置かれます。

以下の例では、変数名は *variable1* です。

```
variable1 = 5  
SAY variable1
```

上記の代入ステートメントの結果、言語処理プログラムは *variable1* に値 5 を割り当て、SAY により以下の結果が出力されます。

```
5
```

変数名は以下の文字から構成できます。

A - Z

大文字の英字

a - z

小文字の英字

0 - 9

数

? ! , _

特殊文字

X'41' - X'FE'

2 バイト文字セット (DBCS) の文字

注: 2 バイト文字は、OPTIONS ETMODE がプログラムの最初の命令である場合にのみ有効です。

以下の制約事項が、変数名に適用されます

- 先頭文字を 0 から 9 の数字またはピリオド (.) にすることはできません。
- 変数名は 250 バイトを超えてはなりません。DBCS 文字が含まれる名前の場合、各 DBCS 文字を 2 バイトとしてカウントし、シフトアウト (SO) およびシフトイン (SI) をそれぞれ 1 バイトとしてカウントします。
- DBCS 名に含まれる DBCS 文字は、SO (X'0E') と SI (X'0F') で区切る必要があります。さらに、
 - SO と SI を隣接させることはできません。
 - SO または SI をネストすることは許可されません。
 - DBCS 名に DBCS ブランク (X'4040') を含めることはできません。
- 変数名を REXX の特殊変数である RC、SIGL、または RESULT にすることはできません。[特殊変数](#)を参照してください。

以下の名前は、許容される変数名の例です。

```
ANSWER ?98B A Word3 number the_ultimate_value
```

また、OPTIONS ETMODE がプログラム内の最初の命令である場合は、以下の DBCS 変数名も有効です。ここで、< はシフトアウトを表し、> はシフトインを表し、X、Y、および Z は DBCS 文字を表します。小文字と数字は、それ自体を表します。

```
<.X.Y.Z> number_<.X.Y.Z> <.X.Y>1234<.Z>
```

変数値

変数の値 (変数名が表す値) は、次のように分類できます。

- **定数**。定数は以下の形で表現されます。

整数 (12)

10 進数 (12.5)

浮動小数点数 (1.25E2)

符号付き数値 (-12)

ストリング定数 ('12')

- **ストリング**。ストリングは 1 つ以上のワードからなり、以下のように引用符で囲まれる場合と囲まれない場合があります。

```
This value can be a string.  
'This value is a literal string.'
```

- **別の変数からの値**。以下に例を示します。

```
variable1 = variable2
```

上記の例では、*variable1* が *variable2* の値に変更されますが、*variable2* は同じままです。

- **式**。以下のように、計算する必要があるものです。

```
variable2 = 12 + 12 - .6      /* variable2 becomes 23.4 */
```

変数に値が割り当てられる前は、変数名が大文字に変換された値が、その変数の値になります。例えば、変数 `new` に値が割り当てられていない場合、

```
SAY new
```

によって以下が生成されます。

```
NEW
```

演習: 有効な変数名の識別

以下の変数名のうち、有効な REXX 変数名はどれですか。

1. eight
2. \$25.00
3. MixedCase
4. nine_to_five
5. 結果

応答

1. 先頭文字が数字であるため、正しくありません。
2. 先頭文字が通貨記号 (\$) であるため、正しくありません。
3. 有効
4. 有効
5. 有効。ただし、これは、サブルーチンからの結果を受け取る場合にのみ使用できる特殊変数名です。

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料](#)で使用されている表記規則および用語も参照してください。

式

式とは、評価する必要があるものを指します。式は、数値、変数、またはストリングと、ゼロ以上の演算子で構成されます。演算子によって、数値、変数、ストリングに対して行う評価の種類が決まります。演算子には、算術、比較、論理、連結の 4 種類があります。

算術演算子

算術演算子は、有効な数値定数、または有効な数値定数を表す変数に作用します。

数値定数のタイプ

12

整数には小数点またはコンマがありません。NUMERIC DIGITS 命令を使用してデフォルトを指定変更しない限り、整数の算術演算の結果には、デフォルトで最大 9 桁の数字を含めることができます。NUMERIC DIGITS 命令について詳しくは、[NUMERIC](#) を参照してください。以下に、整数の例を示します。

```
123456789 0 91221 999
```

12.5

10 進数には小数点が含まれます。10 進数での算術演算の結果は、小数点の前および後の桁数を合わせて最大 9 桁 (NUMERIC DIGITS のデフォルト) に制限されます。以下に、10 進数の例を示します。

```
123456.789 0.888888888
```

1.25E2

指数表記の**浮動小数点数**は、科学計算表記であると言えます。E の後の数値は、小数点が移動する桁数を表します。したがって、1.25E2 (1.25E+2 と表記されることもあります) では小数点が右に 2 桁移動

するため、125 になります。E の後にマイナス符号 (-) が続いている場合、小数点は左に移動します。例えば、1.25E-2 は .0125 です。

浮動小数点数を使用することで、非常に大きい値または非常に小さい値を表すことができます。科学計算表記法 (浮動小数点数) について詳しくは、[指数表記](#)を参照してください。

-12

数値の横に負符号 (-) がある**符号付き値**は、負の値を表します。数値の横に正符号 (+) がある符号付き値は、正の値を表します。数値に符号がない場合、その数値は正の値として処理されます。

算術演算子

Operator (演算子)

意味

+

Add

-

減算

*

乗算

/

除算

%

除算し、余りなしの整数を返します。

//

除算し、余りのみを返します。

**

数値を整数指数に累乗します。

- number

(接頭部 -) 減算 0 - number と同じです。

+number

(接頭部 +) 加算 0 + number と同じです。

数値定数と算術演算子を使用して、以下のような演算式を作成できます。

```
7 + 2      /* result is 9 */
7 - 2      /* result is 5 */
7 * 2      /* result is 14 */
7 ** 2     /* result is 49 */
7 ** 2.5   /* result is an error */
```

除算

除算を表す演算子は 3 つあることに注意してください。それぞれの演算子が異なる方法で除算式の結果を計算します。

/

除算し、おそらく 10 進数を解として表現します。以下に例を示します。

```
7 / 2      /* result is 3.5 */
6 / 2      /* result is 3 */
```

%

除算し、整数を解として表現します。余りは無視されます。以下に例を示します。

```
7 % 2      /* result is 3 */
```

//

除算し、余りのみを解として表現します。以下に例を示します。

```
7 // 2      /* result is 1 */
```

計算の順序

1つの演算式で複数の演算子を使用する場合、数値と演算子の順序が極めて重要になることがあります。例えば、以下の式では、言語処理プログラムはどの演算子を最初に実行するのでしょうか。

```
7 + 2 * (9 / 3) - 1
```

言語処理プログラムは左から右へと進み、以下のように式を評価します。

- 最初に、括弧内の式を評価します。
- 次に、優先度の高い演算子を使用した式を評価してから、優先度の低い演算子を使用した式を評価します。

以下に、算術演算子の優先度を高い順から示します。

表 1. 算術演算子の優先度	
演算子記号	演算子の説明
- +	接頭演算子
**	累乗 (指数)
* / % //	乗算および除算
+ -	加算および減算

したがって、上記の例は次の順で評価されることになります。

1. 括弧内の式

```
7 + 2 * (9 / 3) - 1
           ㄥ___/
           3
```

2. 乗算

```
7 + 2 * 3 - 1
    ㄥ___/
    6
```

3. 加算および減算 (左から右への順)

```
7 + 6 - 1 = 12
```

演算式の使用

プログラムでは、さまざまな方法で演算式を使用できます。以下の例では複数の算術演算子を使用して、値 (ドルとセントの金額) を丸め、余分な小数点以下の桁を除去しています。

```

/***** REXX *****/
/* This program computes the total price of an item including sales */
/* tax, rounded to two decimal places. The cost and percent of the */
/* tax (expressed as a decimal number) are passed to the program */
/* when you run it. */
/*****/

PARSE ARG cost percent_tax

total = cost + (cost * percent_tax)      /* Add tax to cost. */
price = ((total * 100 + .5) % 1) / 100    /* Round and remove extra */
                                         /* decimal places. */
SAY 'Your total cost is $'price'.'

```

図 10. 演算式の使用例

演習: 演算式の計算

以下のプログラムが生成する出力行はどれですか。

```

/***** REXX *****/
pa = 1
ma = 1
kids = 3
SAY "There are" pa + ma + kids "people in this family."

```

以下の式の値はいくつになるでしょうか。

1. $6 - 4 + 1$
2. $6 - (4 + 1)$
3. $6 * 4 + 2$
4. $6 * (4 + 2)$
5. $24 \% 5 / 2$

応答

1. この家族は 5 人家族です。
2. 値は以下のとおりです。
 - a. 3
 - b. 1
 - c. 26
 - d. 36
 - e. 2

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料で使用されている表記規則および用語](#)も参照してください。

比較演算子

比較演算子では、数値またはストリングを比較し、評価を行うことができます。比較演算子を使用する式は、演算式とは異なり、数値を返しません。比較式は、真を表す 1、または偽を表す 0 のいずれかを返します。

一般的な比較を以下に示します。

- 項が等しいかどうか。 ($A = Z$)
- 最初の項が 2 番目の項より大きいかどうか。 ($A > Z$)
- 最初の項が 2 番目の項より小さいかどうか。 ($A < Z$)

例えば、 $A = 4$ 、 $Z = 3$ である場合、上記の比較式の結果は次のようになります。

- $(A = Z) 4 = 3$ ですか? 0 (偽)
- $(A > Z) 4 > 3$ ですか? 1 (真)
- $(A < Z) 4 < 3$ ですか? 0 (偽)

一般的に使用される比較演算子

Operator (演算子)

意味

=

等しい

==

厳密に等しい

\=

等しくない

\==

厳密に等しくない

>

より大きい

<

より小さい

><

より大またはより小 (等しくないと同じ)

>=

より大きいか等しい

\<

より小さくない

<=

より小さいか等しい

\>

より大きくない

注: NOT 文字 (¬) は、バックスラッシュ (¥) と同義です。使用可能かどうかや個人的な好みに応じて、この2つの文字のどちらを使用することもできます。この資料では、バックスラッシュ (¥) 文字を使用します。

厳密な等号演算子と等号演算子

2つの式が**厳密に等しい**場合、空白や大/小文字 (式が文字の場合) を含め、すべてのものが完全に同じであることを意味します。

2つの式が**等しい**場合、これらの式は同じ結果に解決されます。以下の式はすべて真です。

```
'WORD' = word      /* returns 1 */
'word ' ¥== word /* returns 1 */
'word' == 'word'   /* returns 1 */
4e2 ¥== 400 /* returns 1 */
4e2 ¥= 100 /* returns 1 */
```

比較式の使用

多くの場合、IF...THEN...ELSE 命令では比較式を使用します。以下の例では、IF...THEN...ELSE 命令を使用して2つの値を比較しています。この命令について詳しくは、[33 ページの『IF...THEN...ELSE 命令』](#)を参照してください。

```

/***** REXX *****/
/* This program compares what you paid for lunch for two */
/* days in a row and then comments on the comparison. */
/*****

PARSE PULL yesterday /* Gets yesterday's price from input stream */
PARSE PULL today /* Gets today's price */
IF today > yesterday THEN /* lunch cost increased */
    SAY "Today's lunch cost more than yesterday's."
ELSE /* lunch cost remained the same or decreased */
    SAY "Today's lunch cost the same or less than yesterday's."

```

図 11. 比較式の使用例

演習: 比較式の使用

上記の比較式の使用例に基づくと、言語処理プログラムは以下の昼食代から、どのような結果を生成するでしょうか。

昨日の昼食代

今日の昼食代

4.42

3.75

3.50

3.50

3.75

4.42

以下の式はどのような結果になりますか (0 または 1)。

1. "Apples" = "Oranges"
2. "Apples" = "Apples"
3. " Apples" == "Apples"
4. 100 = 1E2
5. 100 ¥= 1E2
6. 100 ¥== 1E2

応答

1. 言語処理プログラムは、以下の文を生成します。
 - a. Today's lunch cost the same or less than yesterday's.
 - b. Today's lunch cost the same or less than yesterday's.
 - c. Today's lunch cost more than yesterday's.
2. 式の結果は次のとおりです。0 は偽、1 は真を意味します。
 - a. 0
 - b. 1
 - c. 0 (最初の " Apples" にはスペースが含まれているため)
 - d. 1
 - e. 0
 - f. 1

論理 (ブール) 演算子

論理式を処理すると、比較式の場合と同じように 1 (真) または 0 (偽) が返されます。論理演算子は、2 つの比較式を結合し、比較式の結果に応じて 1 または 0 を返します。

論理演算子

Operator (演算子)

意味

&

AND

両方の比較式が真の場合は、1 を返します。以下に例を示します。

```
(4 > 2) & (a = a) /* true, so result is 1 */
(2 > 4) & (a = a) /* false, so result is 0 */
```

|

包含 OR

少なくとも一方の比較式が真の場合は、1 を返します。以下に例を示します。

```
(4 > 2) | (5 = 3) /* at least one is true, so result is 1 */
(2 > 4) | (5 = 3) /* neither one is true, so result is 0 */
```

&&

排他 OR

いずれか一方の比較式 (両方ではない) が真であれば 1 を返します。以下に例を示します。

```
(4 > 2) && (5 = 3) /* only one is true, so result is 1 */
(4 > 2) && (5 = 5) /* both are true, so result is 0 */
(2 > 4) && (5 = 3) /* neither one is true, so result is 0 */
```

接頭部 **~,¬**

論理 NOT

否定して反対の応答を返します。以下に例を示します。

```
~ 0 /* opposite of 0, so result is 1 */
~ (4 > 2) /* opposite of true, so result is 0 */
```

論理式の使用

論理式は、複雑な条件付き命令で使用したり、不要な条件を除外するためのチェックポイントとして使用したりできます。一連の論理式を使用する場合は、明確にするために、各式を 1 つ以上の括弧のセットで囲む必要があります。

```
IF ((A < B) | (J < D)) & ((M = Q) | (M = D)) THEN ....
```

以下の例では、論理演算子を使用して決定を行っています。

```

/***** REXX *****/
/* This program receives arguments for a complex logical expression */
/* that determines whether a person should go skiing. The first    */
/* argument is a season and the other two can be 'yes' or 'no'.      */
/*****

PARSE ARG season snowing broken_leg

IF ((season = 'WINTER') | (snowing = 'YES')) & (broken_leg = 'NO')
  THEN SAY 'Go skiing.'
ELSE
  SAY 'Stay home.'
```

図 12. 論理式の使用例

この例に渡される引数が **SPRING YES NO** である場合、IF 節は以下のように変換されます。

```

IF ((season = 'WINTER') | (snowing = 'YES')) & (broken_leg = 'NO') THEN
  false / true / true
  false / /
  true / /
  true /
  true
```

このプログラムを実行すると、以下の結果になります。

```
Go skiing.
```

演習: 論理式の使用

大学への出願を考えている学生が、以下の仕様に従って出願する大学を評価することにしました。

```

IF (inexpensive | scholarship) & (reputable | nearby) THEN
  SAY "I'll consider it."
ELSE
  SAY "Forget it!"
```

大学の授業料は高額ではないものの、奨学金を提供していません。また、評判は高いものの、通学距離が 1000 マイルを超えています。この学生は出願すべきでしょうか。

ANSWER

あり。この条件付き命令は、以下の結果になります。

```

IF (inexpensive | scholarship) & (reputable | nearby) THEN ...
  true / false / true / false
  true / /
  true / true
  true
```

連結演算子

連結演算子は、2つの項を1つに結合します。結合する項は、ストリング、変数、式、または定数にすることができます。連結は、出力のフォーマット設定において重要になることがあります。

2つの項を結合する方法を指示する演算子には、次のものがあります。

Operator (演算子)

意味

ブランク

複数の項を連結し、項と項の間に1つのブランクを挿入します。複数のブランクで項を区切ると、それらは単一のブランクになります。以下に例を示します。

```
SAY true blue /* result is TRUE BLUE */
```

||

項と項の間にブランクを挿入せずに、複数の項を連結します。以下に例を示します。

```
(8 / 2)|| (3 * 3) /* result is 49 */
```

隣接

項と項の間にブランクを挿入せずに、複数の項を連結します。以下に例を示します。

```
per_cent '%' /* if per_cent = 50, result  
is 50% */
```

隣接を使用できるのは、連結する項のタイプが異なる場合 (例えば、リテラル・ストリングと記号)、またはコメントのみが2つの項を区切る場合に限られます。

連結演算子の使用

出力をフォーマット設定する1つの方法は、以下の例のように、変数と連結演算子を使用することです。

```
/****** REXX *****/  
/* This program formats data into columns for output. */  
/******  
sport = 'base'  
equipment = 'ball'  
column = ' '  
cost = 5  
  
SAY sport||equipment column '$'  
cost
```

図 13. 連結演算子の使用例

この例の結果は、以下のとおりです。

```
baseball      $5
```

より洗練された方法で情報をフォーマット設定する場合は、構文解析とテンプレートを使用します。83 ページの『データの解析』を参照してください。

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。CICS 資料で使用されている表記規則および用語も参照してください。

演算子の優先順位

1つの式で複数のタイプの演算子を使用されている場合、言語処理プログラムはすべての演算子を含む全体の優先順位に従います。

以下に、優先順位の高い順に演算子を記載します。

表 2. 全体的な演算子の優先順位	
演算子記号	演算子の説明
¥ または -- +	接頭演算子
**	累乗 (指数)
* / % //	乗算および除算
+ -	加算および減算
ブランク 隣接	連結演算子
== > < など	比較演算子

表 2. 全体的な演算子の優先順位 (続き)	
演算子記号	演算子の説明
&	論理 AND
&&	包含 OR および排他 OR

例

以下の式には複数のタイプの演算子が含まれています。

```
IF (A > 7**B) & (B < 3)
```

値は以下のとおりであるとします。

A = 8

B = 2

C = 10

言語処理プログラムはこの例を以下のように評価します。

- 最初の括弧内の式を評価します。
 - A を 8 に評価します。
 - B を 2 に評価します。
 - 7**2 に評価します。
 - 8 > 49 を偽 (0) として評価します。
- 次の括弧内の式を評価します。
 - B を 2 に評価します。
 - 2 < 3 を真 (1) として評価します。
- 0 & 1 は 0 であると評価します。

演習: 演算子の優先順位

- 以下の例の解はどのようなになるでしょうか。
 - 22 + (12 * 1)
 - 6 / -2 > (45 % 7 / 2) - 1
 - 10 * 2 - (5 + 1) // 5 * 2 + 15 - 1
- 前の 27 ページの『[論理 \(ブール\) 演算子](#)』の演習で取り上げた学生と大学の例で、学生の式から括弧を取り除くと、大学の評価はどのような結果になるでしょうか。

```
IF inexpensive | scholarship & reputable | nearby THEN
  SAY "I'll consider it."
ELSE
  SAY "Forget it!"
```

大学の授業料は高額ではないものの、奨学金を提供していません。また、評判は高いものの、通学距離が 1000 マイルであることを思い出してください。

応答

- 結果は次のようになります。
 - 34 (22 + 12 = 34)
 - 1 (真) (3 > 3 - 1)
 - 32 (20 - 2 + 15 - 1)
- I'll consider it.

& 演算子には以下のような優先順位がありますが、結果は、括弧を取り除く前のバージョンと同じです。

```
IF inexpensive | scholarship & reputable | nearby THEN
  ¥-----/ ¥-----/ ¥-----/ ¥-----/
    true      false      true      false
  ¥ ¥-----/ /
  ¥ false /
  ¥-----/ /
    true
  ¥-----/
    true
```


第4章 プログラム内のフローの制御

一般に、プログラムが実行されると、最初の命令から最後の命令までが1つずつ実行されます。指示しない限り、言語処理プログラムは命令を順次実行します。REXX では、言語処理プログラムに特定の命令をスキップさせる命令、他の命令を繰り返させる命令、または制御をプログラムの別の部分に移す命令を使用して、プログラム内での実行順序を変更できます。

このタスクについて

REXX プログラムの順次実行を変更する REXX 命令は、以下のように分類できます。

条件付き命令

条件付き命令は、式という形で1つ以上の条件を設定します。条件が真であれば、言語処理プログラムはその条件に続くパスを選択します。そうでなければ、言語処理プログラムは別のパスを選択します。REXX の条件付き命令は次のとおりです。

```
IF expression THEN...ELSE  
SELECT WHEN expression ...OTHERWISE...END
```

ループ命令

ループ命令は、言語処理プログラムに命令一式を繰り返すよう指示します。ループは、指定した回数だけ繰り返すことも、条件を使用して繰り返しを制御することもできます。REXX のループ命令は次のとおりです。

```
DO repetitor ...END  
DO WHILE expression ...END  
DO UNTIL expression ...END
```

割り込み命令

割り込み命令は、言語処理プログラムに対し、永久または一時的に、プログラムから完全に抜け出るか、プログラムのある部分から別の部分に移るよう指示します。REXX の割り込み命令は次のとおりです。

```
EXIT  
SIGNAL label  
CALL label ...RETURN
```

条件付き命令

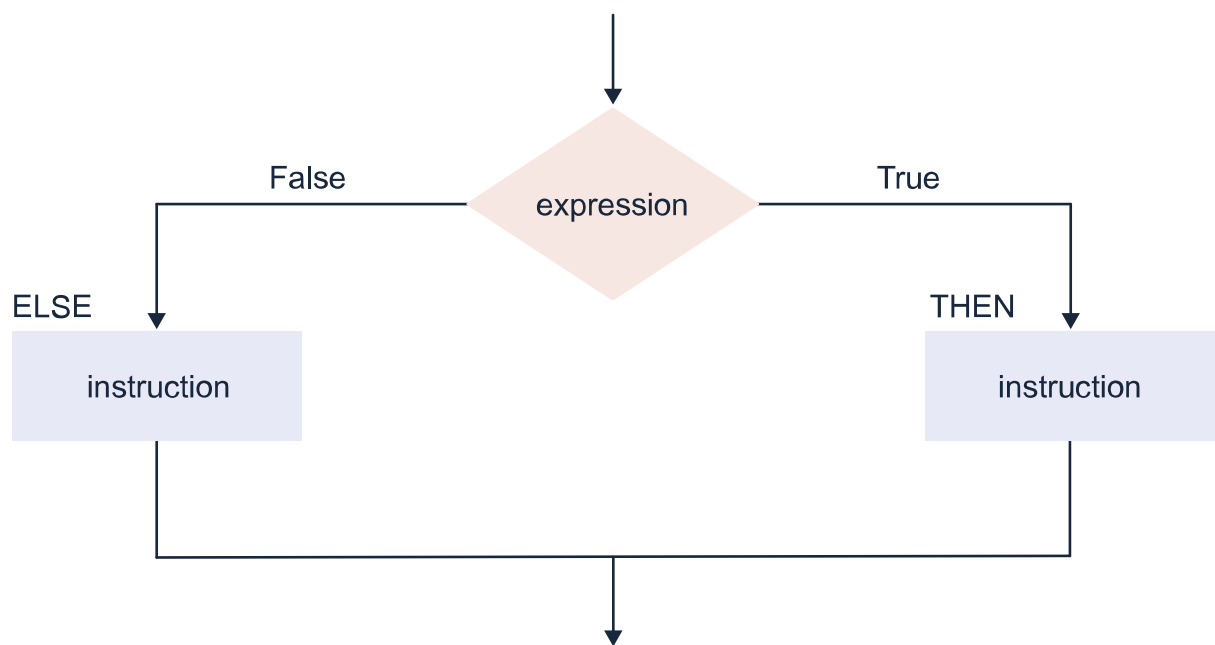
条件付き命令には、次の2つのタイプがあります。

- IF...THEN...ELSE では、プログラムの実行を、2つのオプションのいずれかに送信できます。
- SELECT WHEN...OTHERWISE...END では、プログラムの実行を多数のオプションのいずれか1つに送信できます。

IF...THEN...ELSE 命令

コード例は、IF...THEN...ELSE 命令の指定方法を学習するために役立ちます。

以下のフローチャートは、IF...THEN...ELSE 命令を示しています。



REXX 命令では、コードは以下ようになります。


```
IF expression THEN instruction
    ELSE instruction
```

読みやすさを向上させるために、文節を次のいずれかの方法で配置することもできます。

```
IF expression THEN
    instruction
ELSE
    instruction
```

または

```
IF expression
    THEN
        instruction
    ELSE
        instruction
```

命令全体を 1 行にする場合は、ELSE の前にセミコロンを入れて、THEN 節と ELSE 節を区切る必要があります。

```
IF expression THEN instruction; ELSE instruction
```

一般に、THEN 節と ELSE 節の後には少なくとも 1 つの命令が続ける必要があります。どちらの文節の後にも命令が続かない場合は、文節の横に NOP (ノーオペレーション) を含めることが適切なプログラミング手法です。

```
IF expression THEN
    instruction
ELSE NOP
```

条件に対して複数の命令がある場合は、それらの命令一式を DO で開始して END で終了します。

```
IF weather = rainy THEN
    SAY 'Find a good book.'
ELSE
    DO
        PULL playgolf /* Gets data from input stream */
        If playgolf='YES' THEN SAY 'Fore!'
    END
```

DO と END で囲まないと、言語処理プログラムは、ELSE 節の命令が 1 つのみであると想定します。

ネストされた IF...THEN...ELSE 命令

コード例は、IF...THEN...ELSE 命令の指定方法を学習するために役立ちます。

場合によっては、1 つまたは複数の IF...THEN...ELSE 命令を別の IF...THEN...ELSE 命令の中で使用しなければならないことがあります。同じタイプの命令を別の命令に含めることを、「ネストする」と表現します。IF 命令をネストする場合、各 IF を 1 つの ELSE に対応させること、そして各 DO を 1 つの END に対応させることが重要です。

```
IF weather = fine THEN
    DO
        SAY 'What a lovely day!'
        IF tenniscourt = free THEN
            SAY 'Let's play tennis!'
        ELSE NOP
    END
ELSE
    SAY 'We should take our raincoats!'
```

ネストされた IF と ELSE および DO と END が一致しないと、意外な結果になる場合があります。以下の例のように、DO と END、および ELSE NOP を除外すると、どのような結果になるでしょうか。

```

/***** REXX *****/
/* This program demonstrates what can happen when you do not include */
/* DOs, ENDS, and ELSEs in nested IF...THEN...ELSE instructions.    */
/***** REXX *****/
weather = 'fine'
tenniscourt = 'occupied'

IF weather = 'fine' THEN
  SAY 'What a lovely day!'
  IF tenniscourt = 'free' THEN
    SAY 'Let's play tennis!'
ELSE
  SAY 'We should take our raincoats!'

```

図 14. 命令が欠落している例

このプログラムを見ると、ELSE は最初の IF に属していると想定することでしょう。しかし、言語処理プログラムは、ELSE を最も近くの対応する ELSE がない IF に関連付けます。結果は、次のようになります。

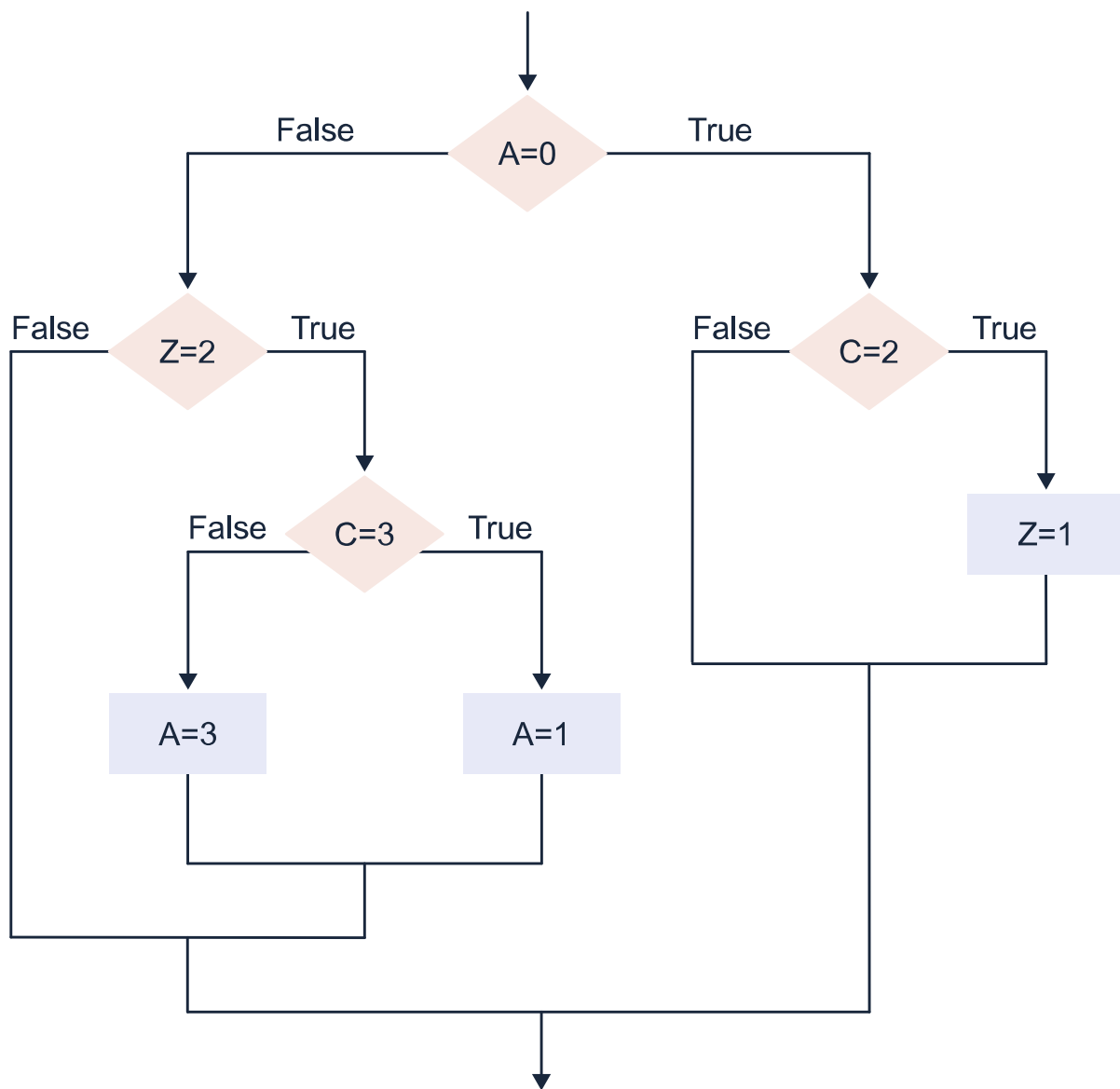
```

What a lovely day!
We should take our raincoats!

```

演習: IF...THEN...ELSE 命令の使用

以下のフローチャートに従った REXX 命令を作成してください。



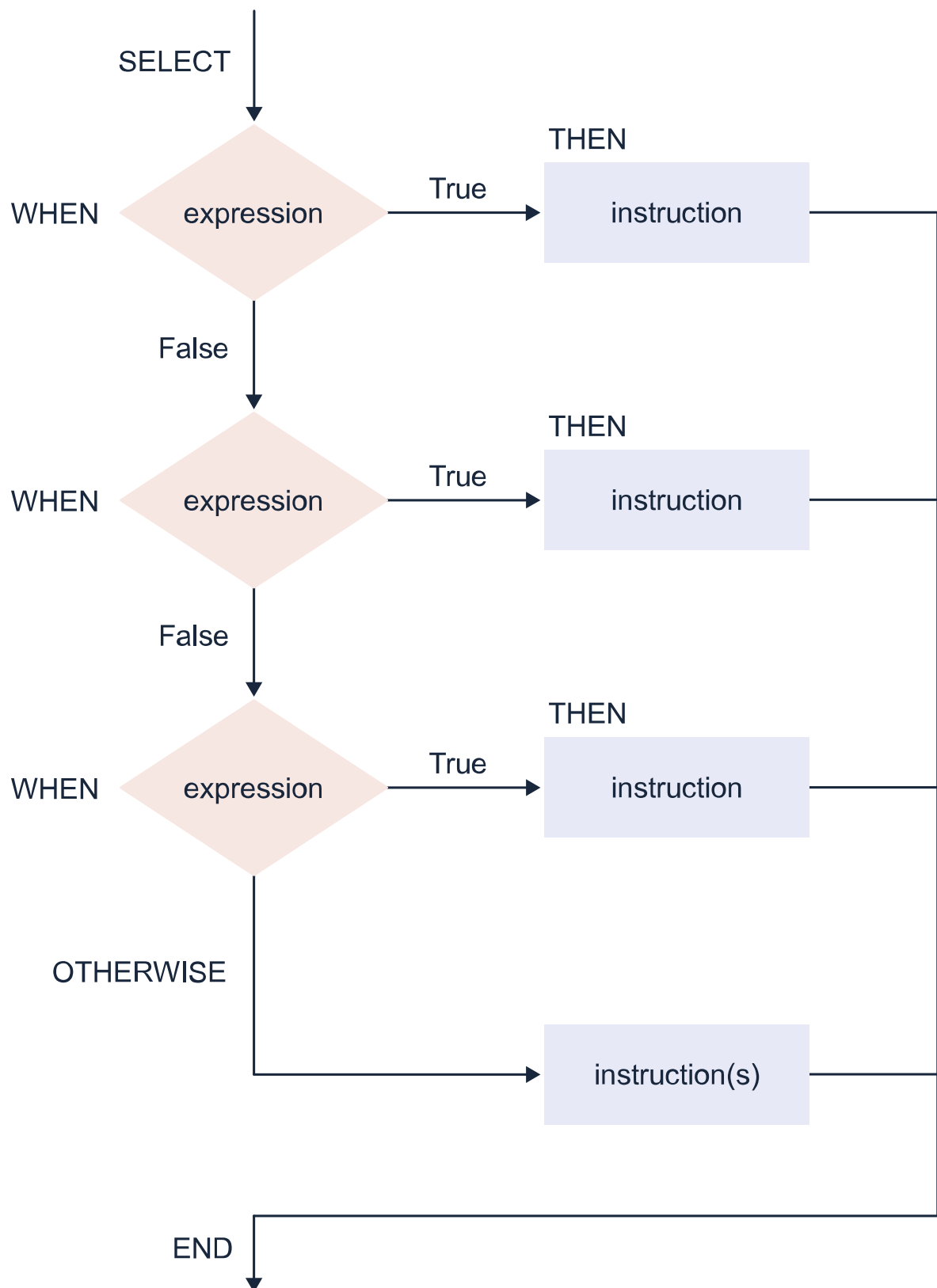
ANSWER

```
IF a = 0 THEN
  IF c = 2 THEN
    z = 1
  ELSE NOP
ELSE
  IF z = 2 THEN
    IF c = 3 THEN
      a = 1
    ELSE
      a = 3
    ELSE NOP
```

SELECT WHEN...OTHERWISE...END 命令

コード例は、SELECT WHEN...OTHERWISE...END 命令の指定方法を学習するために役立ちます。

任意の数の選択項目のうちの 1 つを選択するには、SELECT WHEN...OTHERWISE...END 命令を使用します。
これは、フローチャートでは以下のように表されます。



REXX 命令では、このフローチャートの例は次のようになります。

```

SELECT
  WHEN expression THEN instruction
  WHEN expression THEN instruction
  WHEN expression THEN instruction
  :
  :
  OTHERWISE
    instruction(s)
END

```

言語処理プログラムは、真の式が見つかるまで、WHEN 節を先頭からスキャンします。真の式を見つけると、他にも条件が真の式が含まれている可能性があるとしても、その他すべての式を無視します。WHEN 節で真の式が 1 つも見つからない場合は、OTHERWISE 節の後に続く命令を処理します。

IF...THEN...ELSE の場合と同様に、可能なパスに対して複数の命令がある場合は、それらの命令一式を DO で開始して END で終了します。ただし、OTHERWISE キーワードの後に複数の命令が続く場合、それらの命令に DO と END は必要ありません。

```

/***** REXX *****/
/* This program receives input with a person's age and sex. In */
/* reply, it produces a person's status as follows:          */
/* BABIES - under 5                                          */
/* GIRLS - female 5 to 12                                    */
/* BOYS - male 5 to 12                                       */
/* TEENAGERS - 13 through 19                                */
/* WOMEN - female 20 and up                                  */
/* MEN - male 20 and up                                      */
/*****/
PARSE ARG age sex .

SELECT
  WHEN age < 5 THEN /* person younger than 5 */
    status = 'BABY'
  WHEN age < 13 THEN /* person between 5 and 12 */
    DO
      IF sex = 'M' THEN /* boy between 5 and 12 */
        status = 'BOY'
      ELSE /* girl between 5 and 12 */
        status = 'GIRL'
    END
  WHEN age < 20 THEN /* person between 13 and 19 */
    status = 'TEENAGER'
  OTHERWISE
    IF sex = 'M' THEN /* man 20 or older */
      status = 'MAN'
    ELSE /* woman 20 or older */
      status = 'WOMAN'
END

SAY 'This person should be counted as a' status'.'

```

図 15. SELECT WHEN...OTHERWISE...END の使用例

各 SELECT は、END で終了しなければなりません。各 WHEN を字下げすると、プログラムが読みやすくなります。

演習: SELECT WHEN...OTHERWISE...END の使用

「9 月、4 月、6 月、および 11 月は 30 日、残りのすべての月は、2 月のみを除いて 31 日 ...」

月を表す 1 から 12 のいずれかの数値を入力として使用し、その月の日数を生成するプログラムを作成してください。プログラムを呼び出す際に、ユーザーが月の数値を引数として指定することを前提とします。(月の数値を変数 month に割り当てるために、プログラムに ARG 命令を組み込みます)。その後、プログラムに日数を生成させます。月の値が 2 の場合、これは 28 または 29 になります。

解答

```

/***** REXX *****/
/* This program uses the input of a whole number from 1 to 12 that */
/* represents a month. It produces the number of days in that      */
/* month.                                                           */
/*****

ARG month

SELECT
  WHEN month = 9 THEN
    days = 30
  WHEN month = 4 THEN
    days = 30
  WHEN month = 6 THEN
    days = 30
  WHEN month = 11 THEN
    days = 30
  WHEN month = 2 THEN
    days = '28 or 29'
  OTHERWISE
    days = 31
END

SAY 'There are' days 'days in Month' month'.'

```

図 16. 考えられる解決方法

ループ命令

ループ命令には、反復ループと条件付きループの2種類があります。どちらのタイプであるかを問わず、すべてのループは DO キーワードで開始し、END キーワードで終了します。

反復ループでは、命令を特定の回数繰り返すことができます。条件付きループでは、条件を使用して繰り返しを制御します。条件付きループには、DO WHILE と DO UNTIL の2つのタイプがあります。

反復ループ

最も単純な反復ループは、言語処理プログラムに対し、一連の命令を特定の回数繰り返すよう指示します。回数を指定するために、DO キーワードの後に定数を使用します。

以下の例を実行すると、Hello! が5行出力されます。

```

DO 5
SAY 'Hello!'
END

```

```

Hello!
Hello!
Hello!
Hello!
Hello!

```

以下の例に示すように、定数の代わりに変数を使用することもできます。この場合も結果は同じです。

```

number = 5
DO number
  SAY 'Hello!'
END

```

ループを繰り返す回数を制御する変数は、制御変数と呼ばれます。特に指定しない限り、制御変数の値はループの繰り返しごとに1ずつ増加します。

```

DO number = 1 TO 5
  SAY 'Loop' number
  SAY 'Hello!'
END
SAY 'Dropped out of the loop when number reached' number

```

この例では、ループの番号の後に Hello! の行が 5 回出力されます。番号はループの最後に増加し、ループの先頭でテストされます。

```
Loop 1
Hello!
Loop 2
Hello!
Loop 3
Hello!
Loop 4
Hello!
Loop 5
Hello!
Dropped out of the loop when number reached 6
```

制御変数の増分は、以下のように BY キーワードを使用して変更できます。

```
DO number = 1 TO 10 BY 2
  SAY 'Loop' number
  SAY 'Hello!'
END
  SAY 'Dropped out of the loop when number reached' number
```

以下の例は、前の例と同じような結果になりますが、ループの番号の増分は 2 になっています。

```
Loop 1
Hello!
Loop 3
Hello!
Loop 5
Hello!
Loop 7
Hello!
Loop 9
Hello!
Dropped out of the loop when number reached 11
```

無限ループ

ループの制御変数が最後の番号を取得できない場合はどうなるでしょうか。例えば、以下のプログラム・セグメントでは、count の値が 1 より大きな値に増えません。

```
DO count = 1 to 10
  SAY 'Number' count
  count = count - 1
END
```

count の値は 1 と 0 が交互に入れ替わり、Number 1 という行が無限に出力されることから、無限ループと呼ばれる結果になります。

プログラムが無限ループに入った場合は、オペレーターに連絡してループをキャンセルしてもらう必要があります。許可ユーザーは、**CEMT SET TASK PURGE** コマンドを発行して、EXEC を停止できます。

DO FOREVER ループ

意図的に無限ループを作成したい場合もあります。ファイルの終わりに達するまでファイルからレコードを読み取るプログラムは、その一例です。以下の例のように EXIT 命令を使用することで、条件が満たされた時点で無限ループを終了できます。EXIT 命令について詳しくは、[50 ページの『EXIT 命令』](#)を参照してください。


```

/***** REXX *****/
/* This program processes strings until the value of a string is */
/* a null string. */
/*****
DO FOREVER
  PULL string          /* Gets string from input stream */
  IF string = '' THEN
    PULL file_name
    IF file_name = '' THEN
      EXIT
    ELSE
      DO
        result = process(string) /* Calls a user-written function */
                                /* to do processing on string. */
        IF result = 0 THEN SAY "Processing complete for string:" string
        ELSE SAY "Processing failed for string:" string
      END
    END
  END
END

```

図 17. DO FOREVER ループの使用例

この例は、ユーザー作成の関数にストリングを送信して処理した後、処理が正常に完了したこと、または失敗したことを通知するメッセージを発行します。入力ストリングがブランクの場合、ループは終了し、プログラムも終了します。LEAVE 命令を使用することで、プログラムを終了せずにループを終了することもできます。

LEAVE 命令

LEAVE 命令は、反復ループを即時に終了させます。制御が移る先は、ループの END キーワードの後に続く命令です。以下に、LEAVE 命令の使用例を示します。

```

/***** REXX *****/
/* This program uses the LEAVE instruction to exit from a DO */
/* FOREVER loop. */
/*****
DO FOREVER
  PULL string          /* Gets string from input stream */
  IF string = 'QUIT' then
    LEAVE
  ELSE
    DO
      result = process(string) /* Calls a user-written function */
                              /* to do processing on string. */
      IF result = 0 THEN SAY "Processing complete for string:" string
      ELSE SAY "Processing failed for string:" string
    END
  END
END
SAY 'Program run complete.'

```

図 18. LEAVE 命令の使用例

ITERATE 命令

ITERATE 命令は、ループ内から実行を停止し、ループの先頭にある DO 命令に制御を渡します。DO 命令のタイプに応じて、言語処理プログラムは制御変数を増分してテストするか、条件をテストしてループを繰り返すかどうかを決定します。LEAVE と同様に、ITERATE はループ内で使用します。

```

DO count = 1 TO 10
  IF count = 8
    THEN
      ITERATE
  ELSE
    SAY 'Number' count
END

```

この例の結果は、数字 8 を除く、1 から 10 までの数字のリストになります。

```

Number 1
Number 2
Number 3
Number 4

```

```
Number 5
Number 6
Number 7
Number 9
Number 10
```

演習: ループの使用

以下のループは、どのような結果になりますか。

```
DO digit = 1 TO 3
  SAY digit
END
SAY 'Digit is now' digit
```

```
DO count = 10 BY -2 TO 6
  SAY count
END
SAY 'Count is now' count
```

```
DO index = 10 TO 8
  SAY 'Hup! Hup! Hup!'
END
SAY 'Index is now' index
```

ループを終了する入力が予期した入力と一致しない場合に、無限ループが発生する可能性があります。例えば、上記の LEAVE 命令の使用例で、Quit が入力されたときに、PARSE PULL 命令が PULL 命令に置き換わっていたらどうなるでしょうか。

```
PARSE PULL file_name
```

応答

1. 反復ループの結果は、次のようになります。

a. 1
2
3
Digit is now 4

b. 10
8
6
Count is now 4

c. Index is now 10

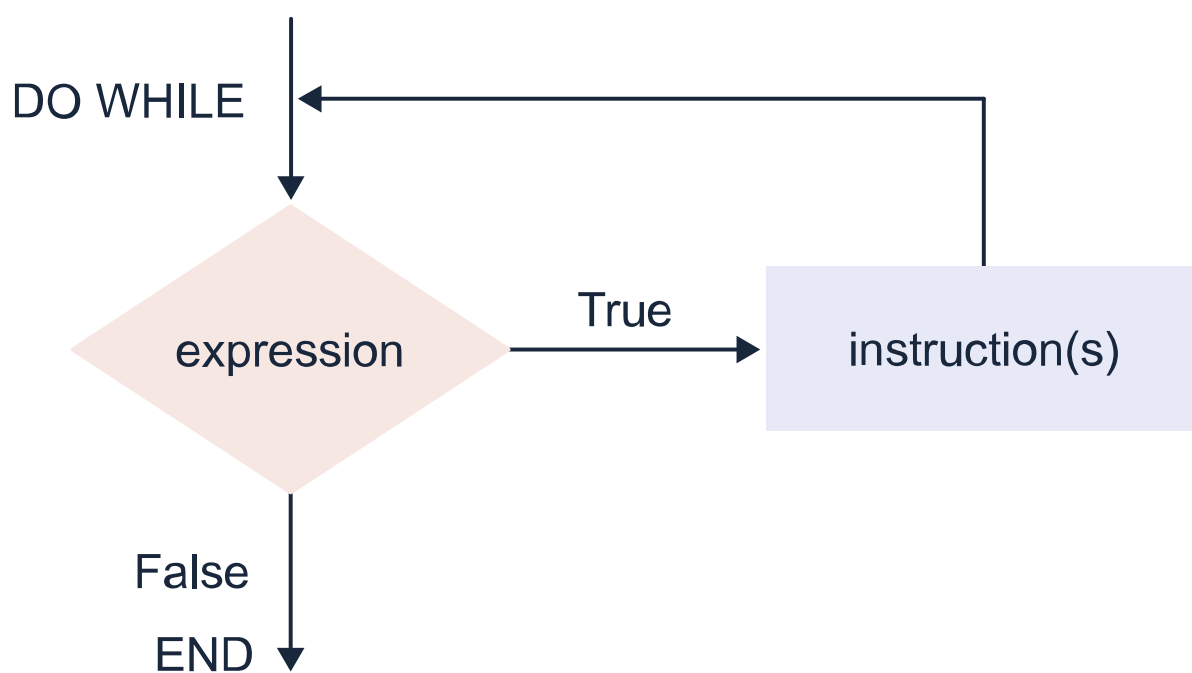
2. Quit は QUIT と等しくないため、プログラムがループを抜けられなくなります。この場合、PARSE キーワードを省略することが推奨されます。このキーワードを省略すれば、入力が quit、QUIT、または Quit のいずれであっても、言語処理プログラムは入力を大文字に変換してから QUIT と比較するためです。

条件付きループ

条件付きループには、DO WHILE と DO UNTIL の 2 つのタイプがあります。どちらのタイプのループも、1 つ以上の式で制御されます。ただし、DO WHILE ループの場合、ループの初回の実行前に式がテストされ、式の結果が真である場合にのみ繰り返されます。DO UNTIL ループの場合は、ループの実行後に少なくとも 1 回式がテストされ、式の結果が偽である場合にのみ繰り返されます。

DO WHILE ループ

DO WHILE ループは、フローチャートでは以下のように表されます。



REXX 命令では、上記のフローチャートの例は以下ようになります。

```
DO WHILE expression /* expression must be true */
  instruction(s)
END
```

条件が真である限りループを実行する必要がある場合は、DO WHILE ループを使用します。DO WHILE はループの先頭で条件をテストします。条件が最初から偽である場合、言語処理プログラムがそのループを実行することはありません。

41 ページの『反復ループ』に記載されている LEAVE の使用例では、DO FOREVER ループの代わりに DO WHILE ループを使用することもできます。ただし、最初のケースでループに入る前に条件をテストできるよう、ループを初期化する必要があります。以下の例では、最初のケースでの初期化が最初の PULL で行われていることに注意してください。

```
/****** REXX ******/
/* This progra uses a DO WHILE loop to send a string to a */
/* user-written function for processing. */
/*******/
PULL string /* Gets string from input stream */
DO WHILE string &= 'QUIT'
  result = process(string) /* Calls a user-written function */
                          /* to do processing on string. */
  IF result = 0 THEN SAY "Processing complete for string:" string
  ELSE SAY "Processing failed for string:" string
  PULL string
END
SAY 'Program run complete.'
```

図 19. DO WHILE の使用例

演習: DO WHILE ループの使用

コンピューター航空便で窓側の席を希望するかどうかについて、乗客の回答を記録したリストがあります。このリストを入力として取る DO WHILE ループを使用したプログラムを作成します。航空便の乗客は 8 人で、窓側の席は 4 つあります。窓側の席がすべて埋まった時点でループを終了します。ループが終了した後、埋まっている窓側の席の数と、処理した回答数を出力してください。

ANSWER

```
/****** REXX ******/
/* This program uses a DO WHILE loop to keep track of window seats */
/* in an 8-seat commuter airline. */
/*******/

window_seats = 0 /* Initialize window seats to 0 */
passenger = 0 /* Initialize passengers to 0 */

DO WHILE (passenger < 8) & (window_seats &= 4)

/*******/
/* Continue while the program has not yet read the responses of */
/* all 8 passengers and while all the window seats are not taken. */
/*******/

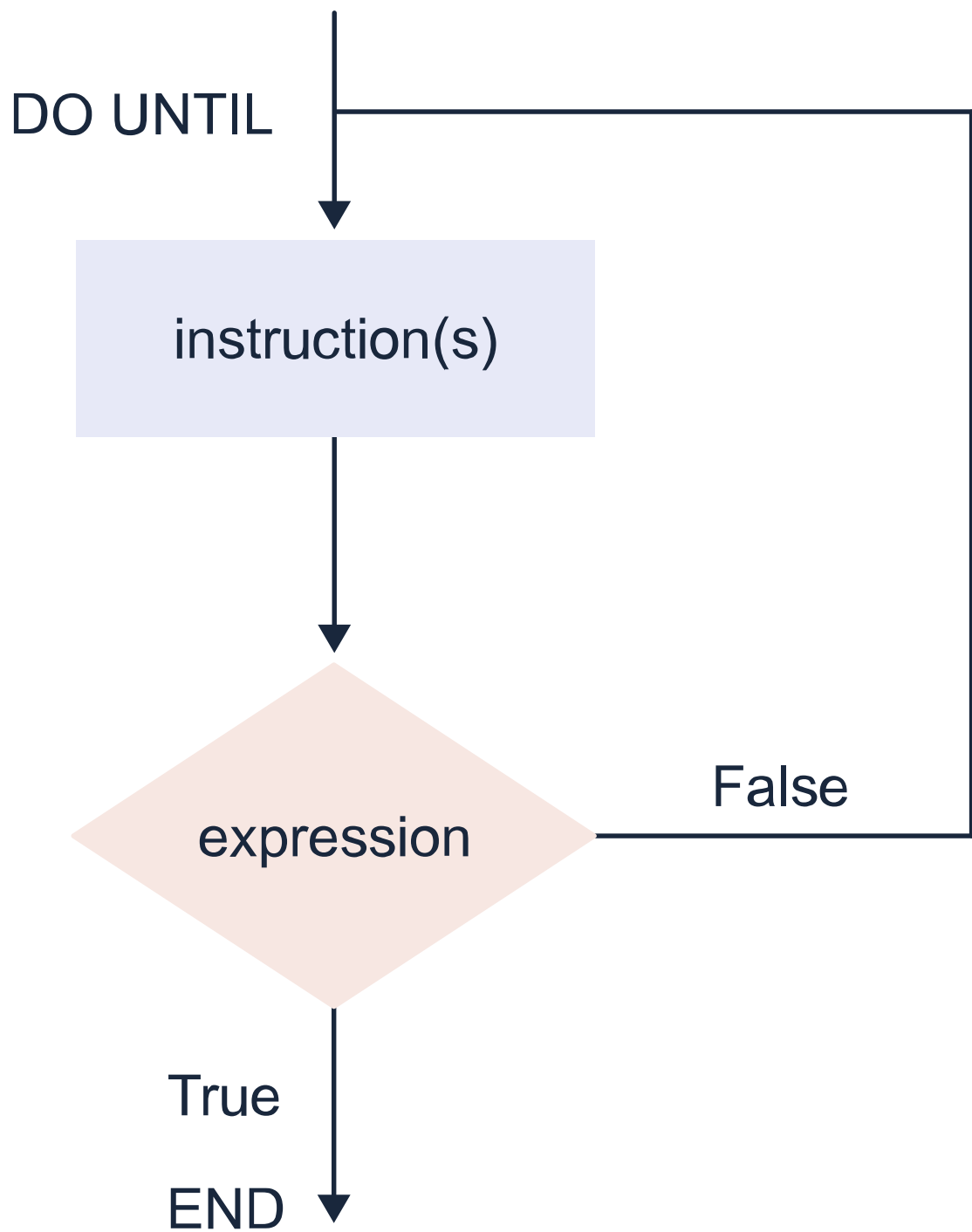
PULL window /* Gets "Y" or "N" from input stream */
passenger = passenger + 1 /* Increase number of passengers by 1 */
IF window = 'Y' THEN
  window_seats = window_seats + 1 /* Increase window seats by 1 */
ELSE NOP
END

SAY window_seats 'window seats were assigned.'
SAY passenger 'passengers were questioned.'
```

図 20. 考えられる解決方法

DO UNTIL ループ

DO UNTIL ループは、フローチャートでは以下のように表されます。



REXX 命令では、上記のフローチャートの例は以下ようになります。

```
DO UNTIL expression /* expression must be false */
  instruction(s)
END
```

条件が真でない場合にループを実行し、条件が真になるまでループを続行するには、DO UNTIL ループを使用します。DO UNTIL ループは、ループの最後で条件をテストし、条件が偽である場合にのみ繰り返します。そうでない場合は、ループは 1 回実行されて終了します。以下に例を示します。

```
/****** REXX *****/
/* This program uses a DO UNTIL loop to ask for a password. If the */
/* password is incorrect three times, the loop ends. */
/****** REXX *****/
password = 'abracadabra'
time = 0
DO UNTIL (answer = password) | (time = 3)
  PULL answer /* Gets ANSWER from input stream */
  time = time + 1
END
```

図 21. DO UNTIL の使用例

演習: DO UNTIL ループの使用

前の演習のプログラムを DO WHILE から DO の UNTIL ループに変更し、同じ結果を得るようにします。DO WHILE ループは式の結果が真であることを検査し、DO UNTIL ループは式の結果が偽であるかどうかを検査するため、通常は、論理演算子が逆になるという点に注意してください。

解答

```
/****** REXX *****/
/* This program uses a DO UNTIL loop to keep track of window seats */
/* in an 8-seat commuter airline. */
/****** REXX *****/

window_seats = 0 /* Initialize window seats to 0 */
passenger = 0 /* Initialize passengers to 0 */

DO UNTIL (passenger >= 8) | (window_seats = 4)

  /****** REXX *****/
  /* Continue while the program has not yet read the responses of */
  /* all 8 passengers and while all the window seats are not taken. */
  /****** REXX *****/

  PULL window /* Gets "Y" or "N" from input stream */
  passenger = passenger + 1 /* Increase number of passengers by 1 */
  IF window = 'Y' THEN
    window_seats = window_seats + 1 /* Increase window seats by 1 */
  ELSE NOP
END
SAY window_seats 'window seats were assigned.'
SAY passenger 'passengers were questioned.'
```

図 22. 考えられる解決方法

複合ループ

反復ループと条件付きループを結合して、複合ループを作成できます。

以下のループは、数量が 50 より少ない間に 10 回反復し、数量が 50 になった時点で停止するように設定されています。

```
quantity = 20
DO number = 1 TO 10 WHILE quantity < 50
  quantity = quantity + number
  SAY 'Quantity = 'quantity ' (Loop 'number')'
END
```

この例の結果は、以下のとおりです。

```
Quantity = 21 (Loop 1)
Quantity = 23 (Loop 2)
```

```

Quantity = 26 (Loop 3)
Quantity = 30 (Loop 4)
Quantity = 35 (Loop 5)
Quantity = 41 (Loop 6)
Quantity = 48 (Loop 7)
Quantity = 56 (Loop 8)

```

DO UNTIL ループで置き換えて、比較演算子を < から > に変更しても、同じ結果を得ることができます。

```

quantity = 20
DO number = 1 TO 10 UNTIL quantity > 50
    quantity = quantity + number
    SAY 'Quantity = 'quantity ' (Loop 'number')'
END

```

ネストされた DO ループ

ネストされた IF...THEN...ELSE 命令と同様に、DO ループ内に他の DO ループを含めることができます。

以下に、単純な例を示します。

```

DO outer = 1 TO 2
    DO inner = 1 TO 2
        SAY 'HIP'
    END
    SAY 'HURRAH'
END

```

この例の出力は次のとおりです。

```

HIP
HIP
HURRAH
HIP
HIP
HURRAH

```

特定の条件が発生した時点でループから抜ける必要がある場合は、LEAVE 命令を使用し、この命令の後にループの制御変数の名前を続けます。LEAVE 命令の対象が内部ループである場合、処理は内部ループを抜けて外部ループに移ります。LEAVE 命令の対象が外部ループである場合、処理は内部ループと外部ループの両方から抜けます。

前の例で内部ループから抜けるには、IF...THEN...ELSE 命令を追加し、IF 命令の後に LEAVE 命令を含めます。

```

DO outer = 1 TO 2
    DO inner = 1 TO 2
        IF inner > 1 THEN
            LEAVE inner
        ELSE
            SAY 'HIP'
        END
    END
    SAY 'HURRAH'
END

```

結果は、次のようになります。

```

HIP
HURRAH
HIP
HURRAH

```

演習: ループの結合

1. 以下のプログラムを実行すると、どのような結果になるでしょうか。

```

DO outer = 1 TO 3
    SAY          /* Produces a blank line */
    DO inner = 1 TO 3
        SAY 'Outer' outer 'Inner' inner
    END
END

```

```
END
END
```

2. LEAVE 命令を追加すると、どうなるでしょうか。

```
DO outer = 1 TO 3
  SAY /* Produces a blank line */
  DO inner = 1 TO 3
    IF inner = 2 THEN
      LEAVE inner
    ELSE
      SAY 'Outer' outer 'Inner' inner
    END
  END
END
```

応答

1. この例を実行すると、以下の出力が生成されます。

```
Outer 1 Inner 1
Outer 1 Inner 2
Outer 1 Inner 3

Outer 2 Inner 1
Outer 2 Inner 2
Outer 2 Inner 3

Outer 3 Inner 1
Outer 3 Inner 2
Outer 3 Inner 3
```

2. 内部ループが実行される度に、1 行の出力が生成されます。

```
Outer 1 Inner 1

Outer 2 Inner 1

Outer 3 Inner 1
```

割り込み命令

割り込み命令には、EXIT、SIGNAL、CALL、および RETURN があります。

プログラムのフローを中断する命令により、プログラムを以下のように動作させることができます。

- 終了する (EXIT)
- ラベルでマークされた、プログラムの別の部分にスキップする (SIGNAL)
- 一時的にプログラム内部またはプログラム外部のサブルーチンに進む (CALL または RETURN)

EXIT 命令

EXIT 命令によって、REXX プログラムは無条件で終了し、プログラムが呼び出された箇所に戻ります。別のプログラムが REXX プログラムを呼び出した場合は、EXIT により、その呼び出し側プログラムに戻ります。

EXIT はプログラムを終了させるだけでなく、呼び出し側プログラムに値を返すこともできます。プログラムがサブルーチンとして別の REXX プログラムから呼び出された場合、REXX の特殊変数 RESULT で値を受け取ります。プログラムが関数として呼び出された場合、関数が呼び出された箇所にある、元の式で値を受け取ります。それ以外の場合は、REXX の特殊変数 RC で値を受け取ります。値は、戻りコードを表すことができ、定数の形式または計算される式の形式にできます。


```

/***** REXX *****/
/* This program uses the EXIT instruction to end the program and */
/* return a value indicating whether a job applicant gets the   */
/* job. A value of 0 means the applicant does not qualify for    */
/* the job, but a value of 1 means the applicant gets the job.   */
/* The value is placed in the REXX special variable RESULT.     */
/***** */
PULL months_experience      /* Gets number from input stream */
PULL references             /* Gets "Y" or "N" from input stream */
PULL start_tomorrow        /* Gets "Y" or "N" from input stream */

IF (months_experience > 24) & (references = 'Y') & (start_tomorrow = 'Y')
THEN job = 1                /* person gets the job */
ELSE job = 0                /* person does not get the job */

EXIT job

```

図 23. EXIT 命令の使用例

外部ルーチンの呼び出しについて詳しくは、62 ページの『サブルーチンと関数』を参照してください。
EXIT 命令について詳しくは、[EXIT](#) を参照してください。

CALL 命令と RETURN 命令

CALL 命令は、制御を内部または外部サブルーチンに渡して、プログラムのフローを中断します。内部サブルーチンは、呼び出し側プログラムの一部です。外部サブルーチンは、別のプログラムです。RETURN 命令は、サブルーチンから呼び出し側プログラムに制御を戻し、オプションで値を返します。

CALL が内部サブルーチンを呼び出すと、CALL キーワードの後に指定されたラベルに制御が渡されます。サブルーチンが RETURN 命令で終了すると、CALL 命令の後に続く命令が処理されます。

instruction(s)

CALL sub1

instruction(s)

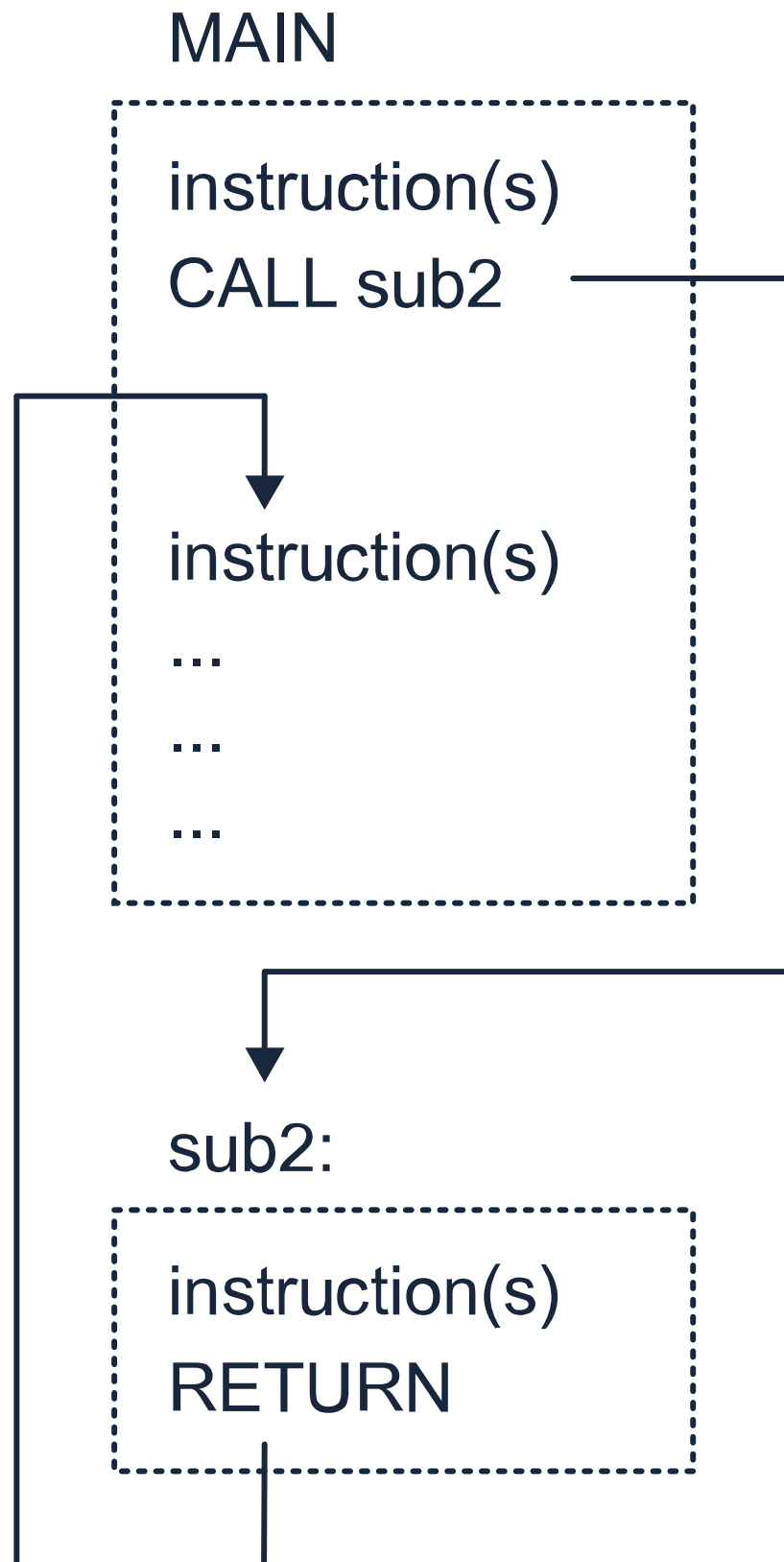
EXIT

sub1:

instruction(s)

RETURN

CALL が外部サブルーチンを呼び出すと、CALL キーワードの後に名前が指定されたプログラムに制御が渡されます。外部サブルーチンが完了したら、RETURN 命令を使用して、呼び出し側プログラムの中斷された場所に戻ることができます。



サブルーチンの呼び出しについて詳しくは、62 ページの『サブルーチンと関数』を参照してください。

ON パラメーターを指定した CALL 命令を使用することで、例外条件 (例えば、エラー状態や障害状態) を REXX が検出すると制御を受け取るルーチンを実装できます。 [条件および条件トラップ](#)を参照してください。

CALL 命令と RETURN 命令について詳しくは、[CALL](#) および [RETURN](#) を参照してください。

SIGNAL 命令

SIGNAL 命令は CALL 命令と同様に、プログラムの通常のフローを中断し、指定されたラベルに制御が渡されるようにします。制御を渡す先のラベルは、SIGNAL 命令の前または後ろに指定できます。CALL とは異なり、SIGNAL は特定の命令に戻って実行を再開することはありません。

SIGNAL をループ内で使用する場合、ループは自動的に終了します。内部ルーチンから SIGNAL を使用する場合、その内部ルーチンは呼び出し側に戻りません。

SIGNAL は、プログラムをテストする場合や、緊急時の措置を取る場合に役立ちます。プログラム内である場所から別の場所に移るための便利な方法として使用すべきではありません。SIGNAL は、[CALL](#) 命令とは異なり、戻るための方法がありません。

ON パラメーターを指定した SIGNAL 命令を使用することで、例外条件 (例えば、エラー状態や障害状態) を REXX が検出すると制御を受け取るルーチンを実装できます。 [条件および条件トラップ](#)を参照してください。

SIGNAL 命令について詳しくは、97 ページの『REXX の特殊変数 RC および SIGL の使用』および [SIGNAL](#) を参照してください。

例

以下の例では、式が true に評価されると、言語処理プログラムがラベル Emergency: に進み、その間にあるすべての命令をスキップします。

IF expression THEN
 SIGNAL Emergency
ELSE
 instruction(s)



Emergency:
instruction(s)

第 5 章 関数

関数とは、データを受信し、それを処理し、値を返すことができる命令のシーケンスです。REXX には、いくつかの種類の関数があります。

- 組み込み関数は、言語処理プログラムに組み込まれています。詳しくは、[58 ページの『組み込み関数』](#)を参照してください。
- ユーザー作成関数は、個々のユーザーが作成する関数、またはインストール済み環境が提供する関数です。これらの関数には、内部関数と外部関数があります。内部関数は、ラベルで開始される現行プログラムの一部です。外部関数は、呼び出し側プログラムの外部にある自己完結型のプログラムです。ユーザー作成関数については、[64 ページの『サブルーチンおよび関数の作成』](#)を参照してください。

関数の種類に関係なく、すべての関数は、関数呼び出しを発行したプログラムに値を返します。関数を呼び出すには、関数名を入力し、その直後に関数に渡す引数を括弧で囲んで続けます (ある場合)。**関数名と左括弧の間にスペースがあってはなりません。**

```
function(arguments)
```

関数呼び出しには、コンマで区切って、最大 20 個までの引数を指定できます。引数は以下のいずれかにすることができます。

- 定数

```
function(55)
```

- 記号

```
function(symbol_name)
```

- 関数が認識するオプション

```
function(option)
```

- リテラル・ストリング

```
function('With a literal string')
```

- 未指定または省略

```
function()
```

- 別の関数

```
function(function(arguments))
```

- 引数タイプの組み合わせ

```
function('With literal string', 55, option)
function('With literal string',, option) /* Second argument
omitted */
```

すべての関数は値を返す必要があります。関数が値を返すと、その値で関数呼び出しが置き換えられます。以下の例では、言語処理プログラムは関数から返された値を 7 に加算して、その合計を算出します。

```
SAY 7 + function(arguments)
```

通常、関数呼び出しは式の中で使用されます。したがって、式と同じように、通常は関数呼び出し自体が命令の中に現れることはありません。

関数の例

関数が表す計算には、多くの命令が必要になることがよくあります。例えば、3つの数値の中から最大数を見つけるための単純な計算は、次のように作成できます。

```
/****** REXX ******/
/* This program receives three numbers as arguments and analyzes */
/* which number is the greatest. */
/*******/

PARSE ARG number1, number2, number3 .

IF number1 > number2 THEN
  IF number1 > number3 THEN
    greatest = number1
  ELSE
    greatest = number3
ELSE
  IF number2 > number3 THEN
    greatest = number2
  ELSE
    greatest = number3

RETURN greatest
```

図 24. 最大数の検索

3つの数値の中から最大数を検索したい場合に、毎回複数の命令を作成するのではなく、自動的に計算して最大数を返す組み込み関数を使用できます。それは、MAXという名前の関数です。MAXは、以下のように使用できます。

```
MAX(number1,number2,number3,...)
```

45、-2、*number*、199の中から最大数を検索し、その最大数を記号 **biggest** に取り込むには、次の命令を作成します。

```
biggest = MAX(45,-2,number,199)
```

組み込み関数

言語処理プログラムには、50個を超える関数が組み込まれています。

これらの組み込み関数は、以下のカテゴリーに分類されます。

算術関数

引数から数値を評価し、特定の値を戻します。

比較関数

数値またはストリング、あるいはその両方を比較して、値を戻します。

変換関数

データ表記のタイプを別のデータ表記のタイプに変換します。

フォーマット設定関数

引数で渡されたストリングに含まれる文字とスペーシングを操作します。

ストリング操作関数

引数で渡されたストリング (またはストリングを表す変数) を分析し、特定の値を戻します。

その他の関数

その他のカテゴリーのいずれにも分類されない関数です。

以下の表で、各カテゴリーの関数について簡単に説明します。詳細な説明については、[関数](#)を参照してください。

算術関数

機能	説明
ABS	入力された数値の絶対値を返します。
DIGITS	NUMERIC DIGITS の現行設定値を返します。
FORM	NUMERIC FORM の現行設定値を返します。
FUZZ	NUMERIC FUZZ の現行設定値を返します。
MAX	指定されたリストのうち最大の数値を、現行の NUMERIC 設定値に従って形式設定して返します。
MIN	指定されたリストのうち最小の数値を、現行の NUMERIC 設定値に従って形式設定して返します。
RANDOM	指定された範囲から、準乱数の負ではない整数を返します。
SIGN	入力値の符号を表す数値を返します。
TRUNC	入力値の整数部分と、オプションで指定された小数点以下の桁数を返します。

比較関数

機能	説明
COMPARE	2 つの入力ストリングが同一の場合は、0 を返します。異なる場合には、一致しない最初の文字の位置を返します。
DATATYPE	入力ストリングが特定のデータ型 (数値、文字など) であることを示すストリングを返します。
SYMBOL	記号の状態 (変数、リテラル、または不正) をそれぞれ示す VAR、LIT、BAD を返します。

変換関数

機能	説明
B2X	入力されたバイナリー・ストリングの 16 進表記を返します。(2 進数から 16 進数への変換)。
C2D	入力された文字ストリングの 10 進表記を返します。(文字から 10 進数への変換)。
C2X	入力された文字ストリングの 16 進表記を返します。(文字から 16 進数への変換)。
D2C	入力された 10 進数ストリングの文字表記を返します。(10 進数から文字への変換)。
D2X	入力された 10 進数ストリングの 16 進表記を返します。(10 進数から 16 進数への変換)。
X2B	入力された 16 進数ストリングの 2 進表記を返します。(16 進数から 2 進数への変換)。
X2C	入力された 16 進数ストリングの文字表記を返します。(16 進数から文字への変換)。
X2D	入力された 16 進数ストリングの 10 進表記を返します。(16 進数から 10 進数への変換)。

フォーマット設定関数

機能	説明
CENTER または CENTRE	入力ストリングを中央揃えにして、指定の長さのストリングを戻します。指定の長さにするために、必要に応じて埋め込み文字を追加します。
COPIES	入力ストリングのコピーを指定の数だけ連結したストリングを戻します。
FORMAT	入力値を丸めてフォーマット設定した上で戻します。
JUSTIFY ¹	ストリングのワードとワードの間に埋め込み文字を追加して、双方のマージンに位置調整されるようフォーマット設定した上で、指定されたストリングを戻します。
LEFT	ストリングの右側を切り捨てるか、埋め込み文字を追加して指定の長さにした上で、ストリングを戻します。
RIGHT	ストリングの左側を切り捨てるか、埋め込み文字を追加して指定の長さにした上で、ストリングを戻します。
SPACE	入力ストリングに含まれるワードとワードの間に指定の数の埋め込み文字を追加した上で、ストリングを戻します。

1. REXX/CICS が提供する非 SAA 組み込み関数です。

ストリング操作関数

機能	説明
ABBREV	所定のストリングが別のストリングに含まれる指定の先行文字数のストリングと等しいかどうかを示すストリングを戻します。
DELSTR	入力ストリングの指定の開始点から指定の文字数を削除した上で、ストリングを戻します。
DELWORD	入力ストリングから、指定のワードを起点に指定のワード数を削除した上で、ストリングを戻します。
FIND ¹	入力ストリング内で検出された指定の句の最初のワードのワード番号を戻します。
INDEX ¹	入力ストリング内で検出された指定のストリングの先頭文字の文字位置を戻します。
INSERT	指定された文字位置に別の入力ストリングを挿入した上で、文字ストリングを戻します。
LASTPOS	所定のストリングが別のストリング内で最後に出現する場所の開始文字位置を戻します。
LENGTH	入力ストリングの長さを戻します。
OVERLAY	オーバーレイのターゲット・ストリングを戻します。このストリングが、2 番目の入力ストリングでオーバーレイされます。
POS	所定のストリングの別のストリング内での文字位置を戻します。
REVERSE	元の文字ストリングを反転させた文字ストリングを戻します。
STRIP	入力ストリングから先頭または末尾の文字、またはその両方を削除した上で、文字ストリングを戻します。
SUBSTR	入力ストリングの指定の文字位置からの部分を戻します。
SUBWORD	入力ストリングの指定のワード数からの部分を戻します。
TRANSLATE	入力ストリングのそれぞれの文字を別の文字に変換するか、または未変更のままにして、文字ストリングを戻します。

機能	説明
VERIFY	入力ストリングが別の入力ストリングの文字のみで構成されているかどうかを示す数値を戻すか、別の入力ストリングの文字と一致していない最初の文字の文字位置を戻します。
WORD	入力ストリングから、番号で指定されたワードを戻します。
WORDINDEX	入力ストリングに含まれる、指定されたワードの最初の文字の文字位置を戻します。
WORDLENGTH	入力ストリングに含まれる、指定されたワードの長さを戻します。
WORDPOS	入力ストリングに含まれる、指定の句の最初のワードのワード番号を戻します。
WORDS	入力ストリングのワード数を戻します。

1. REXX/CICS が提供する非 SAA 組み込み関数です。

その他の関数

機能	説明
ADDRESS	現在、コマンドの送信先となっている環境の名前を戻します。
ARG	プログラムまたは内部ルーチンに対する引数ストリングに関する情報、あるいは引数ストリングを戻します。
BITAND	2つの入力ストリングを、ビットごとに論理的に AND 演算して得られるストリングを戻します。
BITOR	2つの入力ストリングをビットごとに論理的に OR 演算して得られるストリングを戻します。
BITXOR	2つの入力ストリングをビットごとに排他的に OR 演算して得られるストリングを戻します。
CONDITION	現在トラップされている条件に関連した条件情報(名前、状況など)を戻します。
DATE	デフォルト形式(dd mon yyyy)または各種のオプション形式のいずれかで日付を戻します。
ERRORTXT	指定されたエラー番号に関連付けられているエラー・メッセージを戻します。
EXTERNALS ¹	この関数は常に 0 を戻します。
LINESIZE ¹	現在の出力装置の幅を戻します。
QUEUED	この関数が呼び出された時点で外部データ・キューに残っている行数を戻します。
SOURCELINE	ソース・ファイルの最終行の行番号、または番号で指定されたソース行のいずれかを戻します。
TIME	デフォルトの 24 時間クロック形式(hh:mm:ss)または各種のオプション形式のいずれかで現地時間を戻します。
TRACE	現在有効になっているトレース・アクションを戻します。
USERID ¹	現行ユーザー ID を戻します。これは、SETUID コマンドで最後に指定されたユーザー ID、呼び出し側 REXX プログラムのユーザー ID (プログラムが別のプログラムを呼び出している場合)、ジョブを実行中のユーザー ID、またはジョブ名です。
VALUE	指定された記号の値を戻し、オプションで新しい値を割り当てます。
XRANGE	指定された範囲(開始値と終了値を含む)のすべての 1 バイトのコードが昇順で含まれるストリングを戻します。

1. REXX/CICS が提供する非 SAA 組み込み関数です。

組み込み関数を使用した入力のテスト

組み込み関数の中には、入力をテストする便利な方法が用意されているものもあります。プログラムが入力を使用する場合、ユーザーによって無効な入力提供される可能性があります。例えば、[24 ページの『比較演算子』](#)の比較式の使用例では、プログラムが以下の命令で金額 (ドル) を使用しています。

```
PARSE PULL yesterday /* Gets yesterday's price from input stream */
```

プログラムが数値だけをプルするのであれば、プログラムはその情報を正しく処理します。ただし、プログラムが先頭にドル記号が付いた数値または nothing などのワードをプルすると、プログラムがエラーを戻します。エラーが発生しないようにするには、以下のように、DATATYPE 関数を使用して入力を確認できます。

```
IF DATATYPE(yesterday) ≠ 'NUM'
THEN DO
  SAY 'The input amount was in the wrong format.'
  EXIT
END
```

入力のテストに役立つ組み込み関数には、他にも WORDS、VERIFY、LENGTH、SIGN があります。

演習: 組み込み関数を使用したプログラムの作成

ファイル名の長さが 8 文字であるかどうかを確認するプログラムを作成します。名前が 8 文字を超える場合、プログラムは名前を 8 文字の長さにまで切り捨て、短縮名を示すメッセージを送信します。組み込み関数の [LENGTH 関数](#) および [SUBSTR \(サブストリング\) 関数](#) を使用します。

ANSWER

```
/****** REXX *****/
/* This program tests the length of a file name. */
/* If the name is longer than 8 characters, the program truncates */
/* extra characters and sends a message indicating the shortened */
/* name. */
/*******/
PULL name /* Gets name from input stream */

IF LENGTH(name) > 8 THEN /* Name is longer than 8 characters */
DO
  name = SUBSTR(name,1,8) /* Shorten name to first 8 characters */
  SAY 'The name you specified was too long.'
  SAY name 'will be used.'
END
ELSE NOP
```

図 25. 考えられる解決方法

サブルーチンと関数

サブルーチンおよび関数は、データを受け取り、処理し、値を返すことができる命令のシーケンスです。

ルーチンには、以下のタイプがあります。

内部

内部ルーチンは現行プログラム内にあり、ラベルでマークされます。また、そのルーチンを含むプログラムでのみ使用されます。

外部

別個のファイルとして存在する REXX サブルーチンです。

多くの点で、サブルーチンと関数は同じです。ただし、呼び出す方法および値を返す手段という点では、この 2 つには大きな違いがあります。

- サブルーチンの呼び出し

サブルーチン呼び出しには、CALL 命令を使用し、この命令の後に対象のサブルーチン名 (ラベルまたはプログラム・メンバー名) を続けます。オプションで、サブルーチン名の後に最大 20 個の引数をコマンドで区切って続けることができます。このサブルーチン呼び出し全体が命令になります。

```
CALL subroutine_name argument1, argument2,...
```

- 関数の呼び出し

関数を呼び出すには、関数名 (ラベルまたはプログラム・メンバー名) の直後に括弧を続けて使用します。括弧内には、引数を含めることができます。関数名と左括弧の間にスペースがあってはなりません。関数呼び出しは、例えば代入命令などの命令の一部になります。

```
z = function(argument1, argument2,...)
```

- サブルーチンからの戻り値

サブルーチンが値を返すことは必須ではありません。値を返す場合は、RETURN 命令を使用して値を返します。

```
RETURN value
```

呼び出し側プログラムは、RESULT という名前の REXX 特殊変数で値を受け取ります。

```
SAY 'The answer is' RESULT
```

- 関数からの戻り値

関数は値を返すことが**必須**です。関数が REXX プログラムである場合、RETURN 命令または EXIT 命令のいずれかを使用して値を返します。

```
RETURN value
```

呼び出し側プログラムは、関数呼び出しで値を受け取ります。値で関数呼び出しが置き換えられるため、以下の例では z が値となります。

```
z = function(argument1, argument2,...)
```

関数ではなくサブルーチンを作成する場合

サブルーチンと関数のいずれを使用しても、構成される実命令には変わりがないはずです。その命令をプログラム内で使用する方法によって、サブルーチンになるか、関数になるかが決まります。例えば、組み込み関数 SUBSTR は、関数として呼び出すことも、サブルーチンとして呼び出すこともできます。ワードを最初の 8 文字に短縮する関数として SUBSTR を呼び出すには、以下のようになります。

```
a = SUBSTR('verylongword',1,8) /* a is set to 'verylong' */
```

サブルーチンとして SUBSTR を呼び出したとしても、同じ結果になります。

```
CALL SUBSTR 'verylongword', 1, 8  
a = RESULT /* a is set to 'verylong' */
```

サブルーチンまたは関数のどちらを作成するかを決める際は、以下の点について自問自答してください。

- 値を返すかどうかはオプションであるか。オプションであれば、サブルーチンを作成します。
- 値を命令内の式として返す必要があるかどうか。その必要があれば、関数を作成します。

サブルーチンと関数: 類似点と相違点

サブルーチンと関数には、以下のような類似点があります。

- 内部または外部にすることができます。
 - 内部
 - 共通の変数を使用して情報を渡すことができます。

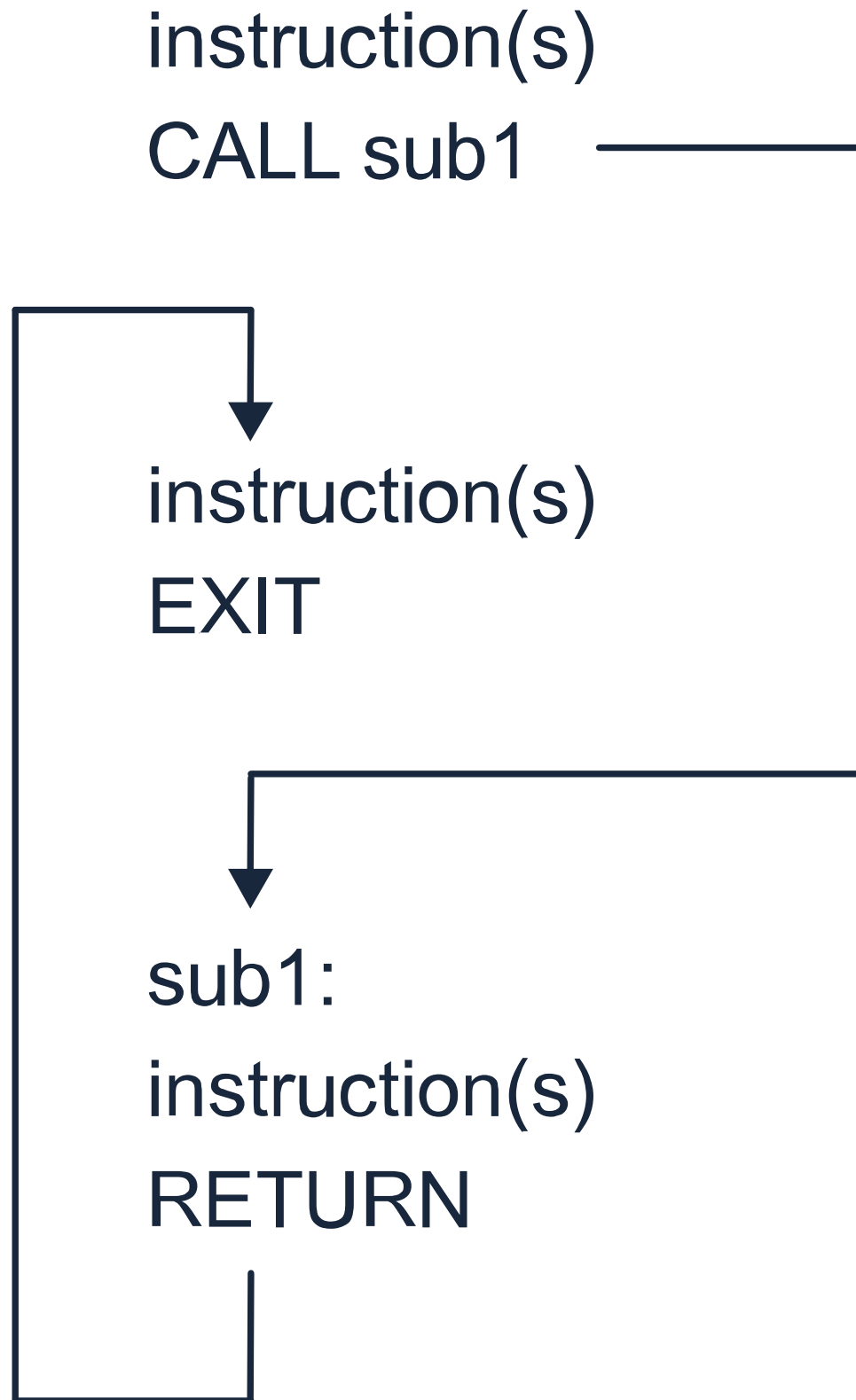
- PROCEDURE 命令を使用して変数を保護できます。
- 引数を使用して情報を渡すことができます。
- 外部
 - 情報を渡すには、引数を使用する必要があります。
 - ARG 命令または ARG 組み込み関数を使用して引数を受け取ることができます。
- RETURN 命令を使用して呼び出し側に戻ります。

表 3. サブルーチンと関数の相違点		
	サブルーチン	関数
呼び出し	CALL 命令を使用して呼び出し、この命令の後にサブルーチン名とオプションで最大 20 個の引数を続けます。	関数の名前を指定して呼び出し、その直後に括弧を続けます。オプションで、括弧内に最大 20 個の引数を含めることができます。
戻り値	呼び出し側に値を返すこともできます。RETURN 命令で値を含めると、言語処理プログラムはその値を REXX の特殊変数 RESULT に割り当てます。	値を返さなければなりません。RETURN 命令で値を指定する必要があります。言語処理プログラムはその値で関数呼び出しを置き換えます。

サブルーチンおよび関数の作成

サブルーチンとは、プログラムが特定のタスクを実行するために呼び出す一連の命令のことです。サブルーチンを呼び出す命令は、CALL 命令です。プログラム内では、同じサブルーチンを呼び出すために CALL 命令を複数回使用することができます。

サブルーチンは終了時に、そのサブルーチン呼び出しの直後に続く命令に制御を戻すことができます。制御を戻す命令は、RETURN 命令です。



関数とは、プログラムが特定のタスクを実行して値を返すために呼び出す一連の命令のことです。[57 ページの『第 5 章 関数』](#)で説明しているように、関数には組み込み関数およびユーザー作成関数があります。

ユーザー作成関数は、組み込み関数と同じように呼び出します。つまり、関数名を指定し、その直後に括弧を続けます。括弧内には引数を含めることができます。関数名と左括弧の間に空白を入れることはできません。括弧内には、最大 20 個の引数を含めることも、引数をまったく含めないこともできます。

```
function(argument1, argument2,...)
```

または

```
function()
```

一般に、関数呼び出しは式の中で使用されるため、関数には戻り値が必要です。

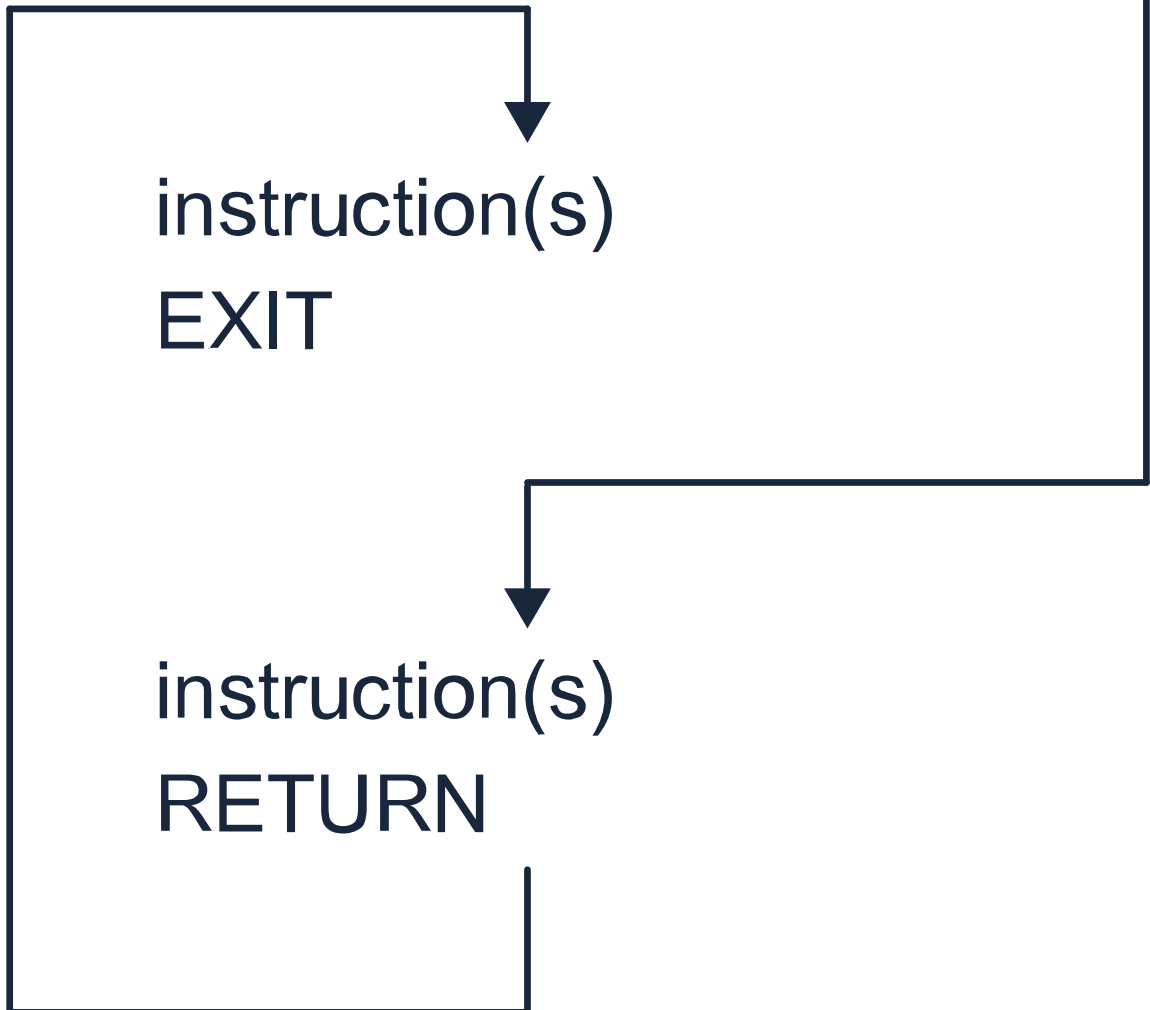
```
z = function(arguments1, argument2,...)
```

関数は終了時に、RETURN 命令を使用して、その関数呼び出しに置き換える値を戻すことができます。

instruction(s)



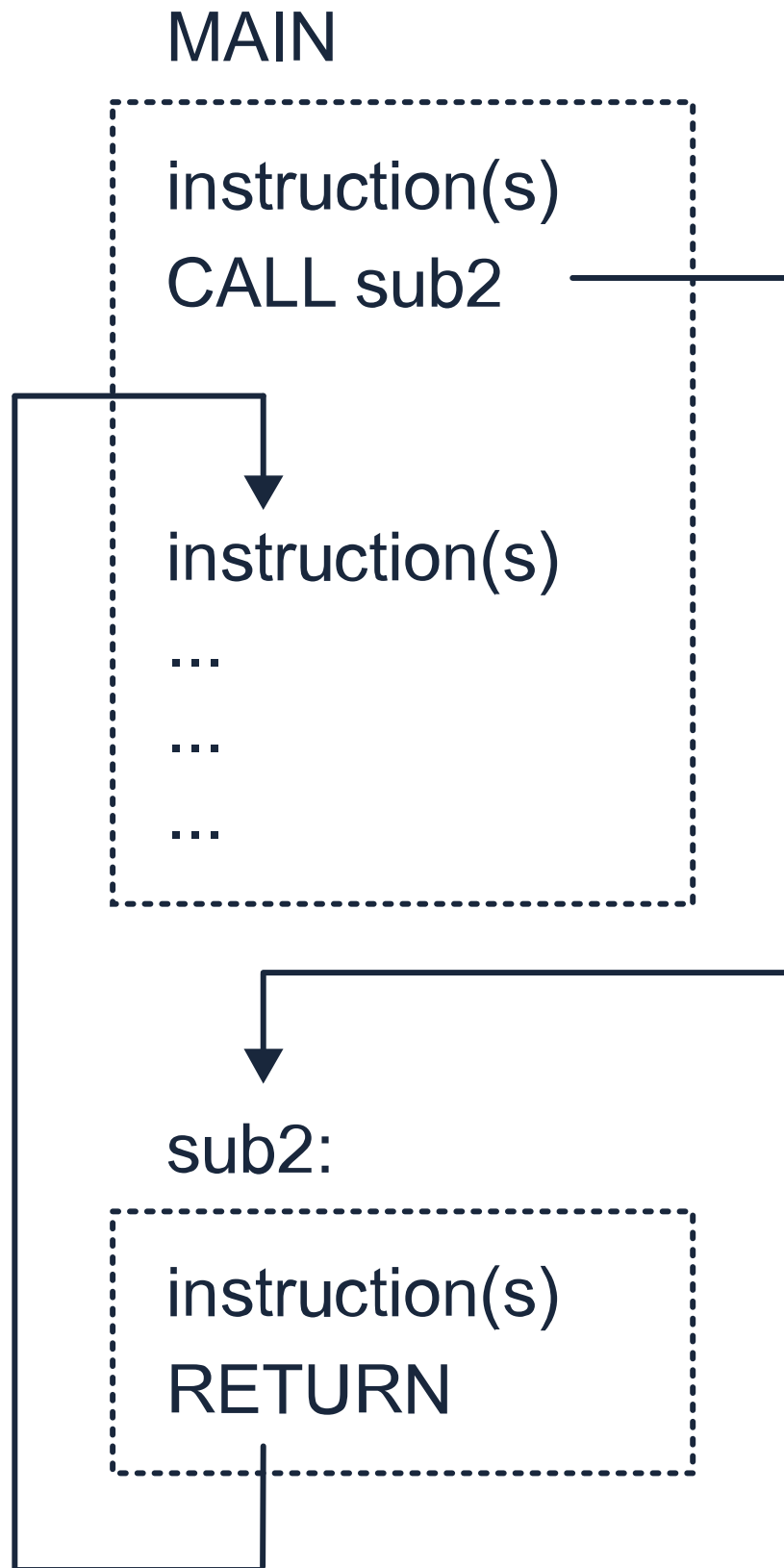
z=func1(arg1, arg2)



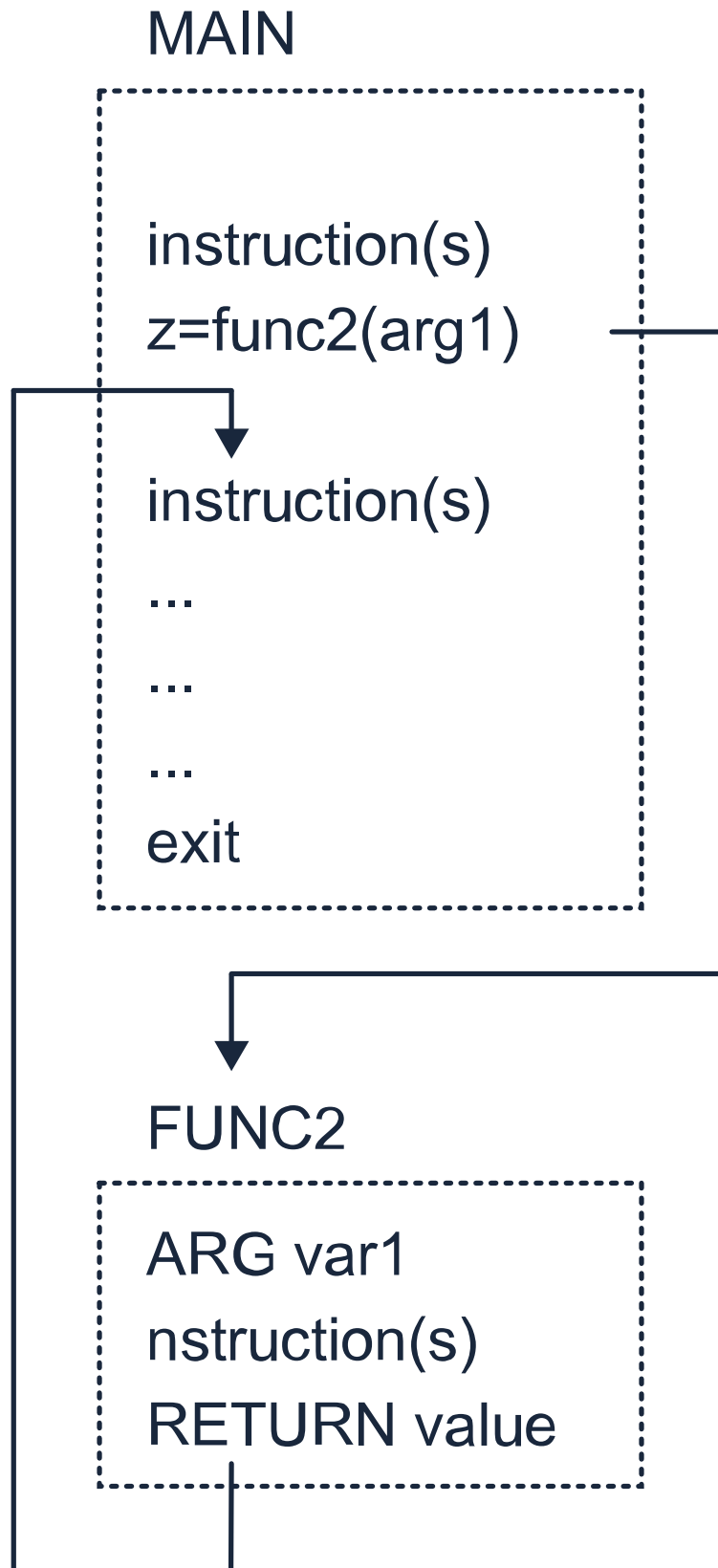
サブルーチンと関数はどちらも、内部 (ラベルでそれらを指定) または外部 (サブルーチンまたは関数を含む REXX ファイル・システムまたは区分データ・セットのメンバー名でそれらを指定) にすることができます。上記の 2 つの例は、`sub1` という名前の内部サブルーチンと `func1` という名前の内部関数を示しています。

重要: 一般に、内部サブルーチンと内部関数はプログラムの本体の後に現れるため、内部サブルーチンまたは内部関数を使用する場合は、プログラムの本体を EXIT 命令で終了することが重要です。

以下に、`sub2` という名前の外部サブルーチンを示します。



以下に、func2 という名前の外部関数を示します。



内部または外部のサブルーチンまたは関数の使用の選択

サブルーチンまたは関数を内部あるいは外部にするかどうかを判断するには、以下のような要因について考慮します。

- サブルーチンまたは関数のサイズ。通常、サイズが非常に大きいサブルーチンおよび関数は、外部にします。一方、サイズが小さければ、呼び出し側のプログラム内に簡単に収まります。
- 情報を受け渡す方法。内部サブルーチンまたは内部関数で変数を使用して情報を渡すほうが速くなります。次のトピックでは、この方法で情報を渡す方法を説明します。
- サブルーチンまたは関数が、複数のプログラムまたはユーザーにとって有用であるかどうか。その場合は、外部サブルーチンまたは外部関数にすることが推奨されます。
- パフォーマンス。関数については、言語処理プログラムは外部関数を検索する前に、内部関数を検索します。関数の完全な検索順序については、[関数およびサブルーチン](#)を参照してください。

情報の受け渡し

プログラムとその内部サブルーチンまたは内部関数は、同じ変数を共用できます。したがって、一般的に共用される変数を使用して、呼び出し側と内部サブルーチンまたは内部関数の間で情報を受け渡すことができます。また、内部サブルーチンまたは内部関数の間では、引数を使用して情報を受け渡すこともできます。

ただし、外部サブルーチンが呼び出し側と変数を共用することはできません。外部サブルーチンに情報を渡すには、引数を使用するか、データ・スタックなどの別の外部の手段を使用する必要があります。(注意: 内部関数は、関数呼び出しに続けて括弧で囲まれた引数を渡す必要はありません。ただし、内部関数と外部関数の両方を含め、すべての関数は値を返す必要があります)。

変数による情報の受け渡し

プログラムとその内部サブルーチンまたは内部関数が同じ変数を共用する場合、変数の値は最後に割り当てられた変数の値です。このことは、変数への値の割り当てがプログラムの本体で行われるのか、サブルーチンまたは関数で行われるのかには依存しません。

以下の例は、サブルーチンに情報を渡す方法を示しています。変数 `number1`、`number2`、および `answer` は共用されています。`answer` の値はサブルーチンで割り当てられて、メインプログラムで使用されます。

```
/****** REXX ******/
/* This program receives a calculated value from an internal */
/* subroutine and uses that value in a SAY instruction.      */
/*******/

number1 = 5
number2 = 10
CALL subroutine
SAY answer          /* Produces 15 */
EXIT

subroutine:
answer = number1 + number2
RETURN
```

図 26. サブルーチンを使用して変数で情報を渡す例

次の例は、サブルーチンではなく関数に情報を渡すという点を除き、上記の例と同じです。サブルーチンでは、`RETURN` 命令に変数 `answer` が含まれています。言語処理プログラムは、関数呼び出しを `answer` に格納された値で置き換えます。

```

/***** REXX *****/
/* This program receives a calculated value from an internal */
/* function and uses SAY to produce that value. */
/*****

number1 = 5
number2 = 10
SAY add()          /* Produces 15 */
SAY answer          /* Also produces 15 */
EXIT

add:
answer = number1 + number2
RETURN answer

```

図 27. 関数を使用して変数で情報を渡す例

プログラムとその内部サブルーチンまたは内部関数で同じ変数を使用すると、問題が生じることがあります。次の例では、プログラムの本体とサブルーチンが、それぞれの DO ループで同じ制御変数 *i* を使用しています。その結果、サブルーチンは *i* = 6 を設定してメイン・プログラムに戻るため、DO ループはメイン・プログラムで 1 回しか実行されないことになります。

```

/***** REXX *****/
/* NOTE: This program contains an error. */
/* It uses a DO loop to call an internal subroutine, and the */
/* subroutine uses a DO loop with the same control variable as the */
/* main program. The DO loop in the main program runs only once. */
/*****

number1 = 5
number2 = 10
DO i = 1 TO 5
  CALL subroutine
  SAY answer          /* Produces 105 */
END
EXIT

subroutine:
DO i = 1 TO 5
  answer = number1 + number2
  number1 = number2
  number2 = answer
END
RETURN

```

図 28. サブルーチンを使用して変数で情報を渡すことによって発生する問題の例

次の例は、サブルーチンではなく関数を使用して情報を渡すという点を除き、上記の例と同じです。

```

/***** REXX *****/
/* NOTE: This program contains an error. */
/* It uses a DO loop to call an internal function, and the */
/* function uses a DO loop with the same control variable as the */
/* main program. The DO loop in the main program runs only once. */
/*****

number1 = 5
number2 = 10
DO i = 1 TO 5
  SAY add()          /* Produces 105 */
END
EXIT

add:
DO i = 1 TO 5
  answer = number1 + number2
  number1 = number2
  number2 = answer
END
RETURN answer

```

図 29. 関数を使用して変数で情報を渡すことによって発生する問題の例

内部サブルーチンまたは内部関数でこのような問題を回避するには、以下の方法を使用できます。

- PROCEDURE 命令を使用します (次のセクションで説明します)。
- サブルーチンまたは関数の変数に、メイン・プログラムとは異なる名前を使用します。サブルーチンの場合、CALL 命令で引数を渡すことができます。[74 ページの『引数による情報の受け渡し』](#)を参照してください。

PROCEDURE 命令を使用した変数の保護

PROCEDURE 命令を、サブルーチンまたは関数のラベルの直後に続けると、サブルーチンまたは関数に含まれるすべての変数とそのサブルーチンまたは関数に対してローカルになります。つまり、プログラムの本体から保護されることになります。また、PROCEDURE EXPOSE 命令を使用して、指定した変数以外のすべての変数を保護することもできます。

以下の例は、サブルーチンまたは関数が PROCEDURE を使用した場合と使用しない場合の結果の違いを示しています。

```
/* ***** REXX ***** */
/* This program uses a PROCEDURE instruction to protect the */
/* variables within its subroutine. */
/* ***** */
number1 = 10
CALL subroutine
SAY number1 number2          /* Produces 10 NUMBER2 */
EXIT

subroutine: PROCEDURE
number1 = 7
number2 = 5
RETURN
```

図 30. PROCEDURE 命令を使用したサブルーチンの例

```
/* ***** REXX ***** */
/* This program does not use a PROCEDURE instruction to protect the */
/* variables within its subroutine. */
/* ***** */
number1 = 10
CALL subroutine
SAY number1 number2          /* Produces 7 5 */
EXIT

subroutine:
number1 = 7
number2 = 5
RETURN
```

図 31. PROCEDURE 命令を使用しないサブルーチンの例

次の 2 つの例は、サブルーチンではなく関数を使用している点を除いて同じです。

```
/* ***** REXX ***** */
/* This program uses a PROCEDURE instruction to protect the */
/* variables within its function. */
/* ***** */
number1 = 10
SAY pass() number2          /* Produces 7 NUMBER2 */
EXIT

pass: PROCEDURE
number1 = 7
number2 = 5
RETURN number1
```

図 32. PROCEDURE 命令を使用する関数の例

```

/***** REXX *****/
/* This program does not use a PROCEDURE instruction to protect the */
/* variables within its function. */
/***** REXX *****/
number1 = 10
SAY pass() number2          /* Produces 7 5 */
EXIT

pass:
number1 = 7
number2 = 5
RETURN number1

```

図 33. *PROCEDURE* 命令を使用しない関数の例

PROCEDURE EXPOSE を使用した変数の公開

特定の変数以外のすべての変数を保護するには、EXPOSE オプションを *PROCEDURE* 命令に指定し、その後サブルーチンまたは関数に公開したままにする変数を続けます。

次の例では、サブルーチンで *PROCEDURE EXPOSE* を使用しています。

```

/***** REXX *****/
/* This program uses a PROCEDURE instruction with the EXPOSE option */
/* to expose one variable, number1, in its subroutine. The other */
/* variable, number2, is set to null and the SAY instruction */
/* produces this name in uppercase. */
/***** REXX *****/
number1 = 10
CALL subroutine
SAY number1 number2          /* produces 7 NUMBER2 */
EXIT

subroutine: PROCEDURE EXPOSE number1
number1 = 7
number2 = 5
RETURN

```

図 34. サブルーチンで *PROCEDURE EXPOSE* を使用する例

次の例は、サブルーチンではなく関数で *PROCEDURE EXPOSE* を使用するという点を除き、上記の例と同じです。

```

/***** REXX *****/
/* This program uses a PROCEDURE instruction with the EXPOSE option */
/* to expose one variable, number1, in its function. */
/***** REXX *****/
number1 = 10
SAY pass() number1          /* Produces 5 7 */
EXIT

pass: PROCEDURE EXPOSE number1
number1 = 7
number2 = 5
RETURN number2

```

図 35. 関数で *PROCEDURE EXPOSE* を使用する例

PROCEDURE 命令について詳しくは、[PROCEDURE](#) を参照してください。

引数による情報の受け渡し

内部または外部のサブルーチンまたは関数に情報を渡す 1 つの方法は、引数を使用することです。サブルーチン呼び出しの際に、*CALL* 命令には最大 20 個の引数を指定して渡すことができます。複数の引数を渡す場合は、以下のようにコンマで区切ります。

```
CALL subroutine_name argument1, argument2, argument3,...
```

関数呼び出しでは、最大 20 個の引数をコンマで区切って渡すことができます。


```
function(argument1,argument2,argument3,...)
```

ARG 命令の使用

サブルーチンまたは関数では、ARG 命令を使用して引数を受け取ることができます。ARG 命令でも、複数の引数はコンマで区切って指定します。

```
ARG arg1, arg2, arg3, ...
```

情報は引数名ではなく位置に基づいて渡されるため、渡す引数の名前は、ARG で指定された引数の名前と同じである必要はありません。送信される最初の引数が、最初の引数として受け取られるといった具合です。CALL 命令または関数呼び出しに、テンプレートを設定することもできます。テンプレートを設定すると、言語処理プログラムは、対応する ARG 命令でそのテンプレートを使用します。テンプレートを使用した構文解析については、[83 ページの『データの解析』](#)を参照してください。

以下の例では、メインルーチンがサブルーチンに情報を送信し、サブルーチンが長方形の外周を計算します。サブルーチンは RETURN 命令に値を指定することによって、その値を変数 `perim` で返します。メインプログラムは、特殊変数 `RESULT` で値を受け取ります。

```
PARSE ARG long wide  
CALL perimeter long, wide  
SAY 'The perimeter is' RESULT 'inches.'  
EXIT
```

```
perimeter:  
  ARG length, width  
  perim = 2 * length + 2 * width  
  RETURN perim
```

The diagram illustrates the flow of arguments between the CALL statement and the perimeter procedure. A green dashed line starts from the 'long' argument in the CALL statement, goes down, then right, then down again to point at the 'length' argument in the procedure. Another green dashed line starts from the 'wide' argument in the CALL statement, goes down, then right, then down again to point at the 'width' argument in the procedure. A third green dashed line starts from the 'RESULT' in the CALL statement, goes down, then right, then down again to point at the 'RETURN' statement in the procedure. A final green dashed line starts from the 'RETURN' statement, goes down, then right, then up to point at the 'EXIT' statement in the CALL statement.

図 36. CALL 命令で引数を渡す例

次の例は、サブルーチンではなく関数で ARG を使用するという点を除き、上記の例と同じです。

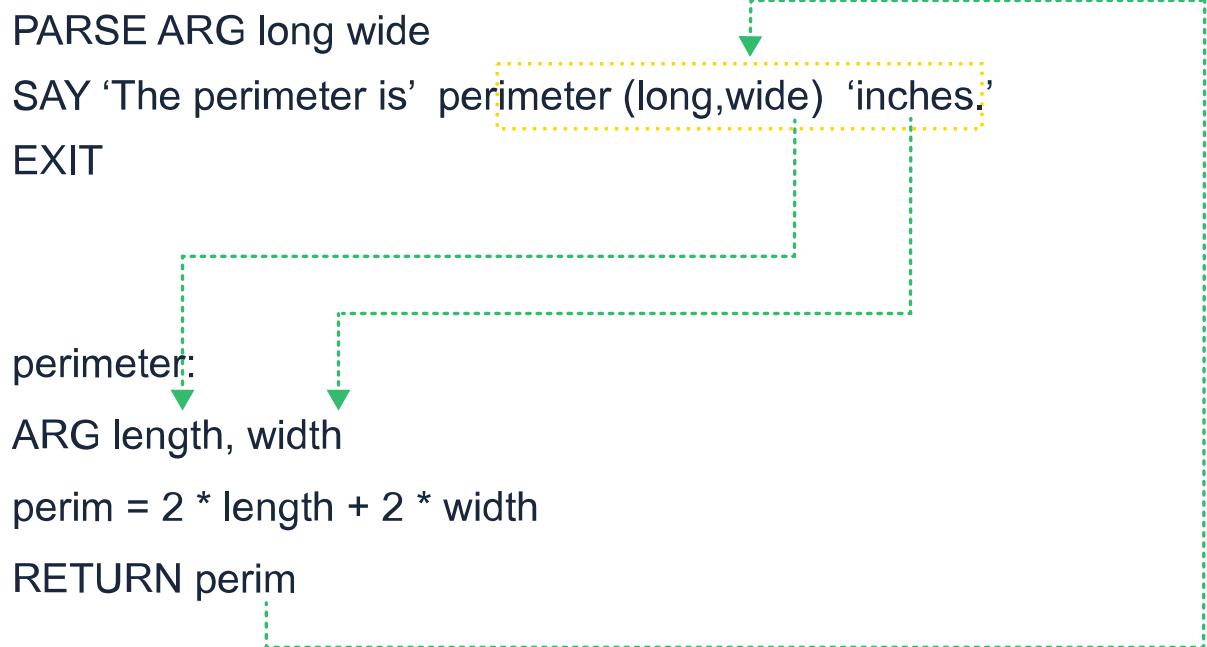


図 37. Call 命令で内部ルーチンに引数を渡す例

上記の 2 つの例で注目すべき点は、`long` と `length`、および `wide` と `width` の位置関係です。また、`perim` 変数から情報を受け取る方法にも注目してください。どちらのプログラムも、`RETURN` 命令に `perim` が含まれています。サブルーチンを使用するプログラムでは、言語処理プログラムは `perim` に格納された値を特殊変数 `RESULT` に割り当てます。関数を使用するプログラムでは、言語処理プログラムは `perimeter(long,wide)` 関数呼び出しを `perim` に格納された値で置き換えます。

ARG 組み込み関数の使用

サブルーチンまたは関数で引数を受け取るもう 1 つの方法は、`ARG` 組み込み関数を使用することです。この関数は、特定の引数の値を返します。引数の位置は、数字で表します。

例えば、前の例では `ARG` 命令の代わりに以下を使用しています。

```
ARG length, width
```

この例では、`ARG` 関数を以下のように使用できます。

```
length = ARG(1) /* puts the first argument into length */
width = ARG(2) /* puts the second argument into width */
```

`ARG` 関数について詳しくは、[ARG](#) を参照してください。

サブルーチンまたは関数からの情報の受信

サブルーチンまたは関数は最大 20 個の引数を受け取ることができますが、`RETURN` 命令で指定できる式は 1 つのみです。

次の式を指定できます。

- 数値

```
RETURN 55
```

- 値 (または値が割り当てられていない場合は、名前) が代入される 1 つ以上の変数

```
RETURN value1 value2 value3
```

- リテラル・ストリング

```
RETURN 'Work complete.'
```

- 値が代入される演算式、比較式、または論理式

```
RETURN 5 * number
```

演習: 内部および外部サブルーチンの作成

コインを投げて表か裏を当てるゲームをシミュレートし、スコアの集計結果を生成するプログラムを作成します。

考えられる入力として、以下の 4 つが必要です。

- 'HEADS'
- 'TAILS'
- '' (ゲームを終了するためのヌル)
- 上記の 3 つ以外 (正しくない応答)

引数を使用せずに、有効な入力の有無をチェックする内部サブルーチンを作成します。有効な入力を外部サブルーチンに送信します。外部サブルーチンは、`RANDOM` 組み込み関数を使用してランダムな結果を生成します。例えば、`HEADS = 0` と `TAILS = 1` の場合、`RANDOM` を以下のように使用します。

```
RANDOM(0,1)
```

有効な入力を、RANDOM から返された値と比較します。同じであれば、ユーザーが1点を獲得します。同じでなければ、コンピューターが1点を獲得します。その結果をメインプログラムに返して、結果を計算します。

解答

```

/***** REXX *****/
/* This program plays a simulated coin toss game. */
/* The input can be heads, tails, or null (") to quit the game. */
/* First an internal subroutine checks input for validity. */
/* An external subroutine uses the RANDOM built-in function to */
/* obtain a simulation of a throw of dice and compares the user */
/* input to the random outcome. The main program receives */
/* notification of who won the round. It maintains and produces */
/* scores after each round. */
/***** REXX *****/
PULL flip /* Gets "HEADS", "TAILS", or "" */
/* from input stream. */
computer = 0; user = 0 /* Initializes scores to zero */
CALL check /* Calls internal subroutine, check */
DO FOREVER
    CALL throw flip /* Calls external subroutine, throw */

    IF RESULT = 'machine' THEN /* The computer won */
        computer = computer + 1 /* Increase the computer score */
    ELSE /* The user won */
        user = user + 1 /* Increase the user score */

    SAY 'Computer score = ' computer ' Your score = ' user
    PULL flip
    CALL check /* Call internal subroutine, check */
END
EXIT

```

図 38. 考えられる解決方法 (メインプログラム)

```

/***** REXX *****/
/* This internal subroutine checks for valid input of "HEADS", */
/* "TAILS", or "" (to quit). If the input is anything else, the */
/* subroutine says the input is not valid and gets the next input. */
/* The subroutine keeps repeating until the input is valid. */
/* Commonly used variables return information to the main program */
/***** REXX *****/
check:
DO UNTIL outcome = 'correct'
    SELECT
        WHEN flip = 'HEADS' THEN
            outcome = 'correct'
        WHEN flip = 'TAILS' THEN
            outcome = 'correct'
        WHEN flip = '' THEN
            EXIT
        OTHERWISE
            outcome = 'incorrect'
            say 'Incorrect input, reply "HEADS", "TAILS" or blank'
            PULL flip
    END
END
RETURN

```

図 39. 考えられる解決方法 (CHECK という名前の内部サブルーチン)

```

/***** REXX *****/
/* This external subroutine receives the valid input, analyzes it, */
/* gets a random "flip" from the computer, and compares the two. */
/* If they are the same, the user wins. If they are different, */
/* the computer wins. The routine returns the outcome to the */
/* calling program. */
/*****

throw:
  ARG input
  IF input = 'HEADS' THEN
    userthrow = 0          /* heads = 0 */
  ELSE
    userthrow = 1          /* tails = 1 */

  compthrow = RANDOM(0,1)  /* choose a random number */
                          /* between 0 and 1 */
  IF compthrow = userthrow THEN
    outcome = 'human'      /* user chose correctly */
  ELSE
    outcome = 'machine'    /* user chose incorrectly */

  RETURN outcome

```

図 40. 考えられる解決方法 (THROW という名前の外部サブルーチン)

演習: 関数の作成

ブランクで区切られた数値のリストを受信し、これらの数値の平均を計算する AVG という名前の関数を作成します。最終的な応答は 10 進数にできます。この関数を呼び出すには、以下を使用します。

```
AVG(number1 number2 number3...)
```

組み込み関数 WORDS (WORDS 関数を参照) および WORD (WORD 関数を参照) を使用します。

ANSWER

```

/***** REXX *****/
/* This function receives a list of numbers, adds them, computes */
/* their average, and returns the average to the calling program. */
/*****

ARG numlist          /* receive the numbers in a single variable */

sum = 0              /* initialize sum to zero */

DO n = 1 TO WORDS(numlist) /* Repeat for as many times as there */
                          /* are numbers */

  number = WORD(numlist,n) /* Word #n goes to number */
  sum = sum + number       /* Sum increases by number */
END

average = sum / WORDS(numlist) /* Compute the average */

RETURN average

```

図 41. 考えられる解決方法

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料で使用されている表記規則および用語](#)も参照してください。

第 6 章 データの操作

このセクションでは、複合変数とステムを使用してデータにアクセスする方法、および構文解析の使用方法について説明します。

複合変数およびステムの使用

データを取得しやすいように、関連するデータをグループにして保管すると役立つ場合があります。例えば、従業員名のリストを配列に保管して、番号で従業員名を取得できるようにします。配列とは、1つ以上の次元でエレメントを配置するものであり、単一の名前で識別されます。例えば、employee という名前の配列に、以下のように名前を含めます。

```
EMPLOYEE
(1) Adams, Joe
(2) Crandall, Amy
(3) Devon, David
(4) Garrison, Donna
(5) Leone, Mary
(6) Sebastian, Isaac
```

一部のコンピューター言語では、エレメントの番号を使用して、配列内のエレメントにアクセスします。例えば、Adams, Joe を取得するには、employee(1) を使用します。REXX では、複合変数を使用します。

複合変数とは

REXX では、複合変数を使用して、変数の配列やリストを作成できます。例えば、複合変数である employee.1 は、語幹とテールで構成されています。

語幹とは、ピリオドで終わる記号を指します。以下に、語幹のいくつかの例を示します。

```
FRED.
Array.
employee.
```

テールは、添え字と似ています。テールは語幹の後に続き、名前の追加の部分で構成され、定数記号 (employee.1)、単純記号 (employee.n)、またはヌルになります。したがって、REXX では添え字を必ずしも数値にする必要はありません。複合変数には、少なくとも 1 つのピリオドと、ピリオドの両側に文字が必要です。以下に、複合変数の例をさらにいくつか示します。

```
FRED.5
Array.Row.Col
employee.name.phone
```

語幹の名前で代入を行うことはできませんが、テールでは代入を使用できます。以下に例を示します。

```
employee.7='Amy Martin'
new=7
employee.new='May Davis'
say employee.7          /* Produces: May Davis */
```

他の REXX 変数と同様に、テールの変数に割り当て済みの値がなければ、その変数の名前が大文字で使われます。

```
first = 'Fred'
last = 'Higgins'
name = first.last          /* NAME is assigned FIRST.Higgins */
                           /* The value FIRST appears because the */
                           /* variable FIRST is a stem, which */
                           /* cannot change. */
SAY name.first.middle.last /* Produces NAME.Fred.MIDDLE.Higgins */
```

複合変数のグループを初期化して配列を設定するには、DO ループを使用できます。

```
DO i = 1 TO 6
  PARSE PULL employee.i
END
```

employee 配列の例で使用されている名前と同じ名前を使用する場合、以下のような複合変数のグループになります。

```
employee.1 = 'Adams, Joe'
employee.2 = 'Crandall, Amy'
employee.3 = 'Devon, David'
employee.4 = 'Garrison, Donna'
employee.5 = 'Leone, Mary'
employee.6 = 'Sebastian, Isaac'
```

名前を複合変数のグループに含めると、名前の番号、または名前の番号が表す変数を使用して、簡単に名前にアクセスできるようになります。

```
name = 3
SAY employee.name      /* Produces 'Devon, David' */
```

複合変数について詳しくは、[複合記号](#)を参照してください。

ステムの使用

複合変数进行处理する際は、変数のコレクション全体を同じ値に初期化すると役立つ場合がよくあります。このような初期化は、ステムを含めた代入を使用することで簡単に行うことができます。例えば、`number.=0`を使用すると、`number.`という名前の配列に含まれるすべての配列エレメントが0に初期化されます。

同じようにして、配列に含まれるすべての複合変数の値を変更できます。例えば、`employee` 名を `Nobody` に変更するには、以下の代入命令を使用します。

```
employee. = 'Nobody'
```

これにより、ステム `employee.` で始まるすべての複合変数に、以前に値が割り当てられているかどうかに関わらず、値 `Nobody` が設定されます。ステムを割り当てた後は、個々の複合変数に新しい値を割り当てることができます。

```
employee.='Nobody'
SAY employee.5      /* Produces 'Nobody' */
SAY employee.10     /* Produces 'Nobody' */
SAY employee.oldest /* Produces 'Nobody' */

employee.new = 'Clark, Evans'
SAY employee.new    /* Produces 'Clark, Evans' */
```

ファイルの読み取りや書き込みを行う場合は、ステムと併せて EXECIO コマンドおよび RFS コマンドを使用できます。[EXECIO](#) および [RFS](#) を参照してください。CICS では、RFS が推奨される入出力方式です。

演習: 複合変数およびステムの使用

1. 次の割り当て命令の後、SAY 命令はどのような出力を生成しますか。

```
a = 3          /* assigns '3' to variable 'A' */
d = 4          /* '4' to 'D' */
c = 'last'     /* 'last' to 'C' */
a.d = 2        /* '2' to 'A.4' */
a.c = 5        /* '5' to 'A.last' */
z.a.d = 'cv3d' /* 'cv3d' to 'Z.3.4' */
```

- a. SAY a
- b. SAY D
- c. SAY c
- d. SAY a.a

- e. SAY A.D
- f. SAY d.c
- g. SAY c.a
- h. SAY a.first
- i. SAY z.a.4

2. 次の割り当て命令の後、SAY 命令はどのような出力を生成しますか。

```
hole.1 = 'full'
hole. = 'empty'
hole.s = 'full'
```

- a. SAY hole.1
- b. SAY hole.s
- c. SAY hole.mouse

応答

1. a. 3
b. 4
c. (最後)
d. A.3
e. 2
f. D.last
g. C.3
h. A.FIRST
i. cv3d
2. a. 空
b. 完全
c. 空

データの解析

解析とは、データを分割して、その構成部分を 1 つ以上の変数に割り当てることです。解析では、データに含まれる各ワードを変数に割り当てることも、データをより小さい部分に分割することもできます。データを列にフォーマット設定する際も、解析が役立ちます。

データを受け取る変数は、テンプレートで指定されます。テンプレートとは、データの分割方法を指示するモデルのことです。テンプレートは、データを受け取る変数のリストといった単純なものにすることができます。より複雑なテンプレートには、パターンを含めることができます。[86 ページの『パターンによる解析』](#)を参照してください。

構文解析命令

REXX 構文解析命令には、PULL、ARG、および PARSE があります。(PARSE には、いくつかのバリエーションがあります)。

PULL 命令

PULL は、入力を読み取り、1 つ以上の変数にそれを割り当てる命令です。プログラム・スタックに情報が含まれている場合、PULL 命令はそのプログラム・スタックから情報を取得します。プログラム・スタックが空の場合、PULL は現在の端末入力装置から情報を取得します。データ・スタックについては、[14 ページの『プログラム・スタックまたは端末入力装置からの情報の取得』](#)を参照してください。

```
/* This REXX program parses the string "Knowledge is power." */
PULL word1 word2 word3
/* word1 contains 'KNOWLEDGE' */
```

```
/* word2 contains 'IS' */
/* word3 contains 'POWER.' */
```

PULL は、文字情報を変数に割り当てる前に大文字に変換します。大文字に変換したくない場合は、PARSE PULL 命令を使用します。

```
/* This REXX program parses the string: "Knowledge is power." */
PARSE PULL word1 word2 word3
/* word1 contains 'Knowledge' */
/* word2 contains 'is' */
/* word3 contains 'power.' */
```

オプションのキーワード UPPER は、PARSE 命令の任意のバリエーションに含めることができます。これにより、言語処理プログラムは、文字情報を変数に割り当てる前に大文字に変換します。例えば、PARSE UPPER PULL を使用すると、PULL を使用した場合と同じ結果になります。

ARG 命令

ARG 命令は、プログラム、関数、またはサブルーチンに引数として渡された情報を取得し、その情報を 1 つ以上の変数に格納します。Knowledge is power. という 3 つの引数を sample という名前の REXX プログラムに渡すには、以下のようにします。

1. プログラムを呼び出して、引数を exec 名の後に続くストリングとして渡します。

```
REXX sample Knowledge is power.
```

2. ARG 命令を使用し、3 つの引数を受け取って変数に取り込みます。

```
/* SAMPLE -- A REXX program using ARG */
ARG word1 word2 word3
/* word1 contains 'KNOWLEDGE' */
/* word2 contains 'IS' */
/* word3 contains 'POWER.' */
```

ARG は、引数を変数に割り当てる前に大文字に変換します。

大文字に変換したくない場合は、ARG ではなく PARSE ARG 命令を使用します。

```
/* REXX program using PARSE ARG */
PARSE ARG word1 word2 word3
/* word1 contains 'Knowledge' */
/* word2 contains 'is' */
/* word3 contains 'power.' */
```

PARSE UPPER ARG は、ARG と同じ結果を返します。この命令は、文字情報を変数に割り当てる前に大文字に変換します。

PARSE VALUE ... WITH 命令

PARSE VALUE...WITH 命令は、指定された式 (リテラル・ストリングなど) を、WITH サブキーワードの後に名前が指定された 1 つ以上の変数に解析します。

```
PARSE VALUE 'Knowledge is power.' WITH word1 word2 word3
/* word1 contains 'Knowledge' */
/* word2 contains 'is' */
/* word3 contains 'power.' */
```

PARSE VALUE は、文字情報を変数に割り当てる前に大文字に変換しません。大文字に変換する必要がある場合は、PARSE UPPER VALUE を使用します。PARSE VALUE では、ストリングの代わりに変数を使用することもできます (その場合は、最初に変数に値を割り当てます)。

```
string='Knowledge is power.'
PARSE VALUE string WITH word1 word2 word3
/* word1 contains 'Knowledge' */
/* word2 contains 'is' */
/* word3 contains 'power.' */
```

または、PARSE VAR を使用して変数を解析することもできます。

PARSE VAR 命令

PARSE VAR 命令は、指定された変数を 1 つ以上の変数に解析します。

```
quote = 'Knowledge is power.'  
PARSE VAR quote word1 word2 word3  
/* word1 contains 'Knowledge' */  
/* word2 contains 'is' */  
/* word3 contains 'power.' */
```

PARSE VAR は、文字情報を変数に割り当てる前に大文字に変換しません。大文字に変換する必要がある場合は、PARSE UPPER VAR を使用します。

ワードへの解析についての詳細

構文解析するデータ内のワード数がテンプレート内の変数の数と同じではない場合の動作について説明します。

前のトピックの例では、構文解析するデータ内のワード数がテンプレート内の変数の数と常に同じでした。構文解析では、テンプレート内で指定されたすべての変数に常に新しい値が割り当てられます。解析するデータに含まれるワードの数よりも多くの変数名がある場合、残った変数はヌル (空の) 値を受け取ります。解析するデータに含まれるワードの数のほうがテンプレート内の変数名より多い場合、各変数に順番にデータの 1 つのワードが取り込まれ、最後の変数に残りのデータのすべてが取り込まれます。

以下の例では、テンプレート内にデータのワード数より多くの変数名があります。したがって、残りの変数にはヌル値が入ります。

```
PARSE VALUE 'Extra variables' WITH word1 word2 word3  
/* word1 contains 'Extra' */  
/* word2 contains 'variables' */  
/* word3 contains '' */
```

以下の例では、データに含まれるワードの数が、テンプレート内の変数名の数を超えています。最後の変数名には、複数のワードと、場合によっては前後に空白が含まれる可能性があります。

```
PARSE VALUE 'More words in data' WITH var1 var2 var3  
/* var1 contains 'More' */  
/* var2 contains 'words' */  
/* var3 contains ' in data' */
```

ワードに解析すると、通常は、各ワードを変数に取り込む前にワードに含まれる前後の空白が削除されます。ただし、データを最後の変数に取り込む際は、解析によって、分離文字としてワードを区切る空白が 1 つ削除されるだけで、残りのワードの前後の空白は残ったままになります。上記の例では、words の前に先行空白が 2 つあります。解析により、'words' を var2 に取り込む前に、ワードを区切る両方の空白と余分な先行空白が削除されます。in の前には、4 つの先行空白があります。var3 は最後の変数であるため、解析によりワードを区切る空白は削除されますが、余分な先行空白は保持されます。したがって、var3 は、' in data' (3 つの先行空白を含む) を受け取ります。

テンプレート内のピリオドは、プレースホルダーとして機能します。これはデータを受け取りません。不要な情報を収集するには、変数のグループ内やテンプレートの最後でピリオドを「ダミー変数」として使用できます。

```
string='Example of using placeholders to discard junk'  
PARSE VAR string var1 . var2 var3 .  
/* var1 contains 'Example' */  
/* var2 contains 'using' */  
/* var3 contains 'placeholders' */  
/* The periods collect the words 'of' and 'to discard junk' */
```

構文解析命令について詳しくは、[PARSE](#) を参照してください。

パターンによる解析

最も単純なテンプレートは、ブランクで区切られた変数名のグループです。これにより、データはブランクで区切られたワードに解析されます。テンプレートにはパターンを含めることもできます。パターンは、ストリング、数値、またはこのいずれかを表す変数にすることができます。

String

テンプレート内でストリングを使用すると、解析では、一致するストリングのために入力データがチェックされます。データを変数に割り当てると、解析では一般に、テンプレート内のストリングと一致する入力ストリングの部分がスキップされます。

```
phrase = 'To be, or not to be?'      /* phrase containing comma */
PARSE VAR phrase part1 ',' part2    /* template containing comma */
                                   /* as string separator */
                                   /*
/* part1 contains 'To be'          */
/* part2 contains ' or not to be?' */
```

この例では、コンマはストリング・セパレーターであるため、'To be' にコンマが含まれていないことに注意してください。part2 に、ブランクで始まる値が含まれていることにも注意してください。解析により、入力ストリングは一致するテキストがある場所で分割されます。一致が開始するまでの部分のデータが1つの変数に取り込まれ、一致が終わった後からのデータが次の変数に取り込まれます。

可変

テンプレート内でセパレーターとして指定するストリングが事前にわからない場合、括弧で囲んだ変数を使用できます。

```
separator = ','
phrase = 'To be, or not to be?'
PARSE VAR phrase part1 (separator) part2
/* part1 contains 'To be'          */
/* part2 contains ' or not to be?' */
```

この例でも、コンマがストリング・セパレーターとして使用されているため、'To be' にはコンマが含まれていません。

Number

テンプレート内で数値を使用して、データを分離する桁を指定できます。符号なし整数は、桁の絶対位置を示します。符号付き整数は、桁の相対位置を示します。

符号なし整数または等号(=)の接頭部が付いた整数は、データを桁の絶対位置で分割します。最初のセグメントは1桁目で始まり、指定された桁番号まで(ただし、その桁内の情報を含めずに)続きます。後続のセグメントは、指定された番号の桁から始まります。

```
quote = 'Ignorance is bliss.'
.....1.....2
PARSE VAR quote part1 5 part2
/* part1 contains 'Igno'          */
/* part2 contains 'rance is bliss.' */
```

以下のコードも上記と同じ結果になります。

```
quote = 'Ignorance is bliss.'
.....1.....2
PARSE VAR quote 1 part1 =5 part2
/* part1 contains 'Igno'          */
/* part2 contains 'rance is bliss.' */
```

数字パターン1の指定はオプションです。数字パターンを使用して解析の開始点を指定しない場合、開始点はデフォルトで1に設定されます。この例は、数字パターン5は=5と同じであることも示しています。

テンプレートに複数の数字パターンがあり、後のものが前のものより小さい場合、解析は小さい数によって指定される桁にループバックします。

```
quote = 'Ignorance is bliss.'
.....1.....2

PARSE VAR quote part1 5 part2 10 part3 1 part4
/* part1 contains 'Igno' */
/* part2 contains 'rance' */
/* part3 contains 'is bliss.' */
/* part4 contains 'Ignorance is bliss.' */
```

テンプレート内の各変数の前と後ろの両方に桁番号が指定されている場合、これら 2 つの番号は変数のデータ開始とデータ終了をそれぞれ示します。

```
quote = 'Ignorance is bliss.'
.....1.....2

PARSE VAR quote 1 part1 10 11 part2 13 14 part3 19 1 part4 20
/* part1 contains 'Ignorance' */
/* part2 contains 'is' */
/* part3 contains 'bliss' */
/* part4 contains 'Ignorance is bliss.' */
```

したがって、数字パターンを使用してデータの一部をスキップすることもできます。

```
quote = 'Ignorance is bliss.'
.....1.....2

PARSE VAR quote 2 var1 3 5 var2 7 8 var3 var 4 var5
SAY var1||var2||var3 var4 var5 /* || means concatenate */
/* Says: grace is bliss. */
```

テンプレート内の符号付き整数は、桁の相対位置に従ってデータを分割します。正負号または負符号はそれぞれ、開始点から右または左への移動を示します。次の例では、**part1** が 1 桁目で開始することに注意してください (開始点を指定する数値がないため、デフォルトの開始点が使用されます)。

```
quote = 'Ignorance is bliss.'
.....1.....2

PARSE VAR quote part1 +5 part2 +5 part3 +5 part4
/* part1 contains 'Ignor' */
/* part2 contains 'ance' */
/* part3 contains 'is bl' */
/* part4 contains 'iss.' */
```

+5 part2 は、解析で **part2** に 6 桁目 ($1+5=6$) から始まるデータを取り込むことを意味します。
+5 part3 は、**part3** に 11 桁目 ($6+5=11$) から始まるデータを取り込むことを意味するといった具合です。負符号を使用する場合も、正符号を使用する場合と同様です。負符号は、データ・ストリング内の相対位置を識別します。負符号により、データ・ストリング内で「後退」(左へ移動)することになります。

```
quote = 'Ignorance is bliss.'
.....1.....2

PARSE VAR quote part1 +10 part2 +3 part3 -3 part4
/* part1 contains 'Ignorance' */
/* part2 contains 'is' */
/* part3 contains 'bliss.' */
/* part4 contains 'is bliss.' */
```

この例では、**part1** が 1 桁目 (デフォルト) で始まる文字を受け取ります。**+10 part2** は、11 桁目 ($1+10=11$) で始まる文字を受け取ります。**+3 part3** は、14 桁目 ($11+3=14$) で始まる文字を受け取ります。**-3 part4** は、11 桁目 ($14-3=11$) で始まる文字を受け取ります。

より柔軟性を持たせるには、解析命令で変数数字パターンを定義して使用することができます。それにはまず、解析命令に先立って、変数を符号なし整数として定義します。次に、括弧で囲んだ変数を解析命令に含め、左括弧の前に以下のいずれかを指定します。

- 右方向に桁を移動することを指定する場合は、正符号 (+)。
- 左方向に桁を移動することを指定する場合は、負符号 (-)。
- 桁の絶対位置を指定する場合は、等号 (=)。

(左括弧の前に +、-、または = がないと、言語処理プログラムは変数を文字列・パターンと見なしません)。以下の例では、変数文字列パターン `movex` を使用しています。

```
quote = 'Ignorance is bliss.'
      ....+....1....+....2

movex = 3                                /* variable position */
PARSE VAR quote part5 +10 part6 +3 part7 -(movex) part8
/* part5 contains 'Ignorance' */
/* part6 contains 'is' */
/* part7 contains 'bliss.' */
/* part8 contains 'is bliss.' */
```

構文解析について詳しくは、[構文解析](#)を参照してください。

引数としての複数の文字列の構文解析

引数を関数またはサブルーチンに渡す際は、構文解析の対象とする複数の文字列を指定できます。引数を構文解析する命令には、`ARG`、`PARSE ARG`、および `PARSE UPPER ARG` 命令があります。これらの構文解析命令のみが、複数の文字列を処理します。

複数の文字列を渡すには、コンマを使用して隣接する文字列を区切ります。

次の例では、3 つの引数を内部サブルーチンに渡します。

```
CALL sub2 'String One', 'String Two', 'String Three'
:
:
EXIT

sub2:
PARSE ARG word1 word2 word3, string2, string3
/* word1 contains 'String' */
/* word2 contains 'One' */
/* word3 contains '' */
/* string2 contains 'String Two' */
/* string3 contains 'String Three' */
```

最初の引数は、2 ワード `String One` であり、`word1`、`word2`、`word3` という 3 つの変数名に解析されます。この引数に 3 番目のワードはないため、3 番目の変数 `word3` はヌルに設定されます。2 番目と 3 番目の引数は、引数全体が変数名 `string2` と `string3` にそれぞれ解析されます。

プログラムまたはサブルーチンに渡された複数の引数の構文解析について詳しくは、[複数の文字列の構文解析](#)を参照してください。

演習 - 構文解析の練習

次の構文解析の例では、どのような結果になりますか。

- ```
quote = 'Experience is the best teacher.'
PARSE VAR quote word1 word2 word3
```

  - `word1 =`
  - `word2 =`
  - `word3 =`
- ```
quote = 'Experience is the best teacher.'
PARSE VAR quote word1 word2 word3 word4 word5 word6
```

 - `word1 =`
 - `word2 =`
 - `word3 =`
 - `word4 =`
 - `word5 =`
 - `word6 =`

3.

```
PARSE VALUE 'Experience is the best teacher.' WITH word1 word2 . .  
word3
```

 - a) word1 =
 - b) word2 =
 - c) word3 =
4.

```
PARSE VALUE 'Experience is the best teacher.' WITH v1 5 v2  
....+....1....+....2....+....3.
```

 - a) v1 =
 - b) v2 =
5.

```
quote = 'Experience is the best teacher.'  
....+....1....+....2....+....3.  
  
PARSE VAR quote v1 v2 15 v3 3 v4
```

 - a) v1 =
 - b) v2 =
 - c) v3 =
 - d) v4 =
6.

```
quote = 'Experience is the best teacher.'  
....+....1....+....2....+....3.  
  
PARSE UPPER VAR quote 15 v1 +16 =12 v2 +2 1 v3 +10
```

 - a) v1 =
 - b) v2 =
 - c) v3 =
7.

```
quote = 'Experience is the best teacher.'  
....+....1....+....2....+....3.  
  
PARSE VAR quote 1 v1 +11 v2 +6 v3 -4 v4
```

 - a) v1 =
 - b) v2 =
 - c) v3 =
 - d) v4 =
8.

```
first = 7  
quote = 'Experience is the best teacher.'  
....+....1....+....2....+....3.  
  
PARSE VAR quote 1 v1 =(first) v2 +6 v3
```

 - a) v1 =
 - b) v2 =
 - c) v3 =
9.

```
quote1 = 'Knowledge is power.'  
quote2 = 'Ignorance is bliss.'  
quote3 = 'Experience is the best teacher.'  
CALL sub1 quote1, quote2, quote3  
EXIT  
  
sub1:  
PARSE ARG word1 . . , word2 . . , word3 .
```

 - a) word1 =
 - b) word2 =
 - c) word3 =

応答

1. a) word1 = Experience
b) word2 = is
c) word3 = the best teacher.
2. a) word1 = Experience
b) word2 = is
c) word3 = the
d) word4 = best
e) word5 = teacher.
f) word6 = "
3. a) word1 = Experience
b) word2 = is
c) word3 = teacher.
4. a) v1 = Expe
b) v2 = rience is the best teacher.
5. a) v1 = Experience
b) v2 = is (v2 に is が含まれていることに注意)
c) v3 = the best teacher.
d) v4 = perience is the best teacher.
6. a) v1 = THE BEST TEACHER
b) v2 = IS
c) v3 = EXPERIENCE
7. a) v1 = 'Experience '
b) v2 = 'is the'
c) v3 = ' best teacher.'
d) v4 = ' the best teacher.'
8. a) v1 = 'Experi'
b) v2 = 'ence i'
c) v3 = 's the best teacher.'
9. a) word1 = Knowledge
b) word2 = Ignorance
c) word3 = Experience

第7章 プログラムからのコマンドの使用

このセクションでは、REXX プログラムでのコマンドの使用について説明します。

コマンドの主なカテゴリーは以下のとおりです。

REXX/CICS コマンド

これらのコマンドを使用して、各種の REXX/CICS 機能にアクセスできます。[REXX/CICS コマンド](#)を参照してください。

CICS コマンド

これらのコマンドは、アプリケーション・プログラムが CICS サービスにアクセスするために使用する EXEC CICS コマンドを実装します。[CICS コマンド・サマリー](#)を参照してください。

制限付きコマンドおよびキーワードが CICS TS 5.5 以降で定義されている場合、それらの制限は REXX プログラムから呼び出される EXEC CICS コマンドには適用されないことに注意してください。詳細については、[特定の CICS API コマンドと SPI コマンドの使用を制御する](#)を参照してください。

SQL ステートメント

これらのステートメントは、動的に準備されて実行されます。[REXX/CICS DB2 インターフェース](#)を参照してください。

EDIT コマンド

これらのコマンドは、REXX/CICS マクロからエディター機能呼び出します。[REXX/CICS テキスト・エディター](#)を参照してください。

RFS コマンド

これらのコマンドは、REXX ファイル・システム (RFS) 用です。[REXX/CICS ファイル・システム](#)を参照してください。

RLS コマンド

これらのコマンドは、REXX リスト・システム (RLS) 用です。[REXX/CICS List System](#)を参照してください。

プログラムがコマンドを発行すると、REXX の特殊変数 RC が戻りコードに設定されます。プログラムでは、この戻りコードを使用して、プログラム内での一連の動作を決定できます。RC は、コマンドが発行されるたびに設定されます。したがって、RC には最後に発行されたコマンドの戻りコードが含まれます。

コマンドでの引用符の使用

一般に、コマンドを他のタイプの命令と差別化するには、単一引用符または二重引用符でコマンドを囲みます。コマンドが引用符で囲まれていないと、式として処理されるため、エラーで終了する可能性があります。例えば、言語処理プログラムはアスタリスク (*) を乗算演算子として扱います。

多くの CICS コマンドは、コマンド内で単一引用符を使用します。このため、当然のことですが、CICS コマンドは二重引用符で囲むことを推奨します。

以下の例は、ワード test を一時記憶域キュー ABC に入れます。

```
"CICS WRITEQ TS QUEUE('ABC') FROM('test')"
```

コマンドでの変数の使用

コマンドに変数が含まれる場合、その変数が引用符で囲まれていると、変数の値は代入されません。言語処理プログラムは、変数が引用符の外部にある場合にのみ、変数の値を使用します。

コマンドとしての別の REXX プログラムの呼び出し

外部ルーチンとして別のプログラムを呼び出すことも、EXEC コマンドを使用して別のプログラムから明示的にプログラムを呼び出すこともできます。

62 ページの『サブルーチンと関数』では、別のプログラムを外部ルーチンとして呼び出す方法を説明しました。EXEC コマンドを使用して、プログラムから明示的に別のプログラムを呼び出すこともできます。外部ルーチンと同様に、明示的または暗黙的に呼び出されたプログラムは、RETURN または EXIT 命令を使用して、呼び出し側に値を返すことができます。外部ルーチンは特殊変数 RESULT に値を渡しますが、それとは異なり、呼び出されたプログラムは REXX の特殊変数 RC に値を渡します。

プログラム内から明示的に別のプログラムを呼び出すには、他の REXX/CICS コマンドと同様に EXEC コマンドを使用します。呼び出されたプログラムを終了するには、呼び出し側に制御が戻されるよう、RETURN または EXIT 命令を使用する必要があります。REXX の特殊変数 RC は、EXEC コマンドからの戻りコードに設定されます。オプションで、RETURN または EXIT 命令で値を呼び出し側に返すこともできます。制御が呼び出し側に再び戻される際に、REXX の特殊変数 RC は、RETURN または EXIT 命令で返される式の値に設定されます。

例えば、CALC という名前のプログラムを呼び出して、このプログラムに 4 つの数値からなる引数を渡すには、以下の命令を含めることができます。

```
"EXEC calc 24 55 12 38"  
SAY 'The result is' RC
```

CALC には、以下の命令が含まれることが考えられます。

```
ARG number1 number2 number3 number4  
answer = number1 * (number2 + number3) - number4  
RETURN answer
```

プログラムからのコマンドの発行

ホスト・コマンド環境の内容、ホスト・コマンド環境にコマンドを渡す方法、およびホスト・コマンド環境の変更方法について説明します。

このタスクについて

コマンドを実行するための環境は、ホスト・コマンド環境と呼ばれます。プログラムが実行される前に、コマンドを処理するアクティブなホスト・コマンド環境が定義されます。言語処理プログラムはコマンドを検出すると、そのコマンドを処理するためにホスト・コマンド環境に渡します。

REXX プログラムがホスト・システムで実行される場合、コマンドを処理するための環境として、少なくとも 1 つのデフォルト環境を使用できます。

ホスト・コマンド環境は以下のとおりです。

REXXCICS

デフォルトの REXX/CICS コマンド環境です。この環境からは、すべての REXX/CICS、SQL、EDIT、RFS、または RLS コマンドを発行できます。ただし、CICS コマンドには接頭部 CICS、SQL ステートメントには接頭部 EXECSQL、EDIT コマンドには接頭部 EDITSVR、RFS コマンドには接頭部 RFS、RLS コマンドには接頭部 RLS を付加する必要があります。

CICS

CICS コマンドのみを発行する、オプションの環境です。ホスト・コマンド・ストリングの最初のワードは、コマンド名です (例: SEND、RECEIVE)。

EXECSQL

SQL ステートメント (SELECT) を CICS/Db2® インターフェースに発行する、オプションの環境です。

EDITSVR

編集セッションを作成する、オプションの環境です。

FLSTSVR

ファイル・リスト・ユーティリティー用のコマンドを実行する、オプションの環境です。

RFS

REXX ファイル・システム用のコマンドを実行する、オプションの環境です。

RLS

REXX リスト・システム用のコマンドを実行する、オプションの環境です。

注: すべてのコマンドにデフォルトの環境 REXXCICS を使用すること (つまり、ADDRESS 命令を指定しないこと) をお勧めします。

コマンドがホスト環境に渡される仕組み

言語処理プログラムは、REXX プログラム内の各式を評価します。この評価結果は、文字ストリングになります (ヌル・ストリングになる場合もあります)。文字ストリングは適宜準備されてホスト・コマンド環境に送信されます。ホスト・コマンド環境はストリングをコマンドとして処理し、処理が完了すると、制御を言語処理プログラムに戻します。ストリングが現在のホスト・コマンド環境に有効なコマンドではない場合は、障害が発生し、ホスト・コマンド環境からの戻りコードが特殊変数 RC に格納されます。

ホスト・コマンド環境の変更

ホスト・コマンド環境は、デフォルト環境 (REXXCICS) または以前に設定した環境から、別の環境に変更できます。

ホスト・コマンド環境を変更するには、ADDRESS 命令を使用し、命令の後に続けて環境の名前を指定します。

ADDRESS 命令には 2 つの形式があり、一方は単一のコマンドにのみ影響し、もう一方はこの命令の後に発行されたすべてのコマンドに影響します。

- 単一のコマンド

ADDRESS 命令でホスト・コマンド環境の名前とコマンドの両方を指定すると、指定されたコマンドのみが指定された環境に送信されます。このコマンドが完了すると、以前のホスト・コマンド環境が再びアクティブになります。

- すべてのコマンド

ADDRESS 命令でホスト・コマンド環境の名前のみを指定すると、そのホスト・コマンド環境内で以降に発行されるすべてのコマンドが、その環境のコマンドとして処理されます。

この ADDRESS 命令は、この命令を使用するプログラムのホスト・コマンド環境にのみ影響します。プログラムが外部ルーチンを呼び出す場合、その呼び出し側プログラムのホスト・コマンド環境に関係なく、デフォルトのホスト・コマンド環境が使用されます。元のプログラムに戻った時点で、ADDRESS 命令が以前に設定したホスト・コマンド環境が再開されます。

アクティブなホスト・コマンド環境の判別

現在アクティブになっているホスト・コマンド環境を調べるには、ADDRESS 組み込み関数を使用します。例えば、次のようになります。

```
curenv = ADDRESS()
```

この例では、curenv はアクティブなホスト・コマンド環境 (例えば、REXXCICS) に設定されます。

第 8 章 プログラム内の問題の診断

プログラムでエラーが発生した場合、そのエラーを突きとめるには、いくつかの方法があります。

- TRACE 命令は、言語処理プログラムが各演算を評価する方法を示します。TRACE 命令を使用して式を評価する方法については、[95 ページの『TRACE 命令による式のトレース』](#)を参照してください。TRACE 命令を使用してホスト・コマンドを評価する方法については、[97 ページの『TRACE 命令によるコマンドのトレース』](#)を参照してください。
- REXX/CICS は、特殊変数 RC および SIGL を以下のように設定します。

RC

コマンドからの戻りコードを示します。

SIGL

関数呼び出し、SIGNAL 命令、または CALL 命令によって制御の移動が発生した行番号を示します。

TRACE 命令による式のトレース

TRACE 命令を使用することで、言語処理プログラムが式を読み取る際に式の各演算をどのように評価するかを表示したり、式の最終結果を表示したりできます。この 2 つのタイプのトレースは、プログラムをデバッグする際に役立ちます。

演算のトレース

式に含まれる演算をトレースするには、TRACE 命令の TRACE I (TRACE Intermediates) 形式を使用します。言語処理プログラムは、以下のように、この命令の後に続くすべての式を構成部分ごとに分割して分析します。

>V> 変数値

トレース・データは、変数の内容です。

>L> リテラル値

トレース・データは、リテラル (ストリング、初期化されていない変数、または定数) です。

>O> 演算結果

トレース・データは、2 つの項の演算結果です。

以下の例では、TRACE I 命令を使用しています。(行番号はプログラムには含まれていません。これらの行番号は、例の後に続く説明を容易にするために使用しているものです)。

```
1 /***** REXX *****/
2 /* This program uses the TRACE instruction to show how */
3 /* an expression is evaluated, operation by operation. */
4 /*****/
5 a = 9
6 y = 2
7 TRACE I
8
9 IF a + 1 > 5 * y THEN
10   SAY 'a is big enough.'
11 ELSE NOP /* No operation on the ELSE path */
```

図 42. REXX による式の評価方法を表示する TRACE

この例を実行すると、SAY 命令が以下の結果を出力します。

```
9 *- IF a + 1 > 5 * y
>V> "9"
>L> "1"
>O> "10"
>L> "5"
>V> "2"
>O> "10"
>O> "0"
```

9 は行番号です。*-* は、この後続く部分 (IF a + 1 < 5 * y) がプログラムからのデータであることを意味します。残りの行は、すべての式を構成部分ごとに分割しています。

結果のトレース

式の最終結果のみをトレースするには、TRACE 命令の TRACE R (TRACE Results) 形式を使用します。言語処理プログラムは、この命令の後に続くすべての式を以下のように分析します。

```
>>> Final result of an expression
```

上記の例で、TRACE 命令のオペランドを I から R に変更すると、以下の結果が表示されます。

```
9 *-* IF a + 1 > 5 * y
>>> "0"
```

演算と結果のトレースに加え、TRACE 命令では他のタイプのトレースも実行できます。[TRACE](#) を参照してください。

演習: TRACE 命令の使用

以下の複合式を使用したプログラムを作成します。

```
IF (a > z) | (c < 2 * d) THEN ....
```

プログラム内で a、z、c、および d を定義し、TRACE I 命令を使用します。

解答

```
/****** REXX *****/
/* This program uses the TRACE nstruction to show how the language */
/* processor evaluates an expression, operation by operation.      */
/****** */
a = 1
z = 2
c = 3
d = 4

TRACE I

IF (a > z) | (c < 2 * d) THEN
    SAY 'At least one expression was true.'
ELSE
    SAY 'Neither expression was true.'
```

図 43. 考えられる解決方法

このプログラムを実行すると、以下の結果が出力されます。

```
12 *-* IF (a > z) | (c < 2 * d)
    >V> "1"
    >V> "2"
    >O> "0"
    >V> "3"
    >L> "2"
    >V> "4"
    >O> "8"
    >O> "1"
    >O> "1"
    *-* THEN
13 *-* SAY 'At least one expression was true.'
    >L> "At least one expression was true."
At least one expression was true.
```

TRACE 命令によるコマンドのトレース

TRACE 命令には、トレースのさまざまなタイプに応じた多数のオプションがあります。例えば、コマンドをトレースするには C、エラーをトレースするには E を使用できます。

TRACE C

TRACE C の後、言語処理プログラムは実行前に各コマンドをトレースしてから、コマンドを実行し、そのコマンドからの戻りコードを現行の端末出力装置に送信します。現行の端末出力装置の指定について詳しくは、SET TERMOUT コマンドを参照してください。SET を参照してください。

TRACE E

TRACE E をプログラムで指定すると、言語処理プログラムは、実行後の戻りコードがゼロ以外の結果になったすべてのホスト・コマンドをトレースし、コマンドからの戻りコードを端末に送信します。

TRACE E が含まれるプログラムが誤ったコマンドを発行すると、そのプログラムからエラー・メッセージ、行番号、コマンド、コマンドからの戻りコードが出力ストリームに送信されます。

TRACE 命令について詳しくは、TRACE を参照してください。

REXX の特殊変数 RC および SIGL の使用

REXX 言語には、RC、SIGL、RESULT という 3 つの特殊変数があります。REXX/CICS は、特定の状況下でこれらの変数を設定し、式の中では常にこれらの変数を使用できます。

REXX/CICS が値を設定しなかった場合、REXX の他の変数と同様に、特殊変数にはその変数自体の名前が大文字で値として設定されます。特殊変数 RC および SIGL の 2 つを使用して、プログラム内の問題の診断に役立てることができます。

RC

RC は戻りコードを表します。言語処理プログラムは、プログラムがコマンドを発行するたびに RC を設定します。コマンドがエラーなしで終了すると、通常、RC は 0 に設定されます。コマンドがエラーで終了した場合、RC はそのエラーに割り当てられている戻りコードに設定されます。

RC 変数が特に役立つのは、IF 命令でプログラムがどのパスを選択すべきかを決定する場合です。

注：すべてのコマンドによって RC の値が設定されるため、プログラムの実行中、RC には同じ値が維持されません。RC を使用する際は、テストしたいコマンドの戻りコードが RC に含まれていることを確認してください。

SIGL

言語処理プログラムは、関数、SIGNAL 命令、または CALL 命令によるプログラム内での制御の移動に関連して、特殊変数 SIGL を設定します。言語処理プログラムは、制御をプログラムの別のルーチンまたは別の部分に移す際に、特殊変数 SIGL を制御の移動が発生した行番号に設定します。(以下の例に記載されている行番号は、例の後の説明を分かりやすくするためのものです。これらの行番号はプログラムには含まれていません)。

```
1 /* REXX */
2 :
3 CALL routine
4 :
5
6 routine:
7 SAY 'We came here from line' SIGL      /* SIGL is set to 3 */
8 RETURN
```

呼び出されたルーチン自体が別のルーチンを呼び出す場合、SIGL は、最新の移動が発生した行番号にリセットされます。

SIGL と SIGNAL ON ERROR 命令は、エラーの原因となったコマンドとエラーの内容を判断するのに役立ちます。SIGNAL ON ERROR がプログラムに含まれている場合、ゼロ以外の戻りコードを返すすべてのホス

ト・コマンドは、`error` という名前のルーチンに制御を移します。エラー・ルーチンは、エラー・メッセージの送信など、通常行われるアクションとは関係なく実行されます。

SIGNAL 命令について詳しくは、[SIGNAL](#) を参照してください。

対話式デバッグ機能を使用したトレース

対話式デバッグ機能は、ユーザーがプログラムの実行を制御できるようにします。言語処理プログラムは、端末から読み取り、出力を端末に書き込みます。

対話式デバッグの開始

対話式デバッグを開始するには、TRACE 命令のオプションの前に ? を指定します (例えば、TRACE ?A)。疑問符とオプションの間に空白があってはなりません。外部ルーチンが呼び出されると、対話式デバッグはその外部ルーチンには引き継がれません。トレース対象のプログラムに外部ルーチンが戻った時点で、対話式デバッグが再開されます。

対話式デバッグのオプション

対話式デバッグが開始した後は、一時停止するたびに、または言語処理プログラムが入力ストリームから読み取るたびに、以下のいずれかを入力できます。

- ヌル行。トレースが続行されます。言語処理プログラムは、次の一時停止または入力ストリームからの読み取りまで、実行を続行します。したがって、ヌル行の入力を繰り返すと、プログラムが終了するまで、一時停止ポイントから次の一時停止ポイントまでの単位で段階的にトレースが実行されることになります。
- 等号 (=)。最後にトレースされた命令が再実行されます。言語処理プログラムは前にトレースした命令を再実行します。再実行で使用される値は、入力ストリームから読み取られた命令によって変更されている可能性があります。(入力を代入命令にすることもできます。これにより、変数の値が変更されます)。
- 追加命令。コマンドや別のプログラムの呼び出しを含め、任意の REXX 命令を入力できます。この入力が処理されてから、プログラム内の次の命令がトレースされます。例えば、TRACE 命令を入力して、トレースのタイプを変更できます。

```
TRACE L /* Makes the language processor pause at labels only */
```

代入命令を入力することもできます。この場合、IF THEN ELSE 命令の特定の分岐が強制的に実行されるように変数の値を変更することで、プログラムのフローを変更できます。以下の例では、RC が前のコマンドによって設定されます。

```
IF RC = 0 THEN
DO
    instruction1
    instruction2
END
ELSE
    instructionA
```

コマンドがゼロ以外の戻りコードで終了すると、ELSE パスが選択されます。最初のパスを選択するよう強制する場合、対話式デバッグ中の入力は、以下のようになります。

```
RC = 0
```

対話式デバッグの終了

以下のいずれかの方法で、対話式デバッグを終了できます。

- TRACE OFF 命令を入力として使用します。対話式デバッグの開始時に出力される以下のメッセージに記載されているように、TRACE OFF 命令によってトレースが終了されます。

```
+++ Interactive trace. TRACE OFF to end debug, ENTER to continue. +++
```

- TRACE ? 命令を入力として使用します。

TRACE オプションの前に接頭部として疑問符を使用すると、対話式デバッグを開始できるだけでなく、終了することもできます。疑問符は、対話式デバッグの前の設定 (オンまたはオフ) を逆にします。したがって、プログラム内で TRACE ?R を使用して対話式デバッグを開始した後、オプションの前に ? を使用した別の TRACE 命令を入力すると、対話式デバッグが終了しますが、指定されたオプションでトレースが続行されます。

- オプションを指定しない TRACE を入力として使用します。入力ストリームでオプションなしの TRACE を指定すると、対話式デバッグがオフになりますが、TRACE Normal を適用したトレースが続行されます。(TRACE Normal では、実行後に失敗したコマンドのみをトレースします)。
- プログラムを最後まで実行させます。トレースを開始したプログラムが終了すると、対話式デバッグは自動的に終了します。プログラムを早期に終了させるには、入力として EXIT 命令を使用します。EXIT 命令により、プログラムと対話式デバッグの両方が終了します。

対話式 TRACE 出力の保管

REXX/CICS には、トレース出力をファイルに経路指定する機能があります。

REXX コマンド SET TERMOUT は、現在の端末装置の代わりに、またはその追加として、行モードの出力 (SAY 出力、TRACE 出力など) をファイルに経路指定します。[SET](#) を参照してください。

第 9 章 REXX/CICS ヘルプ・ユーティリティの使用

REXX/CICS に含まれているオンライン・ヘルプ・ユーティリティを使用して、REXX/CICS に付属している製品資料を検索して表示できます。

このタスクについて

このヘルプ・ユーティリティでは、目次、ヘルプ・トピック、および検索結果パネルが用意されています。

CICS Transaction Server for z/OS® 製品情報の最新リリースに付属している REXX for CICS Transaction Server 製品資料を参照して、オンライン製品資料より新しい更新情報が含まれていないか確認することをお勧めします。

手順

ヘルプへのアクセス

- 以下の方法で、ヘルプにアクセスできます。
 - CICS 環境から、REXX CICHELP と入力します。ヘルプの目次が表示されます。
 - REXX/CICS 対話環境から (前に CICS トランザクション REXX をパラメーターなしで入力しました)、HELP と入力します。ヘルプの目次が表示されます。
 - REXX/CICS テキスト・エディターから、HELP と入力するか、F1 (HELP) キーを押します。そのエディターに関するヘルプが表示されます。
 - ファイル・リスト・ユーティリティから、HELP と入力するか、F1 (HELP) キーを押します。そのユーティリティに関するヘルプが表示されます。

ヘルプ・トピックへのアクセス

- 以下の方法で、ヘルプ・トピックにアクセスできます。
 - REXX/CICS 対話環境から、HELP に続けてトピック・タイトルの検索ストリングを入力します (例: HELP TRANSLATE)。検索ストリング (大文字) がタイトルに含まれている最初のヘルプ・トピックが表示されます。

この方法は、特定の REXX コマンド、関数、またはキーワードのヘルプを検索するのに便利です。
 - 前述のとおりヘルプの目次にアクセスして、リストからトピックを選択します。選択するトピックの行にカーソルを置くか、またはそのトピックの行番号を入力フィールドに入力してから、Enter キーを押します。選択したトピックが表示されます。
 - 前述のとおりヘルプの目次にアクセスして、検索ストリングを入力します。その検索ストリングが含まれたすべてのトピックが検索結果パネルにリストされます。検索では大文字小文字の区別はありません。ほとんどの場合、検索結果パネルには、トピック名と親トピックの両方が表示されます。例えば、ADDRESS を検索した場合は、結果は次のように表示されます。

Keyword instructions > ADDRESS

リストからトピックを選択するには、選択するトピックの行にカーソルを置くか、またはそのトピックの行番号を入力フィールドに入力してから、Enter キーを押します。選択したトピックが表示されます。

検索結果パネルにリストされているすべてのトピックを表示するには、入力フィールドに ALL と入力します。

目次、ヘルプ・トピック、または検索結果パネルのスクロール

- 前方にスクロールして次の画面に進むには、F8 (FORWARD) キーを押します。

現在のヘルプ・トピックの末尾に達している場合は、次のメッセージが表示されます。

End of topic -- press F8 for next topic

目次または検索結果パネルの末尾に達している場合は、次のメッセージが表示されます。

Already at the bottom

- 画面の一部だけ前方にスクロールするには、ヘルプ画面上の任意の行にカーソルを置き、F8 (FORWARD) キーを押します。
カーソルを置いた行が、ヘルプ画面に表示される最初の行になります。
- 後方にスクロールして前の画面に戻るには、F7 (BACKWARD) キーを押します。
現在のヘルプ・トピックの先頭に達している場合は、次のメッセージが表示されます。

Start of topic -- press F7 for previous topic

目次または検索結果パネルの先頭に達している場合は、次のメッセージが表示されます。

Already at the top

- 画面の一部だけ後方にスクロールするには、ヘルプ画面上の任意の行にカーソルを置き、F7 (BACKWARD) キーを押します。
カーソルを置いた行が、ヘルプ画面に表示される最後の行になります。
- ヘルプ、ヘルプ・トピック、または検索結果パネルの終了
- ヘルプ、ヘルプ・トピック、または検索結果パネルを終了するには、F3 (END 機能) キーを押します。
これにより、直前に表示していた画面に戻ります。
- 例えば、目次が表示されているときに F3 キーを押すと、ヘルプにアクセスした方法に応じて、CICS 環境または REXX/CICS 環境に戻ります。トピックが表示されているときに F3 キーを押すと、そのヘルプ・トピックにアクセスした方法に応じて、検索結果パネルまたはヘルプ目次に戻ります。

第 10 章 プログラミング・スタイルおよび手法

プログラムを作成するために使用する方式は、プログラムを書くために使用する言語とちょうど同じくらい重要です。

データの検討

プログラムを書くタスクに取り組む場合、最初に検討することは処理に必要なデータです。入力データのリストを作成します。項目および各項目の可能な値は何ですか？項目に一種の構造かパターンがあれば、図に書いて見えるようにします。次に、同じことを出力データに関しても行います。2つの図を詳しく調べて、両者が適合するか確認します。適合すれば、プログラム設計の過程としては十分です。

次に、ユーザーが使用する仕様書を書きます。これは書かれた仕様書、ヘルプ・ファイルまたはその両方の場合があります。

すべてが終わった後にプログラムを書きます。

例を、次に示します。

ユーザーに「Heads or tails」を遊んでもらう対話式プログラムを書く必要があります。ユーザーは、好む限りゲームで遊ぶことができます。ユーザーは、ゲームを終了するために Quit を「Heads or tails?」の質問に対して応答する必要があります。プログラムは、コンピューターが必ず勝つように変更されています。

このプログラムをどのように書くか考えてみてください。

コンピューターは、以下の命令を使用して開始します。

```
Let's play a game! Type "Heads", "Tails",  
or "Quit"  
and press ENTER.
```

これは、4 種類の入力が可能であることを表します。

- HEADS
- TAILS
- QUIT
- 上記の 3 つ以外

これに対応する出力は以下になるはずです。

- Sorry. It was TAILS. Hard luck!
- Sorry. It was HEADS. Hard luck!
- That's not a valid answer. Try again!

また、このシーケンスは無限に繰り返される必要があり、CICS に戻って終了します。

今や、仕様書、入力データ、および出力データを理解して、プログラムを書く準備ができました。

データを検討せずにとにかく命令を書き始めた場合、余分に長く時間がかかるでしょう。

自己診断テスト

プログラムを書いてください。慎重であれば、あなたが最初に実行すべきでしょう。

```
/* CON EXEC */

/* Tossing a coin. The machine is lucky, not the user */

do forever
  say "Let's play a game! Type 'Heads', 'Tails'",
    "or 'Quit' and press ENTER."
  pull answer

  select
    when answer = "HEADS"
      then say "Sorry! It was TAILS. Hard luck!"
    when answer = "TAILS"
      then say "Sorry! It was HEADS. Hard luck!"
    when answer = "QUIT"
      then exit
    otherwise
      say "That's not a valid answer. Try again!"
  end
  say
end
```

お楽しみ

ここで、お楽しみの時間です。これはとても単純なアーケード・ゲームです。

これを入力して友人と遊んでください。後で、これを改良してもよいでしょう。

```

/* CATMOUSE */

/* The user says where the mouse is to go. But where */
/* will the cat jump? */

say "This is the mouse -----> @"
say "These are the cat's paws ---> ( )"
say "This is the mousehole -----> 0"
say "This is a wall -----> |"
say
say "You are the mouse. You win if you reach",
    "the mousehole. You cannot go past"
say "the cat. Wait for him to jump over you.",
    "If you bump into him you're caught!"
say
say "The cat always jumps towards you, but he's not",
    "very good at judging distances."
say "If either player hits the wall he misses a turn."
say
say "Enter a number between 0 and 2 to say how far to",
    "the right you want to run."
say "Be careful, if you enter a number greater than 2 then",
    "the mouse will freeze and the cat will move!"
say

```

```

/*-----*/
/* Parameters that can be changed to make a different */
/* game */
/*-----*/
len = 14          /* length of corridor */
hole = 14         /* position of hole */
spring = 5       /* maximum distance cat can jump */
mouse = 1        /* mouse starts on left */
cat = len        /* cat starts on right */
/*-----*/
/* Main program */
/*-----*/
do forever
  call display
  /*-----*/
  /* Mouse's turn */
  /*-----*/
  pull move
  if datatype(move,whole) & move >= 0 & move <= 2
  then select
    when mouse + move > len then nop /* hits wall */
    when cat > mouse,
      & mouse + move >= cat          /* hits cat */
    then mouse = cat
    otherwise                        /* moves */
      mouse = mouse + move
  end
  if mouse = hole then leave          /* reaches hole */
  if mouse = cat then leave          /* hits cat */
  /*-----*/
  /* Cat's turn */
  /*-----*/
  jump = random(1,spring)
  if cat > mouse then do              /* cat tries to jump left */
    Temp = cat - jump
    if Temp < 1 then nop              /* hits wall */
    else cat = Temp
  end
  else do                             /* cat tries to jump right */
    if cat + jump > len then nop      /* hits wall */
    else cat = cat + jump
  end
  if cat = mouse then leave
end
/*-----*/
/* Conclusion */
/*-----*/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit

```

```

/*-----*/
/* Subroutine to display the state of play */
/* */
/* Input: CAT and MOUSE */
/* */
/* Design note: each position in the corridor occupies */
/* three character positions on the screen. */
/*-----*/
display:
corridor = copies(" ",3*len)        /* corridor */
corridor = overlay("0",corridor,3*hole-1) /* hole */

if mouse /= len                      /* mouse in hole? */
then corridor = overlay("@",corridor,3*mouse-1)/* mouse */

corridor = overlay("(",corridor,3*cat-2) /* cat */
corridor = overlay(")",corridor,3*cat)
say " |"corridor|"
return

```

☒ 44. CATMOUSE

よくできました。次は、しばらく時間をかけて新しいスキルを実践するか、この先を読み進めてください。

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料](#)で使用されている表記規則および用語も参照してください。

プログラムの設計

方式(言語と同じくらい重要です)についてまだ考え中であれば、CATMOUSE をもう一度見てみましょう。

プログラムは、ネコとネズミ、および回廊内の彼らの位置についてです。いくつかの段階では、彼らの位置は画面に表示される必要があります。一度にすべてを考えるのは複雑すぎるため、最初のステップは以下のように細分化することです。

- **メインプログラム:** 彼らの位置を計算します。
- **表示サブルーチン:** 彼らの位置を表示します。

メインプログラムを見てみましょう。ユーザー(マウスを操作する)は、移動する前にネコとネズミがいる場所を知りたいでしょう。ネコは違います。次のステップは、メインプログラムを以下のようにさらに細分化することです。

```
Do forever
  call Display
  Mouse's move
  Cat's move
end
Conclusion
```

ループの設計方式

ループを設計する方法は、必ず終了するか、また、終了時にはデータが必要な条件を満たしているかという2つの質問をすることです。

CATMOUSE の例では、ループは以下の場合に終了(そしてゲームも終了)します。

1. ネズミが穴に逃げ込む。
2. ネズミがネコにぶつかる。
3. ネコがネズミを捕まえる。

結果

プログラム終了時に、ユーザーは結果を知る必要があります。

```
call display
say who won
```

これまでの成果

これをすべて一緒にすると、以下のようになります。


```

/*-----*/
/* Main program                                */
/*-----*/
do forever
  call display
  /*-----*/
  /* Mouse's turn                              */
  /*-----*/
  ...

  if mouse = hole then leave          /* reaches hole */
  if mouse = cat then leave           /* hits cat    */
  /*-----*/
  /* Cat's turn */
  /*-----*/
  ...

  if cat = mouse then leave
end

/*-----*/
/* Conclusion                                  */
/*-----*/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit

/*-----*/
/* Subroutine to display the state of play      */
/*-----*/
/* Input: CAT and MOUSE                        */
/*-----*/
display:
...

```

説明したばかりの方式は、段階的洗練 と呼ばれることがあります。まず仕様書で始めました (それは不完全の可能性あります)。次に、計画されたプログラムをルーチンに分割し、プログラム全体をコーディングするよりもそれぞれのルーチンをコーディングする方が容易になるようにします。それから、これらのルーチンのそれぞれに関して、最初としては正確にコーディングできたことが確信できるルーチンになるまで、処理を繰り返します。

これを行う間、常に以下の2つの問いを自問します。

- このルーチンが処理するデータは何か。
- 仕様書は完全か。

段階的詳細化: 例

ジョブをより単純なジョブに細分化することを段階的詳細化 と呼びます。

おばあさんは、あなたがセーリングに行く際に着られるよう、暖かいウールの上着を編もうとしています。これは、おばあさんが行いそうなことです。

1. 前面を編む
2. 背面を編む
3. 左腕を編む
4. 右腕を編む
5. 部品を一緒に縫い付ける

これらのジョブのそれぞれは、プルオーバーを編むジョブより説明するのが容易です。コンピューターの専門用語では、ジョブを単純なジョブに細分化することを段階的詳細化 と呼びます。

この段階で、再度仕様書を調べます。船乗りは、プルオーバーを、暗い中で迅速に前後ろを気にせずに着る必要があることがあります。したがって、前面は背面と同じ、両腕も同じにする必要があります。これは以下のようにプログラムにすることができます。

```

do 2
  CALL Knit_body_panel
end

```

```
do 2
  CALL Knit_sleeve
end
CALL sew_pieces_together
```

プログラミングで最善の方式は、上位から始めて、コーディングしやすいレベルになるまでプログラムを洗練させ続けることです。

トップダウンは最善の手法です。

データの再検討

プログラムを洗練させる場合、目的はそれぞれの部分をさらに単純にすることです。

これはほぼ確実に、以下のことを表します。

- 各セグメントか各ルーチンのためのさらに単純な入力データ
- 各セグメントか各ルーチンのためのさらに単純な出力データ
- さらに単純な処理
- したがって、さらに単純なコード

部分が本当に単純であれば、おそらく名前も単純になります。例えば、以下のとおりです。

- Knit cuff

以下のようにはなりません。

- Make ribbing for cuffs and waistband

プログラムの修正

さまざまな手法を使用して、プログラムを修正できます。

プログラムが誤った結果を出力する理由を理解できない場合、以下が可能です。

- プログラムの実行内容が示されるように、プログラムを変更できます。
- REXX 対話式トレース機能を使用できます。[プログラムの対話式デバッグ](#)を参照してください。

どちらの手法を使用する方が好ましいかを評価できます。

プログラムの変更

プログラムを変更して、追加の命令を入れ、何が起きているのかを示すことができます。

以下のように、追加の命令をプログラムに入れることができます。

```
...
say "Checkpoint A. x =" x
...
say "End of first routine"
...
```

プログラムのトレース

REXX 対話式トレース機能を使用して、プログラムで何が起きているかを理解することができます。

TRACE 命令の詳細については、[TRACE](#) を参照してください。

- プログラムがどこに向かっているかを調べるには、TRACE Labels を使用します。例は、プログラムおよびプログラムが画面に表示するトレースを示しています。

```

/* ROTATE */

/* Example: two iterations of wheel, six iterations */
/* of cog. On the first three iterations, "x < 2" */
/* is true. On the next three, it is false.      */
trace L
do x = 1 to 2
wheel:
  do 3
cog:
  if x < 2 then do
true:
    end
  else do
false:
    end
  end
end
done:

```

図 45. ROTATE

これにより、以下のトレースが行われます。

```

rotate
  6 ** wheel:
  8 ** cog:
 10 ** true:
  8 ** cog:
 10 ** true:
  8 ** cog:
 10 ** true:
  6 ** wheel:
  8 ** cog:
 13 ** false:
  8 ** cog:
 13 ** false:
  8 ** cog:
 13 ** false:
 17 ** done:

```

- 言語処理プログラムが式を計算する方法を確認するには、TRACE Intermediates を使用します。
- 正しいデータをコマンドまたはサブルーチンに渡しているかどうかを調べるには、TRACE Results を使用します。
- コマンドからのゼロ以外の戻りコードの確認を確実に開始するには、TRACE Errors を使用します。

コーディング・スタイル

プログラムを読んで内容が正しいかどうかを判別できるようにするために、プログラムは読みやすくする必要があります。

「読みやすい」という意味は、プログラマーによりさまざまです。スタイルの異なる例を以下に示します。好ましいスタイルを選択できます。プログラムを確認する良い方法は、同僚にそのプログラムを読むように依頼することです。同僚が読みやすいと判断するコーディング・スタイルを使用します。

104 ページの『お楽しみ』サンプル・プログラムからの以下のプログラム・フラグメントは、ほとんどの人にとって読みにくいものです。

```

/*****
/* SAMPLE #1: A portion of CATMOUSE
/* not divided into segments and written with no
/* indentation, and no comments. This style is not
/* recommended.
*****/

do forever
call display
pull move
if datatype(move,whole) & move >= 0 & move <=2
then select
when mouse+move > len then nop
when cat > mouse,
& mouse+move >= cat,
then mouse = cat
otherwise
mouse = mouse + move
end
if mouse = hole then leave
if mouse = cat then leave
jump = random(1, spring)
if cat > mouse then do
if cat-jump < 1 then nop
else cat = cat-jump
end
else do
if cat+jump > len then nop
else cat = cat+jump
end
if cat = mouse then leave
end
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit

```

この次の例は、前のに比べると読みやすいです。例はセグメントに分割されて、それぞれには固有の見出しが付いています。右側にあるコメントは、注釈と呼ばれることがあります。これは、読む人が内容の概要を知るのに役立ちます。

```

/*****
/* SAMPLE #2: A portion of CATMOUSE
/* divided into segments and written with 'some'
/* indentation and 'some' comments.
*****/

/*****
/* Main program
*****/
do forever
  call display
  /*****
  /* Mouse's turn
  *****/
  pull move
  if datatype(move,whole) & move >= 0 & move <=2
  then select
    when mouse+move > len then nop      /* hits wall */
    when cat > mouse,
      & mouse + move >= cat,          /* hits cat */
    then mouse = cat
    otherwise                          /* moves */
      mouse = mouse + move
  end
  if mouse = hole then leave          /* reaches hole */
  if mouse = cat then leave          /* hits cat */
  /*****
  /* Cat's turn */
  *****/
  jump = random(1,spring)
  if cat > mouse then do              /* cat tries to jump left */
    if cat - jump < 1 then nop        /* hits wall */
    else cat = cat - jump
  end
  else do                            /* cat tries to jump right */
    if cat + jump > len then nop      /* hits wall */
    else cat = cat + jump
  end
  if cat = mouse then leave
end
/*****
/* Conclusion
*****/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit

```

次のこの例には、一部のプログラマーに人気の機能が追加されています。大文字で書かれたキーワードおよび異なる字下げスタイルは、コードの構造を強調します。また、豊富なコメントは仕様書の詳細を思い出させてくれます。

```

/*****
/* SAMPLE #3: A portion of CATMOUSE
/* divided into segments and written with 'more'
/* indentation and 'more' comments.
/* Note commands in uppercase (to highlight logic)
*****/

/*****
/* Main program
*****/
DO FOREVER
  CALL display
  /*****
  /* Mouse's turn
  *****/
  PULL move
  IF datatype(move,whole) & move >= 0 & move <=2
    THEN SELECT
      WHEN mouse+move > len /* mouse hits wall */
        THEN nop /* and loses turn */
      WHEN cat > mouse,
        & mouse+move >= cat, /* mouse hits cat */
        THEN mouse = cat /* and loses game */
      OTHERWISE mouse = mouse + move /* mouse ... */
    END /* moves to new location */
  IF mouse = hole THEN LEAVE /* mouse is home safely */
  IF mouse = cat THEN LEAVE /* mouse hits cat (ouch) */
  /*****
  /* Cat's turn
  *****/
  jump = RANDOM(1,spring) /* determine cat's move */
  IF cat > mouse /* cat must jump left */
    THEN DO
      IF cat-jump < 1 /* cat hits wall */
        THEN nop /* misses turn */
      ELSE cat = cat-jump /* cat jumps left */
    END
  ELSE DO /* cat must jump right */
    IF cat+jump > len /* cat hits wall */
      THEN nop /* misses turn */
    ELSE cat = cat+jump /* cat jumps right */
  END
  IF cat = mouse THEN LEAVE /* cat catches mouse */
END
/*****
/* Conclusion
*****/
CALL display /* on final display */
IF cat = mouse /* who won? */
  THEN say "Cat wins" /* ... the cat */
  ELSE say "Mouse wins" /* ... the mouse */
EXIT

```

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料](#)で使用されている表記規則および用語も参照してください。

第 2 部 REXX の構成

このセクションでは、REXX を構成して管理する方法の概要を示します。

第 11 章 REXX サポートの構成

REXX プログラムを実行するには、その前に REXX サポートを構成する必要があります。

以下の手順を実行します。

1. RFS ファイル・プールを作成します。
2. リソース定義を作成します。
3. LSRPOOL 定義を確認します。
4. CICSTART メンバーを更新します。
5. CICS 初期設定 JCL を変更します。
6. インストールを検証します。
7. RFS ファイル・プールを形式設定します。
8. ヘルプ・ファイルを作成します。
9. REXX Db2 インターフェースを構成します。

RFS ファイル・プールの作成

REXX ファイリング・システム (RFS) では、データを保管するために複数のファイル・プールが使用されます。

これらは VSAM クラスターのセットとして実装されます。CICSTS56.REXX.SCICJCL 内で提供されているメンバー CICVSAM を使って、2 つの RFS ファイル・プールのための VSAM データ・セットを作成します。必要に応じて、お客様独自の環境に合うようにこのジョブを変更します。例えば、ご使用のインストール済み環境の標準指針に合わせてデータ・セット名を変更することができます。変更が完了したら、ジョブを実行してファイル・プールを定義します。

リソース定義でファイル・プール名を変更したりファイル・プール・コンポーネントを追加したりする場合は、CICSTART メンバー内のファイル・プール定義に、対応する変更を加える必要があります。[115 ページの『リソース定義の作成』](#)および[116 ページの『CICSTART メンバーの更新』](#)を参照してください。

以下の定義が整合していることを確認してください。

- CICSTART 内のファイル・プール定義
- CICRDOD (Development System の場合) または CICRDOR (Runtime Facility の場合) の中のリソース定義
- CICVSAM 内の VSAM クラスター定義

リソース定義の作成

REXX のプロファイル、プログラム、トランザクション、およびファイル・リソースの定義は、提供された CSD またはアップグレードされた CSD 内のグループ CICREXX に含まれています。

CICSTS56.REXX.SCICJCL データ・セット内の CICRDOR ジョブ (Runtime Facility の場合) または CICRDOD ジョブ (Development System の場合) により、REXX/CICS プロファイル、VSAM ファイル、プログラム、トランザクション、および一時データ・キューを含む、製品で必要とされるエントリーを追加します。

一時データ・キューは、REXX/CICS IMPORT および EXPORT コマンドで使用されます。このジョブには、Db2 プランに対してトランザクションを許可する、REXX/CICS SQL インターフェースのための定義も含まれています。

JCL を編集し (JCL の最初のコメント内で説明されているように、エントリーのコメント解除してください)、ジョブを実行します。

IMPORT および EXPORT コマンドの TD キューの変更

REXX/CICS 環境では、区分データ・セットからメンバーを IMPORT する際、または区分データ・セットに RFS ファイルを EXPORT する際に、動的割り振りを使用します。

CICSTS56.REXX.SCICJCL データ・セット内のメンバー CICRDOD または CICRDOR を使って、IMPORT の入力として使用する一時データのエントリーを3つと、EXPORT の出力として使用する一時データのエントリーを3つ定義します。これにより、区分データ・セットに対して、3 ユーザーによる同時の IMPORT と、3 ユーザーによる同時の EXPORT を可能にします。

ユーザーの要件に合うように TDQ エントリーの数を変更します。しかし、入力用エントリーと出力用エントリーが少なくとも1つずつは必要です。TDQUEUE NAME は REX で始まらなければならず、その接尾部は有効な文字でなければなりません。REX で始まる TDQUEUE 名を使用する他のアプリケーションを使用しないでください。これは、IMPORT および EXPORT がこの名前を使用するため、ファイルが破壊されてしまう場合があるためです。

LSRPOOL 定義の確認

RFS ファイルの最大キー長は 252 であり、提供される VSAM 定義では制御インターバル・サイズとして 18K が使用されるため、RFS に使用される LSRPOOL がこれらの値をサポートしていることを確認することが重要です。

これを実行しないと、システム・コンソールに OPEN エラーが表示される可能性があります。詳しくは、[LSRPOOL リソースおよびローカル共用リソース \(LSR\) または非共用リソース \(NSR\)](#)を参照してください。

CICSTART メンバーの更新

hlq.CICSTS56.REXX.SCICEXEC データ・セット内のメンバー CICSTART には、REXX/CICS 環境のデフォルト定義が含まれています。CICSTART は、CICS システムが始動した後、CICS/REXX プログラムを使用する最初のトランザクションが出される時に実行されます。

CICSTART は、環境の永続的な構成を保持します。提供されている CICSTART メンバーを編集するか、またはそのメンバーを独自の CICSTART exec のモデルとして使用してください。

- 初期設定

疑似会話型モードをオフに設定する (PSEUDO OFF) 初期設定ステートメントと、RLS ディレクトリーを検査する (RLS CKDIR) 初期設定ステートメントを、提供されている CICSTART メンバーの開始時に示されるとおりに使用します。

- RFS ファイル・プールを定義する

提供されている CICSTART メンバーは、2つのファイル・プールを定義します。

```
'FILEPOOL DEFINE POOL1 RFSDIR1 RFSP00L1 (USER'  
  IF RC ^= 0 THEN EXIT RC  
'FILEPOOL DEFINE POOL2 RFSDIR2 RFSP00L2 (USER'  
  IF RC ^= 0 THEN EXIT RC
```

[115 ページの『RFS ファイル・プールの作成』](#)のステップでファイル・プール名を変更したり、ファイル・プールを追加したりした場合は、CICSTART exec 内のファイル・プール定義に、対応する変更を加える必要があります。

- ユーザー ID に権限を付与する

提供されている CICSTART メンバーは、ユーザー ID RCUSER に権限を付与します。その権限を必要とする組織内のユーザーに権限を付与するコマンドが CICSTART exec に含まれていることを確認してください。

ファイル・プールをフォーマット設定するための権限を必要とするすべてのユーザーを含めてください。そうしないと、FILEPOOL FORMAT コマンドが戻りコード -4 で失敗します。

CICS セキュリティーがオフになっている領域 (つまり、SIT パラメーター SEC=NO) については、CICS デフォルト・ユーザー ID のために AUTHUSER コマンドを CICSTART に組み込みます。例えば、領域のデフォルト・ユーザー ID が SYSA である場合、次のコマンドを組み込みます。

```
'AUTHUSER SYSA'
```

- 必要に応じて CICS トランザクション ID を REXX exec に関連付ける

CICSTART には、CICS DEFINE TRANSACTION コマンドによって定義された CICS トランザクション ID を呼び出し対象の REXX exec に関連付けるために、DEFTRNID ステートメントを含める必要があります。提供されている CICSTART メンバーは、REXX、EDIT、および FLST の各トランザクションをそれぞれ、CICRXTRY exec、CICEDIT exec、および CICFLST exec に関連付けます。これらの CICS トランザクションは、CICRDOR および CICRDOD の JCL ファイルで定義されています。[115 ページの『リソース定義の作成』](#)を参照してください。

ご使用の CICSTART exec には、REXX/CICS トランザクションを実行するように定義するすべての CICS トランザクションのために、DEFTRNID ステートメントを含める必要があります。

- REXX exec を実行するためのデフォルトを設定する

ご使用の環境に必要な SETSYS コマンドを組み込みます。例えば、インストール済み環境で REXX/CICS exec を実行するための言語、検索 PF キー、および疑似会話型設定を設定できます。

デフォルトでは、疑似会話型モードはオンです (ただし、CICSTART exec 自体は会話型モードで実行する必要があります)。疑似会話型モードについて詳しくは、[PSEUDO](#) および [ユーザーに表示するダイアログの処理](#)を参照してください。

- ヘルプ・パスを定義する

提供されている CICSTART メンバーでは、現行のヘルプ・ユーティリティのヘルプ・パス HELPPTH2 と、互換性のために以前のヘルプ機能のヘルプ・パス HELPPTH が定義されています。

```
HELPPATH = 'POOL2:\BOOK'  
'RLS VARPUT HELPPATH \SYSTEM\DEFAULTS'  
IF RC = 0 THEN EXIT  
HELPPTH2 = 'POOL2:\HELP'  
'RLS VARPUT HELPPTH2 \SYSTEM\DEFAULTS'  
IF RC = 0 THEN EXIT RC
```

以前のヘルプ機能がインストールされていない新しいインストール済み環境の場合、HELPPTH を定義する 3 つのステートメントを削除できます。

- 必要に応じて exec をプリロードするための EXECLOAD を使用する

exec をプリロードするための EXECLOAD ステートメントを組み込むことができます。exec をプリロードすると、ユーザーごとに exec をロードする必要がないためパフォーマンスが向上します。また、すべての同時ユーザーが同じコピーを共用できるため、必要なストレージが少なくなります。

提供されている CICSTART メンバーで EXECLOAD コマンドのリストを使用するか、必要な exec のために EXECLOAD コマンドを組み込むことができます。

提供されている CICSTART メンバーには、CICEDIT exec、CICESVR exec、および CICEPROF exec 用の EXECLOAD コマンドが組み込まれています。これらの exec は、REXX/CICS テキスト・エディターのコンポーネントです。このエディターは REXX Development System のみで提供され、独自の REXX プログラムを開発するときに使用することができます。ご使用のインストール済み環境で REXX/CICS テキスト・エディターを使用している場合、これらの EXECLOAD コマンドを含めることができます。

詳しくは、[EXECLOAD](#) を参照してください。

CICS 初期設定 JCL の変更

DD ステートメントを CICS 始動ジョブおよび DFHRPL 連結に追加します。

CICS 始動ジョブに、次の DD ステートメントを追加します。

```
//CICAUTH DD DSN=CICSTS56.REXX.SCICMDS,DISP=SHR  
//CICEXEC DD DSN=CICSTS56.REXX.SCICEXEC,DISP=SHR  
//CICUSER DD DSN=CICSTS56.REXX.SCICUSER,DISP=SHR
```

REXX データ・セットの DD ステートメントを DFHRPL 連結に追加します。

```
//DFHRPL DD DSN=CICSTS56.REXX.SCICLOAD,DISP=SHR
```

REXX/CICS 環境では、CICS でリソース定義を持たない、CICCMDS、CICEXEC、および CICUSER という 3 つのデータ・セット連結の DD 名を使用します。これらのデータ・セットは区分データ・セットで、MVS 機能を使用してアクセスします。

CICCMDS

CICCMDS の DD 名連結は、CICSTS56.REXX.SCICCMDS データ・セットの参照で始まります。このデータ・セットには、REXX/CICS 環境の許可コマンドをインプリメントする exec が含まれています。許可ユーザー、または許可コマンドの使用を許可された exec のみが、これらの exec にアクセスできます。独自の許可コマンドを使用して REXX/CICS 環境を拡張する場合は、データ・セットをこの DD 名連結に連結します。

CICEXEC

CICEXEC の DD 名連結は、CICSTS56.REXX.SCICEXEC データ・セットの参照で始まります。このデータ・セットには、REXX/CICS 環境によって提供されている、許可コマンドを使用する exec が含まれています。許可コマンドを使用する独自の exec を作成して REXX/CICS 環境を拡張する場合は、その exec が入っているデータ・セットをこの DD 名に連結します。

CICUSER

CICUSER の DD 名連結は、CICSTS56.REXX.SCICUSER データ・セットの参照で始まります。このデータ・セットには、REXX/CICS 環境によって提供されている、許可コマンドを使用しない exec が含まれています。許可コマンドを使用しない独自の exec を作成して REXX/CICS 環境を拡張する場合は、その exec が入っているデータ・セットをこの DD 名に連結します。

これらのデータ・セット連結にアクセスするために使用する機能では、CICS 領域が待ち状態になるのを避けるために CICS WAIT EXTERNAL の機能を使用します。

RFS ファイル・プールの形式設定

CICSTART で定義されているすべての REXX ファイリング・システム (RFS) ファイル・プールの形式設定をする必要があります。

1. 必要なすべての構成タスクが完了していることを確認し、必要に応じて CICS を再始動します。
2. CICSTART で許可ユーザーとして定義されたユーザー ID を使用してサインオンします。
3. REXX と入力します (これは、CICRXTY exec に関連したデフォルト・トランザクション ID です)。

このアクションによって、REXX および REXX/CICS コマンドを対話式に実行できる対話環境を提供する REXXTY ユーティリティが呼び出されます。画面上部に以下の行が表示され、右下隅には READ が表示されます。カーソルは左下隅にあります。

Enter a REXX command or EXIT to quit

4. CICSTART で定義されたすべてのファイル・プールに以下のコマンドを入力して、ファイル・プールを使用できるように準備します。

'FILEPOOL FORMAT poolname'

poolname は、CICSTART exec で指定したファイル・プール名です (デフォルトは POOL1 および POOL2)。

REXX コマンドを区切るために単一引用符または二重引用符を使用することをお勧めします。

対話環境では、画面上で次に使用可能な行に各コマンドがエコー出力され、要求された出力も表示されます。

FILEPOOL FORMAT コマンドは、正常に完了したことを示す情報を表示しませんが、画面の右下隅に READ の語が表示されます。

5. **FILEPOOL FORMAT** コマンドが正常に終了したかどうかを調べるには、SAY RC (アポストロフィなし) を入力します。使用可能な次の行に 0 が表示されると、コマンドが成功したことを示しています。

ファイル・プールを再形式設定しようとする、以下のメッセージが表示されます。

Subcommand return code = 1836

- すべての RFS ファイル・プールが形式設定されるまで、この処理を続けます。ファイル・プールを形式設定するのは、新しいファイル・プールを定義した場合、または既存のファイル・プールのクラスターを削除して再定義した場合のみです。

ファイル・プールを形式設定する際に、または REXX や REXX/CICS のコマンドや命令を対話式に実行している間に、画面がいっぱいになったら、右下隅に MORE 標識が表示されます。画面をクリアするには、Enter キーを押します。いつでも Clear キーを押してデータの画面をクリアできます。対話環境を終了するには、PF3 キーを押します。このアクションは EXIT 命令の入力をシミュレートします。ユーザー自身で EXIT 命令を (アポストロフィなしで) 入力することもできます。

対話環境で RETRIEVE キーを押すと、以前に入力したコマンドが再呼び出しされます。このキーのデフォルトのシステム設定は PF12 です。RETRIEVE キーを押すと、直前に入力した行が入力位置に再表示されます。その後、この領域を変更し、Enter を押して命令を再実行できます。RETRIEVE キーを複数回押すと、前に入力したコマンドが新しい順に入力域に表示されます。

インストールの検証

インストールが成功したことを検証するために、exec が提供されています。

- REXX/CICS REXXTRY ユーティリティで、CALL CICIVP1 を入力して、インストール検査プログラムを実行します。exec からの出力は以下のようになります (画面に MORE と表示されたら、必ず Enter キーを押してください)。

```
EXEC CICIVP1
***-----***
*** This is a test REXX program running under CICS-TS ***
*** It was loaded from CICUSER-CICIVP1 ***
***-----***

What is your name?
```

- 名前を入力し、Enter キーを押します。出力は以下のようになります。

```
Welcome to REXX/CICS for CICS-TS , xxxx

Invoking nested exec CICIVP2 (which has tracing on)
  20 ** say 'You entered CICIVP2 exec'
    >>> "You entered CICIVP2 exec"
You entered CICIVP2 exec
  21 ** call CICIVP3
You entered CICIVP3 exec which has tracing off
  22 ** exit
Back to CICIVP1 exec

      This is fullscreen output to terminal xxxx
      Now input some data and press ENTER or a PF key

The AID key that was pressed = ENTER
The cursor was at (Row Col): 24 7
The data that was entered (Row Col Data): 24 1 <DATA>
Example of more than one screen
1000 assignment statements have been executed
2000 assignment statements have been executed
3000 assignment statements have been executed
4000 assignment statements have been executed
5000 assignment statements have been executed
6000 assignment statements have been executed
7000 assignment statements have been executed
8000 assignment statements have been executed
9000 assignment statements have been executed
10000 assignment statements have been executed
11000 assignment statements have been executed
12000 assignment statements have been executed
13000 assignment statements have been executed
14000 assignment statements have been executed
15000 assignment statements have been executed
16000 assignment statements have been executed
17000 assignment statements have been executed
18000 assignment statements have been executed
19000 assignment statements have been executed
```

```
20000 assignment statements have been executed
Today's date is dd mmm yyyy
The time is hh:mm:ss
REXX/CICS CICIVP1 is now finished
```

ヘルプ・ファイルの作成

REXX/CICS に含まれているオンライン・ヘルプ・ユーティリティを使用して、REXX/CICS に付属している製品資料を検索して表示できます。

このタスクについて

インストールの一部として、CICS TS 製品のデリバリー・データ・セットから REXX/CICS ファイル・システムに情報をロードする必要があります。

手順

1. REXX ファイリング・システム (RFS) のファイル・プールを定義し、ヘルプ・ユーティリティのヘルプ・パスを設定します。

hlq.CICSTS56.REXX.SCICEXEC データ・セットで提供される *CICSTART* メンバーには、RFS ファイル・プールを定義するステートメントと、REXX ヘルプのヘルプ・パスを設定するためのステートメントが含まれています。それらのステートメントを確認し、必要に応じてお客様の要件に合うように変更してください。詳しくは、[116 ページの『CICSTART メンバーの更新』](#)を参照してください。

提供されている *CICSTART* メンバーでは、現行のヘルプ・ユーティリティのヘルプ・パス *HELPPTH2* が定義されています。互換性のために、以前のヘルプ機能のヘルプ・パス *HELPPTH* も定義されています。以前のヘルプ機能がインストールされており、対象となるインストール手順が完了するまで使用する必要がある場合を除き、*HELPPTH* は必要ありません。

2. ステップ [120 ページの『3』](#) でヘルプ情報をロードするために使用するユーザー ID に、関係するデータ・セットを CICS TS 製品デリバリー・ライブラリーから REXX ファイリング・システムにインポートする権限が付与されていることを確認します。

hlq.CICSTS56.REXX.SCICEXEC データ・セット内の *CICHLOAD exec* は、提供されている *hlq.CICSTS56.REXX.SCICDOC* データ・セットと *hlq.CICSTS56.REXX.SCICPNL* データ・セットからヘルプ情報をインポートします。その情報を使用して、REXX ヘルプで使用される RFS にファイルが作成されます。

この権限を確認する 1 つの方法は、関係するユーザー ID を高位修飾子として使用して、*hlq.CICSTS56.REXX.SCICDOC* および *hlq.CICSTS56.REXX.SCICPNL* のコピーを作成することができます。

3. REXX/CICS を開始し、EXEC *CICHLOAD* コマンドを発行します。

CICHLOAD は、MVS ライブラリーからヘルプ情報とパネル定義をインポートします。以下のメッセージが表示されたら、前のステップで設定したデータ・セット名を指定します。

```
Please enter the MVS dataset name of the help package library
Example: hlq.CICSTS55.REXX.SCICDOC
```

```
Please enter the MVS dataset name of the panels library
Example: hlq.CICSTS55.REXX.SCICPNL
```

提供されている *CICSTS56.REXX.SCICDOC* データ・セットには、オンライン・ヘルプ情報を入れた *CICRXHLP* メンバーが入っています。*CICHLOAD* プログラムはこれを処理して、オンライン・ヘルプ・ユーティリティで使用されるファイルを作成します。ファイルは、指定したヘルプ・パス *HELPPTH2* に作成されます。

このデータ・セットにある *CICR3270* または *CICR3820* メンバーは、*CICRXHLP* メンバーと、[CICS Transaction Server for z/OS 製品情報](#)に付属の PDF に置き換えられます。

CICS Transaction Server for z/OS 製品情報の最新リリースに付属している REXX for CICS Transaction Server 製品資料を参照して、オンライン製品資料より新しい更新情報が含まれていないか確認することをお勧めします。

REXX Db2 インターフェースの構成

このステップが必要となるのは、REXX EXECSQL コマンド環境を Db2 サポートのために有効にしている場合だけです。このステップを実行するには、その前に Db2 を完全にインストールしておく必要があります。

CICSTS56.REXX.SCICJCL データ・セットのメンバー CICRDOD または CICRDOR は、REXX トランザクションでの Db2 プランの使用を許可します。

提供されたトランザクションを REXX/CICS 環境用に変更する場合、または Db2 インターフェース・コードを使用する新規トランザクションを実装する場合は、Db2 エントリー定義も変更または追加する必要があります。

Db2 プランへの CICSQL プログラムの BIND

CICSTS56.REXX.SCICJCL データ・セット内の CICBIND ジョブを使って、CICSQL を正しい Db2 パッケージにバインドします。ジョブを編集し、実行してください。

使用している Db2 のレベルによっては、このジョブで条件コード 4 を受け取る場合があります。

第 12 章 REXX/CICS システムの定義と管理

REXX/CICS のシステム定義、カスタマイズ、管理について説明します。

REXX/CICS 許可コマンドと許可コマンド・オプション

いくつかの REXX/CICS コマンドやコマンド・オプションは、許可されたものと見なされています。

[REXX/CICS コマンド](#)を参照してください。REXX/CICS 許可コマンドは、以下の場合にのみ実行できます。

- exec を発行するユーザー ID が REXX/CICS 許可ユーザーである場合。許可ユーザーは AUTHUSER コマンドで定義されます。
- exec が REXX/CICS 許可サブライブラリーからロードされた場合。許可ライブラリーは、(CICS 始動プロシージャー/JCL で) DD 名 CICAUTH または CICEXEC に割り振られた MVS 区分データ・セットです。

これらの規則は、コマンドを試行する exec が許可 exec によって発行されたかどうかに関係なく、適用されます。

システム・プロファイル exec

CICS システム再始動の後、最初のユーザー exec が実行される前に、CICSTART という名前のシステム・プロファイル exec が発行されます。

通常、システム・プロファイル exec には、DD 名 CICAUTH または CICEXEC に割り振られる MVS PDS REXX 許可ライブラリー内に存在する必要があるシステム・カスタマイズ・コマンド、許可サブライブラリー定義、許可ユーザー定義、および許可コマンド定義が含まれています。

CICS システムの再始動後にユーザーが初めて REXX/CICS に入ったときに、CICSPROF という名前のシステム・ユーザー・プロファイル exec が発行されます。この exec には、すべてのユーザーが実行する必要があるセットアップ手順が含まれています。CICSPROF は、ユーザー・プロファイルも呼び出します。

ユーザー・プロファイルはユーザーが作成し、保守する exec です。これによってユーザーは REXX/CICS 環境をカスタマイズできます (パスの設定、RETRIEVE キーの変更、他の exec の呼び出しなど)。このプロファイルはユーザー個人の RFS ディレクトリーに配置する必要があります。

関連資料

[116 ページの『CICSTART メンバーの更新』](#)

hlq.CICSTS56.REXX.SCICEXEC データ・セット内のメンバー CICSTART には、REXX/CICS 環境のデフォルト定義が含まれています。CICSTART は、CICS システムが始動した後、CICS/REXX プログラムを使用する最初のトランザクションが出される時に実行されます。

MVS PDS REXX 許可ライブラリー

DD 名 CICAUTH および CICEXEC に割り振られる MVS 区分データ・セットは、すべて REXX/CICS 許可ライブラリーと見なされます。

複数のデータ・セットが連結されている場合、それらは連結の順序で検索されます。CICSTART exec は CICEXEC にあります。

- ユーザーは、REXX/CICS PATH コマンドを使用して、独自の非許可ライブラリーの動的割り振りを行うことができます。
- すべてのユーザーは CICEXEC データ・セットから exec を実行できます。また、CICEXEC からロードされたすべての exec は REXX/CICS 許可コマンドを使用できます。
- CICEXEC JCL DD ステートメントにあるデータ・セットは、可変ブロック化形式でなければなりません。

許可ユーザーの定義

AUTHUSER コマンドを使用して、ユーザーを許可ユーザーとして指定できます。

すべての AUTHUSER コマンドを CICSTART exec 内、または CICSTART exec から発行される exec 内に入れることをお勧めします。116 ページの『CICSTART メンバーの更新』を参照してください。

システム・オプションの設定

システム・オプションは、REXX/CICS SETSYS コマンドを使用して指定できます。

システム全体にわたる SETSYS コマンドを CICSTART exec 内に入れることをお勧めします。116 ページの『CICSTART メンバーの更新』を参照してください。

REXX ファイル・システム (RFS) ファイル・プールの定義

RFS ファイル・プールを定義したり、初期化したり、ファイルを追加したりするには、FILEPOOL のさまざまなコマンドを使用します。RFS ファイル共用許可には、RFS AUTH コマンドを使用します。

- FILEPOOL DEFINE コマンドを使用して、RFS ファイル・プールを定義します。
- FILEPOOL FORMAT コマンドを使用して、各ファイル・プール内の最初のファイルを初期化します。
- FILEPOOL ADD コマンドを使用して、VSAM ファイルを RFS ファイル・プールに追加します。

RFS ファイル共用の許可

RFS AUTH コマンドを使用して、ファイル共用の許可を指定します。

通常は、自分が所有するリソースの共用を許可することができます。REXX/CICS 許可ユーザーは、自分が作成したすべての RFS ディレクトリーの共用の許可を指定できます。

CICSTART のための PLT エントリーの作成

CICREXD または CICREXR プログラムを呼び出す CICS プログラム・ロード・テーブル (PLT) エントリーを作成することにより、CICS システム初期設定の直後に CICSTART exec を発行することができます。

そのようにしない場合、領域を始動した後の最初の REXX/CICS ユーザーにより CICSTART exec が実行されます。

セキュリティ出口

置き換え可能なセキュリティ出口 CICSECX1 および CICSECX2 について説明します。IBM は、カスタマイズや置き換えが可能なサンプルのアセンブラー出口を提供しています。

このセクションには、プロダクト・センシティブ・プログラミング・インターフェース情報が含まれています。

注：これらの出口は、REXX/CICS と同じ領域に存在する必要があります (例えば、分散プログラム・リンクの使用は許可されません)。

CICSECX1

CICSECX1 は、MVS データ・セットのアクセス・セキュリティ出口です。この出口は EXEC CICS LINK によって呼び出され、パラメーターは COMMAREA で渡されます。

パラメーター



重要：このトピックには、プロダクト・センシティブ・プログラミング・インターフェースと関連ガイダンス情報が含まれています。

COMMAREA では、出口への入力に以下が入ります。

Parameter	バイト数	データ・タイプ	説明
1	4	フルワード	戻りコード
2	8	文字	CICS サインオン ID
3	4	文字	機能を要求
4	3		IBM で使用するために予約済み
5	44	文字	MVS データ・セット

戻りコード

0

許可される機能要求

4

リジェクトされた機能要求

関数 ID

A

ALLOCATE REQUEST

E

EXPORT 要求

F

FREE REQUEST

I

IMPORT 要求

CICSECX2

CICSECX2 は、REXX File System アクセス・セキュリティー 出口です。

パラメーター



重要: このトピックには、プロダクト・センシティブ・プログラミング・インターフェースと 関連 ガイダンス情報が含まれています。

COMMAREA では、出口への入力に以下が入ります。

Parameter	バイト数	データ・タイプ	説明
1	4	フルワード	戻りコード
2	8	文字	CICS サインオン ID
3	1	文字	機能を要求
4	3		IBM で使用するために予約済み
5	4	フルワード	完全修飾 RFS ファイル ID スtring のアドレス
6	4	フルワード	RFS ファイル ID スtring のディレクトリー・パスの長さ

戻りコード

0

許可される機能要求

ゼロ以外
許可されません

関数 ID

A
Alter

R
Read (読み取り)

U
更新

第 13 章 パフォーマンスの考慮事項

クライアント/サーバーのサポートは、パフォーマンス上の利点をかなり見込める REXX 機能です。この機能により、アプリケーション機能を提供するために、ネストされた REXX ではなくサーバー REXX exec がしばしば使用されます。そのようなサーバーのパフォーマンス上の特性は、管理の容易さの面で優れています。ネストされた exec と比べたときのサーバー exec の利点として、サーバー exec を開始すると、終了する前に複数のクライアント要求を処理できることです。そのため、パスの長さが短くなり、応答時間が向上して、多くの場合にシステム・リソースの使用量が少なくなります。詳しくは、[ハイレベルで、最適な透過的 REXX クライアント・インターフェース](#) を参照してください。

典型的な CICS アプリケーション・プログラミングとは異なり、EXEC CICS SUSPEND コマンドを発行する必要はありません。それは、REXX/CICS が一定の間隔でプログラムを自動的に一時停止することによって処理されます。

REXX/CICS exec は VSAM ベースの REXX ファイル・システムのファイル内、または MVS 区分データ・セットに常駐させることができます。CICS は、起動時に特定の MVS 区分データ・セットを CICS 領域に割り振ります。REXX PATH (データ・セット名が指定された場合のみ)、IMPORT、EXPORT、および ALLOC コマンドで、SVC 99 を使用することにより、MVS 区分データ・セットを動的に割り振ることができます。SVC 99 を過度に使用すると、CICS 領域のパフォーマンスが低下することがあります。

第 14 章 セキュリティー

REXX トランザクションは、CICS トランザクション・セキュリティを使用して制御できます。REXX トランザクションは、それが実行される CICS 領域の特質により、広く使用できるようにすることも、何人かの個人に限定することもできます。

REXX/CICS は、CICS 提供のコマンド・レベル・インタープリター・トランザクション (CECI) のより洗練されたバージョンと見なすことができます。REXX トランザクション (REXX exec の発行に使用) は CECI トランザクションとよく似ており、CICS トランザクション・セキュリティを使用して制御できます。

注: 既存の REXX exec を実行するには REXX トランザクションは必要ありませんが、ユーザーまたはプログラマーが REXX exec を作成または変更した後にテストするためには必要です。

REXX/CICS による複数トランザクション ID のサポート

REXX/CICS は、REXX 以外のトランザクション ID (TRANID) を REXX/CICS サポート・プログラムに関連付ける機能をサポートしています。

このケースでは、発行される REXX exec の名前は直前の DEFTRNID コマンドによって決まります。これにより、REXX で、トランザクション・セキュリティを exec ごとに引き続き使用することが可能になります。

REXX/CICS ファイル・セキュリティ

RFS ディレクトリー・レベルでのアクセスは、RFS AUTH コマンドおよび RFS 置き換え可能セキュリティ出口によって制御されます。

CICAUTH DD 名によってアクセスされる許可 exec へのアクセスは、DEFCMD コマンドおよび DEFSCMD コマンドの AUTH パラメーターによって制御されます。

動的に割り振られたデータ・セットへのアクセスは、データ・セット置き換え可能セキュリティ出口によって制御されます。

REXX/CICS コマンド・レベル・セキュリティ

コマンド・レベル・セキュリティを使用して、CICS (および他の製品やシステム) の機能へのアクセスを制御できます。

現在のソフトウェア・プラクティスによっては、CICS リソース・セキュリティに依存するだけでは、その有効性に限界がある場合もあります。セキュリティ管理を追加する目的で、REXX/CICS の設計にはコマンド・レベル・セキュリティという概念が取り入れられています。REXX/CICS におけるほとんどの機能はコマンドとしてアクセスされるので、コマンド・レベル・セキュリティを使用して、CICS (および他の製品やシステム) の機能へのアクセスを制御できます。例えば、VSAM ファイル・アクセスは、READ、WRITE、および REWRITE コマンドを使用して行われます。

REXX/CICS コマンド・レベル・セキュリティは、DEFSCMD および DEFCMD AUTH パラメーターの使用と、許可 REXX/CICS ライブラリー・サポートのプロビジョンによって制御されます。

コマンド実行セキュリティは、特定の REXX/CICS コマンドまたはコマンド・キーワードの使用を制御します。この制御は一般に、特定のコマンド (またはコマンド・オプション) を許可済みとして指定することによって行います。このようなコマンド指定は、DEFCMD および DEFSCMD コマンドで行います。許可コマンドを正しく実行するためには、以下のいずれかの条件に該当する必要があります。

1. このコマンドは、CICS 始動 JCL プロシーチャーの DD 名 CICAUTH または CICEXEC に割り当てられた MVS PDS からロードされた exec から実行する必要があります。
2. このコマンドは許可ユーザーによって実行される必要があります。AUTHUSER コマンドでユーザーに権限を付与できます。

REXX/CICS 許可コマンド・サポート

どの REXX/CICS コマンドも、REXX/CICS システム管理者が許可対象として指定できます。許可コマンドは、許可 REXX/CICS ユーザーが発行した exec、または許可 REXX/CICS ライブラリーからロードされた exec でのみ正常に実行できます。

許可 exec ライブラリー CICAUTH にアクセスできるのは、REXX/CICS 許可ユーザーのみです。すべてのユーザーは、CICEXEC 許可ライブラリーにある exec を実行できます。すべてのユーザーは、CICUSER 無許可ライブラリーにある exec を実行できます。許可ユーザーは、既存の許可ユーザーが定義することも、許可 exec で定義することもできます。REXX/CICS 初期設定時 (CICS 再始動後の最初の REXX/CICS トランザクション時) に呼び出される REXX/CICS CICSTART exec は、自動的に許可されます。これは、許可ユーザーおよびライブラリーを定義するための論理プレースです。

REXX/CICS ライブラリーへのアクセスは容易に制御できるので、これは CICS 実動プログラム・ライブラリーへのアクセスの制御に対応する論理側の制御になります。サイトで重要と思われるコマンド (READ、WRITE、DELETE など) を、実動領域で許可対象と定義することもできます。これは、許可コマンドを発行する exec を作成でき、許可コマンドを含むこれらの exec をすべてのユーザーが呼び出せるか他の許可ユーザーのみが呼び出せるかを決定できるのは、許可ユーザーのみであることを意味します。

注: CICS START、LINK、および XCTL コマンドを REXX/CICS 許可コマンドと再定義することにより、外部 API への REXX/CICS exec のアクセス許可を制御できます。

セキュリティ定義

一般ユーザー、許可ユーザー、許可コマンド、許可 exec、システム・ライブラリーなどの REXX/CICS のセキュリティ定義について説明します。

REXX/CICS 一般ユーザー

AUTHUSER コマンドで許可済みと定義されていない REXX/CICS ユーザーは、REXX/CICS 許可コマンドを使用できません。ただし、これらのユーザーは、ユーザー・コマンド (DEF CMD コマンドを使用して定義) および exec を定義、作成、変更、使用することができます。ユーザーは、CICEXEC ライブラリーにある REXX/CICS 許可 exec を使用することもできます (ただし、定義、作成、変更することはできません)。

個々のユーザーの情報は、ユーザー ID を指定することによって REXX/CICS 環境で保守されます。各ユーザーは一意的に識別される必要があり、各ユーザーは REXX/CICS 環境に 1 度だけサインオンする必要があります。同じユーザー ID を持つ 2 人のユーザーが同時に操作を行うと、異常な結果を招くおそれがあります。

REXX/CICS 許可ユーザー

許可ユーザーは AUTHUSER コマンドで定義され、許可 REXX/CICS コマンド (AUTH オプションを指定した DEF CMD または DEF SCMD コマンドを使用して定義されたコマンド) を使用することが許可されます。

REXX/CICS 許可コマンド

許可コマンドとは、許可ユーザーのみが使用できる REXX/CICS コマンド、または許可 exec からのみ使用できる REXX/CICS コマンドのことです。許可コマンドは、AUTH オプションを指定した DEF CMD または DEF SCMD コマンドを使用して定義されます。

REXX/CICS 許可 exec

許可 exec は、DD 名 CICEXEC または CICAUTH からロードされたプログラム (exec) であり、許可済みと見なされます。つまり、これらのプログラムは許可 REXX/CICS コマンドを使用することが許可されます。すべての REXX/CICS ユーザーは CICEXEC 許可プログラムにアクセスできますが、CICAUTH 許可プログラムにアクセスできるのは許可ユーザーだけです。

REXX/CICS システム・ライブラリー

REXX 言語で書かれたすべての許可コマンドは、DD 名 CICAUTH に連結された MVS 区分データ・セットからロードする必要があります。これらは、IBM が提供する場合と、お客様 (またはベンダー) が提供する場合があります。

すべての許可 exec は、DD 名 CICEXEC または CICAUTH に連結された MVS 区分データ・セットからロードする必要があります。これらは、IBM が提供する場合と、お客様 (またはベンダー) が提供する場合があります。

許可されていないものの、すべての REXX/CICS ユーザーが共用するユーザー exec は、DD 名 CICUSER に割り振られた、または連結された MVS 区分データ・セットに置くことができます。

注:

1. DEFCMD または DEFSCMD の AUTH オプションは、それ自体が許可コマンド・オプションです。つまり、そのコマンドを発行するユーザーが許可ユーザーであるか、許可ライブラリーからロードされた exec からそのコマンドが発行された場合のみ、AUTH を使用できます。
2. EXECLOAD および EXECDROP コマンドは許可コマンドです。したがって、許可サブライブラリーからの exec に対して EXECLOAD を実行できるのは、許可ユーザーまたは許可 exec のみです。

第 3 部 REXX for CICS Transaction Server: リファレンス

REXX/CICS または REXX for CICS Transaction Server は、アプリケーションの開発、カスタマイズ、プロトタイプ作成、およびプロシージャーのために REXX ベースのネイティブ言語環境および関連するランタイム機能を提供します。

REXX/CICS は、サポートされるすべてのリリースの CICS Transaction Server で実行できます。

このリファレンスでは、REXX for CICS インタープリター (これ以降、インタープリターまたは言語処理プログラムと呼びます) および REstructured eXtended eXecutor (REXX) 言語について説明します。本書は、経験のあるプログラマー、特に、ブロック構造の高水準言語 (例えば、PL/I、ALGOL、Pascal など) の使用経験のある方を対象にしています。

説明には、言語の用途および構文や、プログラムが REXX/CICS インタープリターの下で実行中に言語処理プログラムが言語をどのように「解釈する」かが含まれています。本書は、以下についても説明しています。

- ユーザーが REXX および言語処理プログラムとインターフェースで接続できるようにするプログラミング・サービス
- REXX 処理や、言語処理プログラムがシステム・サービス (例えば、ストレージ要求や入出力要求) にアクセスしたり使用したりする方法をユーザーがカスタマイズできるようにするカスタマイズ・サービス

第 15 章 製品機能の概要

REXX/CICS の製品機能について説明します。

以下の製品機能について説明します。

- [135 ページの『REXX/CICS 下での SAA レベル 2 REXX 言語サポート』](#)
- [135 ページの『REXX EXEC の解釈実行のためのサポート』](#)
- [135 ページの『REXX EXEC およびデータ用の CICS ベースのテキスト・エディター』](#)
- [135 ページの『REXX EXEC およびデータ用の VSAM ベースのファイル・システム』](#)
- [136 ページの『EXEC CICS コマンドの動的サポート』](#)
- [136 ページの『CEDA および CEMT のトランザクション・プログラムへの REXX インターフェース』](#)
- [136 ページの『ハイレベルなクライアント/サーバー・サポート』](#)
- [136 ページの『REXX で作成されたコマンドのサポート』](#)
- [136 ページの『REXX コマンドのコマンド定義』](#)
- [136 ページの『システム・プロファイルおよびユーザー・プロファイルの EXEC のサポート』](#)
- [137 ページの『仮想記憶域内の共用 EXEC』](#)
- [137 ページの『Db2/SQL インターフェース』](#)

REXX/CICS 下での SAA レベル 2 REXX 言語サポート

REXX/CICS は、現在 REXX 言語レベル 3.48 であり、ストリーム入出力および REXX 言語処理プログラム出口を除くすべての Systems Application Architecture® (SAA) REXX レベル 2 の機能を備えています。

REXX EXEC の解釈実行のためのサポート

REXX EXEC の解釈実行により、REXX EXEC を最初にコンパイルせずに、作成および実行することができます。インタープリターを使用することで、極めて生産性の高い開発、カスタマイズ、プロトタイピング、およびコマンド・リスト (CLIST) 処理環境が実現されます。これは、高速な開発サイクル、ソース・レベルの対話式デバッグ、およびネイティブの CICS ベース開発環境が 1 つの統合パッケージで提供されるためです。

注：REXX EXEC は、任意の CICS 対応言語で作成された CICS プログラムおよびトランザクションを自由に呼び出すことができます。

REXX EXEC およびデータ用の CICS ベースのテキスト・エディター

TSO ISPF/PDF エディターや VM/CMS XEDIT エディターと似たネイティブの CICS テキスト・エディターが REXX/CICS の一部として用意されているので、EXEC (および他のデータ) の作成や変更を CICS で直接行うことも、CICS ベースのアプリケーション・プラットフォームで行うこともできます。提供される VSAM ベースの REXX ファイル・システム (RFS) にあるファイルと、従来型の多重仮想記憶 (MVS) 区分データ・セット内にあるファイルの編集がサポートされます。

注：PATH、IMPORT、EXPORT、ALLOC、およびエディター GETPDS コマンドで指定された区分データ・セットは、SVC 99 を使用して動的に割り振られます。他の区分データ・セット (DD 名 CICAUTH、CICEXEC、および CICUSER に連結されたもの) は、領域の始動時に割り振られます。システム・パフォーマンスへの影響を最小限に抑えるため、この手法は主にファイルのマイグレーションに利用することをお勧めします。

REXX EXEC およびデータ用の VSAM ベースのファイル・システム

REXX/CICS には、REXX ファイル・システム (RFS) という階層構造で、拡張対話式エグゼクティブ (AIX®)、および VM 共用ファイル・システムに似たハイレベルのファイル・システムが組み込まれています。RFS は、各 REXX ユーザーに、EXEC およびデータを格納するためのファイル・システムを自動的に提供します。ファイル・リスト・ユーティリティはこのファイル・システムの操作をサポートし、テキスト・エ

ディターはこのファイル・システムのメンバーの編集をサポートし、実行対象の EXEC はこのファイル・システムからロードされます。このファイル・システムは、パフォーマンス、セキュリティー、および移植性の理由から VSAM をベースにしています。

EXEC CICS コマンドの動的サポート

REXX/CICS では、ほとんどの EXEC CICS アプリケーション・プログラミング・コマンドがサポートされています。これは動的インターフェースです (EXEC CICS コマンド変換の前処理ステップは不要)。このサポートは、ADDRESS CICS コマンド環境を追加することにより、提供されます。

CEDA および CEMT のトランザクション・プログラムへの REXX インターフェース

このインターフェースにより、REXX EXEC から CEDA コマンドおよび CEMT コマンドを発行し、その後の出力を端末に表示するのではなく REXX 変数に簡単に入れることができます。これを使用すると CICS の管理および操作アクティビティーの多くを簡単に自動化できるので、プログラマーの役に立ちます。

ハイレベルなクライアント/サーバー・サポート

REXX は、REXX EXEC がクライアントとして働く (REXX/CICS サーバーに対して要求を行う) ことを可能にし、また、REXX/CICS がサーバーとして働く (REXX/CICS クライアントの要求を待機し、処理することができる) ことを可能にすることにより、REXX EXEC に統合クライアント/サーバー・サポートを提供します。

REXX/CICS サーバーがクライアントからの要求を待機 (WAITREQ) し、クライアント REXX 変数の内容を取得 (C2S) および設定 (S2C) できるようにする REXX/CICS 機能が提供されます。

注: サーバーは、クライアントのネストされた EXEC として実行するのではなく、並列エンティティーとして実行します。

サーバーは、自動サーバー始動 (ASI) を使用して、最初の要求を受け取ったときに自動的に始動します。

REXX で作成されたコマンドのサポート

REXX/CICS は、ユーザーが REXX で新しい REXX/CICS コマンドを作成できるようにサポートします。これらのコマンドは、ネストされた REXX EXEC として機能せず、ネストされた異なる REXX EXEC が、そのコマンドを発行したユーザー EXEC の REXX 変数の値を取得および設定することができます。したがって、REXX で作成されたコマンドは、アセンブラーまたは他の言語で作成されたコマンドと同様の機能を持つことができます。また、(ビルディング・ブロック構造で) システム開発を迅速化するためにコマンドを REXX で短時間で作成し、その後、パフォーマンス要件により指示された場合にアセンブラー (または他の任意の CICS 対応言語) で選択的に再作成することができます。

REXX コマンドのコマンド定義

REXX/CICS には、システム管理者およびユーザーがシステム全体またはユーザー単位で新しい REXX コマンドを動的に定義できる機能が組み込まれています。REXX は、他の製品、アプリケーション、およびシステム・サービスと円滑に対話できます。新しいコマンドまたは既存のコマンドに対するコマンド定義機能を提供する目的は、REXX を使用することによって、さまざまな製品とサービスの高速かつ一貫したハイレベルな統合を容易にすることです。REXX コマンド定義は、REXX/CICS DEFSCMD コマンドおよび DEFSCMD コマンドを使用して行われます。

システム・プロファイルおよびユーザー・プロファイルの EXEC のサポート

ファイル・システムおよびユーザー環境の調整を容易にするために、REXX/CICS は、CICSTART、CICS PROF、およびユーザー PROFILE EXEC が存在する場合には、それらの実行を試みます。CICSTART はシステム・プロファイル EXEC (STARTUP プロファイル) であり、CICS システムの再始動後、最初のユーザー EXEC が実行される前に発行されます。CICS PROF は、システム・ユーザー・プロファイル EXEC であり、CICS システムの再始動後に初めてユーザーが REXX/CICS に入ったときに発行されます。CICS PROF は、ユーザー PROFILE も呼び出します。

仮想記憶域内の共用 EXEC

REXX/CICS は、仮想記憶域内にある REXX EXEC の共用コピーをサポートします。共用 EXEC は REXX アプリケーションの対話式応答時間を短縮し、共用により仮想記憶域の合計所要量が削減されます。EXEC は、EXECLOAD コマンドを使用してプリロードすることができます。システム全体の共通 EXEC は、CICSTART EXEC に EXECLOAD コマンドを配置することによるプリロードの有効な候補です。

Db2/SQL インターフェース

REXX プログラムに SQL ステートメントおよび Db2 コマンドを含めることができます。これらのステートメントは、動的に解釈されて実行されます。SQL ステートメントおよび Db2 コマンドの結果は、REXX プログラム内で使用するために REXX 変数に入れます。

第 16 章 構文図の読み方

REXX コマンド構文は、以下の構造を使用して表しています。

- 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。

>>- 記号は、ステートメントの始まりを示します。

--> 記号は、ステートメント構文が次の行に続くことを示します。

>-- 記号は、ステートメントが前の行から続いていることを示します。

-->< 記号は、ステートメントの終わりを示します。

完全なステートメントではない構成単位の構文図は、>-- 記号で始まり、--> 記号で終わります。

- 必須指定の項目は水平線 (メインパス) 上に示しています。

▶▶ STATEMENT — *required_item* ▶▶

- 任意指定の項目はメインパスの下に示しています。

▶▶ STATEMENT — *optional_item* ▶▶

- 複数の項目から選択できる場合は、一連の選択項目を縦に並べて示しています。

いずれかの項目を選択することが必須である場合は、項目の 1 つをメインパス上に示しています。

▶▶ STATEMENT — *required_choice1*
— *required_choice2* ▶▶

選択することが任意の場合は、メインパスの下に一連の選択項目を示しています。

▶▶ STATEMENT — *optional_choice1*
— *optional_choice2* ▶▶

- 一連の選択項目の中にデフォルト値がある場合は、その項目をメインパスの上に示し、その他の項目をメインパスの下に示しています。

▶▶ STATEMENT — *default_choice*
— *optional_choice*
— *optional_choice* ▶▶

- メインラインの上の左に戻る矢印は、その項目を繰り返し指定できることを示しています。

▶▶ STATEMENT — *repeatable_item* ▶▶

縦に並んだ一連の選択項目の上に繰り返しの矢印がある場合は、選択項目を繰り返し選択できることを示しています。

- 縦線で囲まれた項目は、その項目が構文図の一部であり、メインの構文図の後にその部分の詳しい構文図が表されていることを示しています。

►► STATEMENT — fragment ◄◄

fragment

►► expansion_provides_greater_detail ◄◄

- キーワードは大文字で表しています (例: PARM1)。表示どおりに正確なスペルで指定する必要がありますが、大/小文字を区別する必要はありません。変数はすべて小文字で表しています (例: *parm*x)。これらはユーザーが指定する名前または値を表しています。
- 句読点、括弧、算術演算子、またはこの種の記号が示されている場合は、構文の一部として入力する必要があります。

第 17 章 REXX の一般的な概念

REstructured eXtended eXecutor (REXX) 言語は、コマンド・プロシーチャー、アプリケーション・フロントエンド、ユーザー定義マクロ (エディターのサブコマンドなど)、ユーザー定義 XEDIT サブコマンド、プロトタイピング、およびパーソナル・コンピューティングに特に適しています。

REXX は、PL/I に似た汎用プログラム言語です。REXX には、IF、SELECT、DO WHILE、LEAVE などの一般的な構造化プログラミング命令に加えて、便利な組み込み関数が多数用意されています。

この言語で作成するプログラムの形式に、制限はありません。1 行に複数の文節を置くことも、1 つの文節が複数行に渡ることもできます。字下げを行うこともできます。したがって、プログラムの構造がはっきり分かる形式でコーディングすれば、プログラムを読みやすくすることができます。

変数の値は、使用可能なストレージにすべての変数が収容できる範囲であれば、その長さに制限はありません。

実際の最大値: 単一のストレージ要求が 16 MB 境界の固定限度を超えることはできません。この限度は、変数にすべての制御情報を加えたもののサイズに適用されます。また、数値結果を保持するために取得されるバッファーにも適用されます。

記号 (変数名) の長さは、250 文字までに制限されています。

配列の構成などの目的のために、複合シンボルを使用できます。以下に例を示します。

```
NAME.Y.Z
```

Y および Z は、変数の名前または定数シンボルにすることができます。

REXX プログラムは、REXX ファイル・システムのディレクトリー内または MVS 区分データ・セット内に配置できます。通常、REXX プログラムのファイル・タイプは、EXEC です。

REXX プログラムは、言語処理プログラム (解釈プログラム) が実行します。つまり、プログラムは、別の形式に変換 (コンパイル) されることなく、行単位およびワード単位で処理されます。これには、プログラムが構文エラーで失敗した場合にエラー個所が明確に示されるという利点があります。ユーザーが問題を理解して解決するのに役立ちます。

構造および一般的な構文

REXX プログラムの最初にコメントを入れることをお勧めします。REXX プログラムは、一連の文節で構成されます。

REXX プログラムはコメントで開始することをお勧めします。REXX プログラムの最初にコメントを入れることは、REXX/CICS では必須ではありません。しかし、移植性の理由から、以下の例のように、各 REXX プログラムの最初の行に、REXX という語を含むコメントを入れることをお勧めします。

```
/* REXX program */
...
...
EXIT
```

図 46. REXX プログラム ID を使用した例

REXX プログラムは、一連の文節で構成されており、文節は、以下の項目で構成されます。

- ゼロまたは複数のブランク (無視されます)
- 一連のトークン ([143 ページの『トークン』](#)を参照)
- ゼロまたは複数のブランク (無視されます)
- セミコロン (;) 区切り文字。これは、行末、いくつかのキーワード、またはコロン (:) によって暗黙指定される場合もあります。

概念的には、処理される前に各文節が左から右にスキャンされ、文節を構成しているトークンが検出されます。この段階で、命令キーワードが認識され、コメントが除去され、複数のブランク (リテラル・ストリング内のブランクは除く) が 1 個のブランクに変換されます。演算子文字と特殊文字に隣接するブランクも除去されます (143 ページの『トークン』を参照)。

実際の最大値: 文節の長さは 16K を超えてはなりません。

文字

文字は、データの制御または表現に使用される、定義されたエレメントのセットの 1 つのメンバーです。

通常、1 つの文字は単一のキー・ストロークで入力できます。1 つの文字のコード化表現は、その文字をデジタル形式で表したものになります。文字 (例えば、英字 A) は、そのコード化表現 (つまりエンコード方式) とは異なります。コード化文字セット (ASCII および EBCDIC など) ごとに、文字 A のエンコード方式が異なります (それぞれ、10 進値の 65 および 193)。特に記載がない限り、この説明で使用する文字は、意味を表すものであり、特定の文字コードを意味するものではありません。文字とその表現の間で変換を行う特定の組み込み関数の場合は、この例外になります。関数 C2D、C2X、D2C、X2C、および XRANGE は、使用している文字セットに依存しています。

コード・ページは、文字セットの中の各文字のエンコードを指定します。以下の点に注意してください。

- コード・ページによっては、REXX が有効なものとして定義している文字 (例えば、論理 NOT 文字である `~`) をすべては含んでいないものがあります。
- REXX で有効と定義している文字の中には、コード・ページが異なるとエンコードが異なるものがあります (例えば、感嘆符の `!` など)。

2 バイト文字セットの詳細については、453 ページの『第 33 章 2 バイト文字セット (DBCS) サポート』を参照してください。

コメント

コメントとは、一連の文字を `/*` および `*/` で区切ったものです (1 行に収まることも複数行にわたることもあります)。これらの区切り文字で囲まれた範囲には、任意の文字を使用できます。

コメントには、それぞれが、必要な区切り文字で開始され終了されていれば、他のコメントを含めることができます。これをネストされたコメントといいます。コメントはどこに入れても、またどのような長さにしてもかまいません。コメントはプログラムには何の影響も与えませんが、区切り文字として働きます。2 つのトークンの間にコメントが 1 つでもあれば、単一のトークンとしては扱われません。

```
/* This is an example of a valid REXX comment */
```

リテラル・ストリングの一部として `/*` または `*/` を含むコード行をコメントに変える場合は、特に注意してください。以下のプログラム・セグメントで考えてみます。

```
01  parse pull input
02  if substr(input,1,5) = '/*123'
03      then call process
04  dept = substr(input,32,5)
```

2 行目と 3 行目をコメント化する場合に、次のように変更することは誤りです。言語処理プログラムは、リテラル・ストリング `/*123` の一部である `/*` を、ネストされたコメントの始まりと解釈してしまうためです。そして、対応するコメントの終わり (`*/`) を探すので、プログラムの残りの部分が処理されなくなってしまう。

```
01  parse pull input
02  /* if substr(input,1,5) = '/*123'
03      then call process
04  */ dept = substr(input,32,5)
```

`/*` または `*/` を含むリテラル・ストリングの場合のこの種の問題は、連結を使用することによって回避できます。2 行目を以下のようにしてください。

```
if substr(input,1,5) = '/' || '/*123'
```

次のようにすれば、2 行目および 3 行目をコメントに正しく変えることができます。

```
01  parse pull input
02  /* if substr(input,1,5) = '/' || '*123'
03      then call process
04  */ dept = substr(input,32,5)
```

2 バイト文字セットの文字については、453 ページの『[第 33 章 2 バイト文字セット \(DBCS\) サポート](#)』および 177 ページの『[OPTIONS](#)』 命令を参照してください。

トークン

トークンとは、文節を構築する低レベルの構文の単位です。

REXX で書かれたプログラムは、トークンから構成されます。これらは、ブランクかコメント、またはトークン自身の性質によって区別されます。トークンのクラスには、以下のものがあります。

リテラル・ストリング:

リテラル・ストリングとは、何らかの文字を含む、単一引用符 (') または二重引用符 (") で区切られた文字列です。二重引用符で区切られたストリングの中で " 文字を表すには、二重引用符 "" を 2 つ重ねて使用します。同様に、単一引用符で区切られたストリングの中で ' 文字を表すには、単一引用符 '' を 2 つ重ねて使用します。リテラル・ストリングは定数であり、処理時にその内容が変更されることはありません。

文字が入っていないリテラル・ストリング (つまり、長さが 0 のストリング) を、ヌル・ストリングと呼びます。

有効なストリングの例を以下に示します。

```
'Fred'
"Don't Panic!"
'You shouldn't't'          /* Same as "You shouldn't" */
''                          /* The null string          */
```

- 直後に (が続くストリングは、関数の名前と見なされます。
- 直後に記号 X または x が続くストリングは、16 進数ストリングと見なされます。
- 直後に記号 B または b が続くストリングは、バイナリー・ストリングと見なされます。

以下に、これらの形式について説明します。

実際の最大値: リテラル・ストリングには、最大 250 文字まで使用できます。計算結果の長さが制限される要因になるのは、使用可能な記憶域の容量のみです。詳細については、[141 ページの『第 17 章 REXX の一般的な概念』](#)の注を参照してください。

16 進ストリング:

16 進数ストリングは、16 進表記のエンコード方式を使用して表現されるリテラル・ストリングです。これは、ゼロ個以上の 16 進数 (0 から 9、a から f、A から F) がペアでグループ化された任意の文字列です。必要であれば、ストリングの先頭に 0 が 1 つあるものと見なして、偶数桁の 16 進数になるようにします。必要に応じて 1 つ以上のブランクで数字のグループを分けることができます。文字列全体を単一引用符または二重引用符で区切り、直後に記号 b または B を続けます (x または X をより長い記号に含めてはいけません)。ブランクは読みやすくするためのものであり、バイト境界にのみ置くことができます (ストリングの先頭または末尾には置けません)。言語処理プログラムは、これらのブランクを無視します。16 進ストリングは、与えられた 16 進数をパックすることによりできる、リテラル・ストリングです。16 進数をパックするとブランクが取り除かれ、ペアになった 16 進数字がそれに相当する文字に変換されます。例えば、'C1'X は A に変換されます。

文字自体を直接入力できない場合でも、16 進数ストリングを使用してプログラムの中に文字を含めることができます。有効な 16 進数ストリングの例を以下に示します。

```
'ABCD'x
"1d ec f8"X
"1 d8"x
```

注: 16 進ストリングは、数値を表すストリングではありません。そうではなく、ユーザーがエンコード方式 (つまり、マシンに依存する方式で) で文字を表せるようにするための回避策です。EBCDIC では、'40'X がブランクのコードです。いずれにしても、'.....'x 形式のストリングは、ストリングを直接的に表す代わりの代替手段であるといえます。EBCDIC では、'40'x とブランクが同じであるのと同様に、'C1'x と 'A' は同じであり、同じように扱わなければなりません。

また、アセンブラー言語では 16 進数は数値の前に X を付けて表されることに注意してください。REXX は、前述の形式の 16 進数のみ受け入れます。本資料では、両方の方法で 16 進数を表している場合がありますが、REXX で 16 進数ストリングをコーディングするときは、数値の後に X を置いてください。

実際の最大値: 16 進数ストリングのバック長 (ブランクが除去されたストリング) は、最大 250 バイトでなければなりません。

2 進ストリング:

2 進数ストリングは、2 進表記のエンコード方式を使用して表現されるリテラル・ストリングです。これは、ゼロ個以上の 2 進数 (0 または 1) が 8 (バイト) または 4 (ニブル) のグループになっている任意の文字列です。最初のグループが 4 桁より少ない場合があります。その場合は、合計 4 桁にするために、最初の数字の左側に最大 3 つの 0 があると想定されます。数字のグループはオプションで 1 つ以上のブランクで区切られ、文字列全体は、対応する単一引用符または二重引用符によって区切られ、直後にシンボル b または B 記号が続きます。(b または B をより長い記号に含めてはいけません。) ブランクは読みやすくするためのものであり、バイト境界またはニブル境界にのみ置くことができます (ストリングの先頭または末尾には置けません)。言語処理プログラムは、これらのブランクを無視します。

2 進ストリングは、与えられた 2 進数をパックすることで作成されるリテラル・ストリングです。2 進数字の数が 8 の倍数でない場合は、パックの前に、先行ゼロが左側に追加されて 8 の倍数にされます。2 進ストリングを使用すると、ビットごとに文字を明示指定することができます。

有効な 2 進数ストリングの例を以下に示します。

```
'11110000'b      /* == 'f0'x      */
"101 1101"b      /* == '5d'x      */
'1'b             /* == '00000001'b and '01'x */
'10000 10101010'b /* == '0001 0000 1010 1010'b */
'b              /* == ''          */
```

実際の最大値: 16 進数ストリングのバック長 (ブランクが除去されたストリング) は、最大 250 バイトでなければなりません。

記号:

記号は、以下の文字セットから選択された文字のグループです。

- 英字 (A から Z および a から z)

コード・ページによっては、英小文字の a から z が含まれていないものがあります。

- 数字 (0 から 9)
- 文字 . ! ? および _ (下線)。

感嘆符 ! のエンコードは、使用するコード・ページによって異なります。

- 2 バイト文字セット (DBCS) 文字 (X'41' から X'FE')。これらの文字がシンボルで有効になるためには、ETMODE が有効である必要があります。

シンボル内の英字の小文字はすべて大文字に変換 (つまり、小文字 a から z が大文字 A から Z に変換) されてから使用されます。

有効な記号の例を以下に示します。

```
Fred
Albert.Hall
WHERE?
<.H.E.L.L.O>          /* This is DBCS */
```

2 バイト文字セット (DBCS) の文字については、[453 ページの『第 33 章 2 バイト文字セット \(DBCS\) サポート』](#)を参照してください。

シンボルが数字またはピリオドで始まっていない場合、そのシンボルは変数として使用でき、それに値を割り当てることができます。シンボルに値を割り当てなかった場合、その値はシンボルそのものの文字になり、英大文字に (つまり 英小文字の a から z は英大文字の A から Z) に変換されます。数字またはピリオドで始まるシンボルは定数シンボルであり、値を割り当ててはできません。

指数形式での数字の表現に対応するために、他のシンボル形式が 1 つ許可されます。この記号は、先頭が数字 (0 から 9) またはピリオドで、末尾が E または e で、その後に任意指定の符号 (- または +) が続いた後に、1 桁以上の数字 (この後に他の記号文字を続けることはできません) が続く形になります。ここで使用している符号は、記号の一部であり、演算子ではありません。

指数表記での有効な数値の例を以下に示します。

```
17.3E-12
.03e+9
```

実際の最大値: 記号を構成するときは、最大 250 文字まで使用できます。記号が変数の場合に、その値が制限される要因になるのは、使用可能な記憶域の容量のみです。詳細については、[141 ページの『第 17 章 REXX の一般的な概念』](#)の注を参照してください。

数値:

1 つ以上の 10 進数の数字から成る文字ストリングであり、任意指定の接頭部として正符号または負符号が付きます。また、必要に応じて、小数点を表す単一のピリオド (.) を含みます。数値には、従来型の指数表記で 10 の累乗を接尾部として付けることもできます。この接尾部は、E (大文字または小文字) で始まり、オプションでその後に正または負の符号、さらにその後に 10 の累乗を定義する 10 進数が 1 つ以上続きます。文字ストリングが数値として使用されると常に、NUMERIC DIGITS 命令によって指定された精度 (デフォルトは 9 桁) に丸めが行われる可能性があります。数値の詳しい定義については、[241 ページの『第 21 章 数値と算術演算』](#)を参照してください。

数値には、先行ブランク (符号がある場合は、その前後に) と、末尾ブランクを付けることができます。数値の数字の間または指数部分に、ブランクを入れることはできません。記号またはリテラル・ストリングが数値であることもあります。しかし、数値を変数の名前にすることはできません。

有効な数値の例を以下に示します。

```
12
'-17.9'
127.0650
73e+128
' + 7.9E5 '
'0E000'
```

数値は引用符で囲んでも囲まなくてもかまいません。式の中の -17.9 (引用符がない場合) は、単なる数値ではありません。これは、負演算子の後に正の数が続いたものです (その左側に項がなければ接頭部の負の場合もあります)。この演算結果は数値です。

整数はゼロ個 (つまり、なし) の小数部分を持つ数値であり、言語処理プログラムは、通常、これを指数表記では表現しません。つまり、整数では、小数点の前の桁数が、NUMERIC DIGITS の現行設定 (デフォルトで 9) を超えることはありません。

実際の最大値: 指数表記で表現される数値の指数は、最大 9 桁まで可能です。

演算子文字:

文字: + - \ / % * | & = ~ > < および文字列 >= <= \> \< \= >< <> == \== // && ||
** ~> ~< ~ = ~== >> << >>= \<< ~<< \>> ~>> <=< /= /== は演算子 ([148 ページの『演算子』](#)を参照) を表します。これらのいくつかは解析テンプレートでも使用され、等号は割り当てを示すためにも使用されます。演算子文字に隣接するブランクは除去されます。したがって、以下は同じことを意味しています。

```
345>=123
345 >=123
345 >= 123
345 > = 123
```

上記の文字の中には、一部の文字セットでは使用できないものがあります。そのような場合は、適切な変換を使用します。特に、縦線 (|) の or 文字は、分割縦線として表されることがあります。

言語全体を通して、否定文字である `¬` は、バックスラッシュ (`\`) と同義です。使用可能かどうかや個人的な好みに応じて、この 2 つの文字のどちらを使用することもできます。

特殊文字

以下の文字は、演算子からの個々の文字と一緒に使用すると、リテラル・ストリングの外側にある場合、特別な意味を持ちます。

```
, ; : ) (
```

これらの文字は、特殊文字のセットを構成します。それらは、トークン区切り文字として作用し、それらのいずれかに隣接するブランクは除去されます。括弧の外側に隣接するブランクは例外です。このようなブランクは、さらに別の特殊文字に隣接している場合にのみ削除されます。ただし、その特殊文字も括弧であり、ブランクがその外側にある場合を除きます。例えば、言語処理プログラムは `A (Z)` からはブランクを除去しません。(これは連結であり、関数呼び出しである `A(Z)` とはちがいます)。また、言語処理プログラムは `(A) + (Z)` のブランクは除去します(これは `(A)+(Z)` と同等であるためです)。

以下の例は、文節がどのようにトークンから構成されるかを示しています。

```
'REPEAT' A + 3;
```

これは、以下の 6 つのトークンから構成されます。

- リテラル・ストリング ('REPEAT')
- ブランク演算子
- 記号 (A (値を持つことができる))
- 演算子 (+)
- 2 つ目の記号 (3 (数値であり記号でもある))
- 文節区切り文字 (;)

A と + の間のブランクおよび、+ と 3 の間のブランクは除去されます。ただし、'REPEAT' と A の間のブランクの 1 つは、演算子として残ります。そのため、この文節は、以下のように作成されたものとして扱われます。

```
'REPEAT' A+3;
```

暗黙指定されるセミコロン

文節内の最後にくるコンポーネントは、セミコロン区切り文字です。言語処理プログラムは、行末、特定のキーワードの後、およびコロン (1 つのシンボルの後に続いている場合) の後にセミコロンを暗黙指定します。

つまり、セミコロンのコーディングは、1 行に複数の文節がある場合にのみ、または最後の文字がコンマである命令を終わらせるために必要です。

通常、行末は文節の終わりを意味するため、REXX は、行の最後に暗黙にセミコロンがあるものと想定します。ただし、以下の例外があります。

- ストリングの途中で行が終わる場合。
- コメントの途中で行が終わる場合。文節は、次の行に継続します。
- 最後のトークンが継続文字 (コンマ) であり、しかもその行がコメントの途中で終わらない場合。(コメントはトークンでないことに注意してください。)

REXX は、コロン (単一シンボルの後にある場合はラベル) の後、および特定のキーワード (正しいコンテキストの中にある場合) の後に、自動的にセミコロンを暗黙指定します。この影響を受けるキーワードは、ELSE、OTHERWISE、および THEN です。これらの特殊ケースによって、タイプ・ミスがかなり減ります。

注: コメント区切り文字を形成する 2 文字 (/ * および * /) を行の終わりで分割して (つまり、/ と * をそれぞれ別の行に書いて) はなりません。分割すると、暗黙指定のセミコロンが追加されてしまい、それらの

文字を正しく認識できなくなるためです。ストリング内で1つのリテラル引用符を形成する2つ連続した文字も、この行末規則が適用されます。

継続

次の行に文節を続ける方法の1つとして、コンマを使用することができます。このコンマを、継続文字と呼びます。

このコンマは、機能的にはブランクで置き換えられるため、セミコロンが暗黙指定されることはありません。行の終わりの前にある継続文字に、1つまたは複数のコメントを続けることができます。この継続文字は、ストリングの中で使用することはできず、使用しても、ストリング自身の一部として処理されます。コメントについても同じことがあてはまります。このコンマは、実行トレースに残ることに注意してください。

以下の例は、文節を継続するための継続文字の使用法を示しています。

```
say 'You can use a comma',  
    'to continue this clause.'
```

この結果は、下記のように表示されます。

```
You can use a comma to continue this clause.
```

式および演算子

REXX における式とは、1つまたは複数のデータを各種の方法で組み合わせ、通常は元のデータとは異なる結果を生み出すための一般的なメカニズムです。

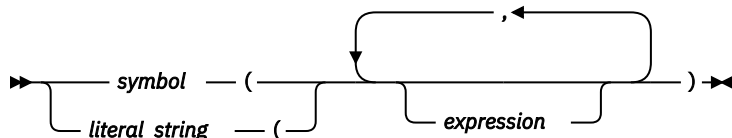
式

式は、実施する演算を示す演算子がゼロ個または複数個、間に入った1つまたは複数の項 (リテラル・ストリング、記号、関数呼び出し、または副次式) で構成されます。

副次式は、式の中で左括弧と右括弧で囲まれた項です。

項としては、以下があります。

- リテラル・ストリング (引用符で囲みます)。これは定数です。
- 記号 (引用符で囲みません)。大文字に変換されます。数字またはピリオドで始まっていない記号は、変数の名前にすることができます。その場合は、その変数の値が使用されます。それ以外の場合は、記号は定数ストリングとして処理されます。記号は、複合記号にすることもできます。
- 関数呼び出し ([193 ページの『第 19 章 関数』](#)を参照) の形式は、以下のとおりです。



式の評価は左から右に向かって行われますが、評価順序は、通常の代数計算のように括弧および演算子優先順位によって変わります ([151 ページの『括弧および演算子の優先順位』](#)を参照してください)。式の計算中にエラーが発生しない限り、式全体が計算されます。

データはすべて、「型のない」文字ストリングの形式です (データは2進、16進、配列のような、宣言された特定の型ではないため)。したがって、式が評価されると、結果は文字ストリングになります。項および結果 (算術式および論理式を除く) は、ヌル・ストリング (長さが0のストリング) の場合があります。REXX では、結果の最大長に制限がありません。1回のストレージ要求で言語処理プログラムが使用できるストレージは、16 MB に制限されていますので、注意が必要です。詳細については、[141 ページの『第 17 章 REXX の一般的な概念』](#)の注を参照してください。

演算子

演算子とは、1つまたは2つの項に対して実施される、加算などの演算を表すものです。

下記の情報では、演算子 (接頭演算子は除きます) が2つの項にどのように作用するのかを説明します。これらの項には、記号、ストリング、関数呼び出し、中間結果、または副次式があります。接頭演算子は、その後に続く項または副次式に作用します。演算子の文字に隣接するブランク (およびコメント) は、演算子に影響を与えません。したがって、複数の文字で構成される演算子には、組み込みブランクおよびコメントを入れることができます。さらに、式の中にあるが、別の演算子に隣接していない1つまたは複数のブランクは、演算子としても作用します。演算子には、以下の4種類があります。

- Concatenation (連結)
- 算術
- 比較
- 論理

文字列連結演算子

連結演算子は、1つ目のストリングの右端に2つ目のストリングを付加することで、2つのストリングを結合して1つのストリングを作成します。

連結するときには、間にブランクが入る場合と、入らない場合があります。連結演算子は、次のとおりです。

(ブランク)

1個ブランクを間に入れて、項を連結します。

||

ブランクを間に入れずに、項を連結します。

(隣接)

ブランクを間に入れずに、項を連結します。

|| 演算子を使用することにより、間にブランクを入れずに強制的に連結できます。

隣接演算子は、別の演算子で区切られていない2つの項の間にあるものと想定されます。隣接演算子が想定されるのは、2つの項が構文的に異なる (リテラル・ストリングとシンボルなど) 場合、または2つの項がコメントのみで区切られている場合です。

例

構文的に異なる項の例として、Fred の値が 37.4 のときに、Fred '%' の結果が 37.4% になる場合があげられます。

PETER 変数の値が 1 であれば、(Fred) (Peter) の結果は 37.41 になります。

隣接する次のような2つの EBCDIC のストリングがあるものとします。これらのストリングは、1つが16進数であり、もう1つがリテラルです。

```
'c1 c2'x'CDE'
```

この式の結果は、ABCDE です。

:

以下の例では、隣接演算子が暗黙指定されないため、この式は無効です。

```
Fred/* The NOT operator precedes Peter. */¬Peter
```

しかし、以下の例では、隣接演算子が暗黙指定されるため、結果は 37.40 になります。

```
(Fred)/* The NOT operator precedes Peter. *//(¬Peter)
```

算術演算子

以下の算術演算子を使用して、有効な数値である文字ストリングを結合できます ([143 ページの『トークン』](#)を参照)。

+

加算

-

減算

*

乗算

/

除算

%

整数の除算 (除算を行って、結果の整数部分を返す)

//

剰余 (除算を行って、結果が負になる可能性があるので、モジュロではなく剰余を返す)

**

累乗 (数値を整数の累乗にする)

接頭部 -

0 - 数値という減算と同じ

接頭部 +

0 + 数値の加算と、同じはたらきをする

算術演算の精度、有効な数値の形式、および演算規則については、[241 ページの『第 21 章 数値と算術演算』](#)を参照してください。演算結果が指数表記で示される場合は、丸めが行われた可能性があります。

比較演算子

比較演算子は 2 つの項を比較し、比較の結果が真であれば値 1 を返し、真でなければ 0 を返します。

厳密な比較演算子では必ず、演算子を定義する文字を 2 つ重ねて指定します。==、\==、/==、および != 演算子は、2 つのストリングが完全に一致するかどうかをテストします。2 つのストリングが厳密に等しいと見なされるには、(各文字が) まったく同じで、かつ長さが同じであることが必要です。同様に、>> や << などの厳密な比較演算子は、文字単位の単純な比較を行い、比較対象のどちらのストリングにも埋め込みを行いません。2 つのストリングの比較は、左から右への順に行われます。一方のストリングが他方のストリングよりも短く、他方のストリングの先頭から始まる一部である場合、そのストリングは他方のストリングよりも小さい (少ない) ことになります。厳密な比較演算子は、2 つのオペランドについて数値比較も行いません。

その他のすべての比較演算子では、比較する両方の項が数値であれば、数値比較が行われます (先行ゼロは無視されます。[245 ページの『数値の比較』](#)を参照してください)。そうでない場合には、両方の項が文字ストリングとして処理されます (先行および後続のブランクは無視され、短い方のストリングの右側にブランクが埋め込まれます)。

文字比較および厳密な比較演算では、いずれも大文字小文字が区別され、正確な照合順序は、実装に使用されている文字セットによって決まります。例えば、EBCDIC 環境では、小文字の英字が大文字よりも先であり、0 から 9 までの数字がすべての英字よりも先になります。

比較演算子およびその演算は、以下のようになります。

=

項が、(数値的に、または埋め込みなどを行ったときに) 等しい場合に真

!=, !=, /=

項が等しくない場合に真 (= の逆)

>

より大

<

より小

><
より大またはより小 (等しくないと同じ)

<>
より大またはより小 (等しくないと同じ)

>=
より大か等しい

¥<, ¬<
より小さくない

<=
より小か等しい

¥>, ¬>
より大きくない

==
項が厳密に等しい (同じ) 場合に真

¥==, ¬==, /==
項が厳密に等しくない場合に真 (== の逆)

>>
厳密により大

<<
厳密により小

>>=
厳密により大か等しい

¥<<, ¬<<
厳密により小さくない

<<=
厳密により小か等しい

¥>>, ¬>>
厳密により大きくない

注: 言語全体を通して、否定文字である ¬ は、バックスラッシュ (\) と同義です。これらの 2 つの文字は、使用可能かどうかや個人的な好みに応じて、どちらの文字も使用できます。バックスラッシュは、以下の演算子で使用できます: ¥ (接頭部 not)、¥=、¥==、¥<、¥>、¥<<、および ¥>>。

論理 (ブール) 演算子

文字ストリングは、0 の場合は値 false を、また、1 の場合は値 true を持つものと見なされます。論理演算子は、必要に応じて値を 1 つまたは 2 つ (0 および 1 以外の値は許可されません) とり、0 または 1 を返します。

&
AND
両方の項が真であれば 1 を返します。

|
包含 OR
いずれかの項が真であれば 1 を返します。

&&
排他 OR
いずれかの項 (両方ではない) が真ならば 1 を返します。

Prefix ¥, ¬
論理 NOT
否定。1 を 0 にし、0 を 1 にします。

括弧および演算子の優先順位

式の評価は左から右に向かって行われますが、この順序は、括弧および演算子の優先順位によって変わります。

括弧 (関数呼び出しを識別する括弧を除く) が検出された場合は、その項が要求されたときに、括弧で囲まれた副次式全体が直ちに計算されます。

以下のシーケンスが検出された場合、`operator2` の優先順位の方が `operator1` よりも高ければ、副次式 (`term2 operator2 term3`) が最初に評価されます。この規則は、必要に応じて繰り返し適用されます。

```
term1 operator1 term2 operator2 term3
```

ただし、個々の項は、式の中で左から右に向かって評価されます (つまり、項は検出されると同時に評価されます)。優先順位の規則によって左右されるのは、演算の順序だけです。

例えば、`*` (乗算) は `+` (加算) よりも優先順位が高いため、`3+2*5` の結果は `13` になります (左から右へ規則通り計算して得られる `25` ではありません)。加算してから乗算を行うようにするには、この式を `(3+2)*5` と書き直します。括弧を追加することで、最初の 3 つのトークンが副次式になります。同様に、`-3**2` という式も、負の接頭演算子の優先順位の方が累乗演算子よりも高いので、`(-9)` ではなく `9` となります。

演算子の優先順位を、以下に示します。

1. `+` `-` `~` `\` (接頭演算子)
2. `**` (累乗)
3. `*` `/` `%` `//` (乗算および除算)
4. `+` `-` (加算および減算)
5. (ブランク) `||` (隣接) (ブランクを間にはさんだ連結、またはブランクを間にはさまない連結)
6. 比較演算子:
 - `=` `>` `<`
 - `==` `>>` `<<`
 - `≠` `≠=`
 - `><` `<>`
 - `≠>` `≠>`
 - `≠<` `≠<`
 - `≠==` `≠==`
 - `≠>>` `≠>>`
 - `≠<<` `≠<<`
 - `>=` `>>=`
 - `<=` `<<=`
 - `/=` `/==`
7. `&` (AND)
8. `|` `&&` (OR、排他 OR)

例

記号 `A` は変数で値が `3` であり、`DAY` も変数でその値が `Monday` であり、その他の変数は初期設定されていないとします。式の例は、以下のように評価されます。

```
A+5          -> '8'
A-4*2        -> '-5'
A/2          -> '1.5'
0.5**2       -> '0.25'
(A+1)>7       -> '0'          /* that is, False */
```

```

' '=' -> '1' /* that is, True */
' ==' -> '0' /* that is, False */
' _=' -> '1' /* that is, True */
(A+1)*3=12 -> '1' /* that is, True */
'077'>'11' -> '1' /* that is, True */
'077' >> '11' -> '0' /* that is, False */
'abc' >> 'ab' -> '1' /* that is, True */
'abc' << 'abd' -> '1' /* that is, True */
'ab' << 'abd' -> '1' /* that is, True */
Today is Day -> 'TODAY IS Monday'
'If it is' day -> 'If it is Monday'
Substr(Day,2,3) -> 'ond' /* Substr is a function */
'!'xxx'!' -> '!XXX!'
'000000' >> '0E0000' -> '1' /* that is, True */

```

注：最後の例で、演算子が >> ではなく > の場合は、異なる解となります。すなわち、'0E0000' は指数表記の有効な数値なので、数値比較が行われます。そうすると、この場合は '0E0000' と '000000' は、等しいものとされます。REXX の優先順位は、通常の代数計算および他のコンピューター言語での優先順位と同じなので、REXX の優先順位によって特に問題となるようなことは、通常はありません。以下に示すように、通常の表記法とは異なる点が 2 つあります。

- 負の接頭演算子の優先順位は、常に、累乗演算子よりも高くなります。
- 累乗演算子は、(他の演算子と同様に) 左から右に向かって計算されます。

以下に例を示します。

```

-3**2    == 9 /* not -9 */
-(2+1)**2 == 9 /* not -9 */
2**2**3  == 64 /* not 256 */

```

文節および命令

文節は、さらにヌル文節、ラベル、命令、割り当て、キーワード命令、コマンドに分類することができます。

ヌル文節

ブランクだけかコメントだけ、またはその両方だけで構成されている文節は、ヌル文節で、完全に無視されます (ただし、その文節内にコメントが入っているために、適宜トレースされる場合は除きます)。

注：ヌル文節は、命令ではありません。例えば、IF 命令内の THEN または ELSE の後にセミコロンを余分にに入れても、(PL/I の場合のように) ダミー命令を使用したことと同じにはなりません。この目的のためには、NOP 命令が提供されています。

ラベル

1 つの記号の後にコロンが続いている文節が、ラベルです。この文脈内では、コロンによってセミコロン (文節分離文字) が暗黙指定されるので、セミコロンは不要です。ラベルは、CALL 命令、SIGNAL 命令、および内部関数呼び出しのターゲットを識別します。複数のラベルを、命令の前に置くことができます。ラベルはヌル文節として扱われ、デバッグに役立つよう、選択的にトレースできます。

連続した任意の数の文節をラベルにできます。これにより、他の文節の前に複数のラベルを置くことができます。ラベルは重複してもかまいませんが、プログラム内で重複している場合、制御が渡されるのはその最初のラベルだけです。その後に出てくる重複ラベルは、トレースすることはできますが、CALL、SIGNAL、または関数呼び出しのターゲットとしては使用できません。

ラベルには、DBCS 文字を使用することができます。453 ページの『[第 33 章 2 バイト文字セット \(DBCS\) サポート](#)』を参照してください。

命令

命令とは、1 つまたは複数の文節で構成され、言語処理プログラムが取る何らかの処置を表します。命令には、割り当て、キーワード命令、またはコマンドがあります。

割り当て

`symbol=expression` 形式の 1 つの文節が、割り当てと呼ばれる命令です。割り当てとは、変数に (新しい) 値を与えることです。153 ページの『割り当ておよび記号』を参照してください。

キーワード命令

キーワード命令は、命令を表すキーワードで最初の文節が始まる、1 つ以上の文節です。キーワード命令は、外部インターフェースと制御の流れを制御します。キーワード命令の中には、命令をネストして組み込めるものもあります。次の例では、DO 構造 (DO、その後に続く命令グループ、およびそれに関連する END キーワード) が、単一のキーワード命令と見なされます。

```
DO
  instruction
  instruction
  instruction
END
```

サブキーワードは、ある特定の命令のコンテキスト内に予約されたキーワードです (例えば、DO 命令のシンボル TO および WHILE)。

コマンド

コマンドとは、1 つの式のみからなる 1 つの文節です。式が調べられ、その結果が、コマンド・ストリングとして何らかの外部環境に渡されます。

割り当ておよび記号

変数とは、REXX プログラムの実行中に値が変わる可能性があるオブジェクトです。変数の値を変更するプロセスを、その変数への新しい値の割り当てと呼びます。

変数の値は、任意の長さの単一の文字ストリングであり、任意の文字を含められます。

変数に新しい値を割り当てる場合は、ARG 命令、PARSE 命令、PULL 命令、VALUE 組み込み関数、または変数プール・インターフェースを使用します。変数の値を変更する場合は、代入命令そのものを使用するのが最も一般的です。次の形式の文節はすべて代入と見なされます。

```
symbol=expression;
```

`expression` の結果が、等号の左側の `symbol` で指定された変数の新しい値になります。VM では現在、式を省略すると、変数にヌル・ストリングがセットされます。ただし、ヌル・ストリングは、`symbol=''` のように明示的に変数に設定することをお勧めします。以下に例を示します。

```
/* Next line gives FRED the value "Frederic" */
Fred='Frederic'
```

変数を表すシンボルは、数字 (0 から 9 まで) やピリオドで始まるものであってはなりません。(変数名の最初の文字に対するこの制約がないと、数字を再定義することになります。例えば、`3=4;` は、3 という変数に値 4 を与えることになります。)

シンボルに値を割り当てていない場合でも、そのシンボルを式の中で使用できます。これは、シンボルはいつでも定義された値を持っているためです。値を割り当てていない変数は、初期設定されません。その値は、シンボル自体の文字が大文字に変換 (つまり、小文字 `a` から `z` を大文字 `A` から `Z` に変換) されたものになります。ただし、その値が複合シンボルの場合は、シンボルの派生名です (154 ページの『複合記号』を参照)。以下に例を示します。

```
/* If Freda has not yet been assigned a value, */
/* then next line gives FRED the value "FREDA" */
Fred=Freda
```

REXX におけるシンボルの意味は、そのコンテキストによって変わります。(キーワードではなく) 式の中の項であるシンボルは、定数シンボル、単純シンボル、複合シンボル、および語幹の 4 つのグループのいずれかに属します。定数シンボルに新しい値を割り当てることはできません。名前が単一の値に対応してい

る変数には、単純シンボルを使用することができます。配列やリストなど、もっと複雑な変数の集まりに対しては、複合シンボルと語幹を使用できます。

定数記号

定数記号は、数字 (0 から 9) またはピリオドで始まります。

定数記号の値を変更することはできません。これは、単に、シンボルの文字 (つまり、英小文字を英大文字に変換したもの) からなるストリングに過ぎません。

定数記号を、以下に示します。

```
77
827.53
.12345
12e5      /* Same as 12E5 */
3D
17E-3
```

単純記号

単純記号とは、ピリオドを含んでおらず、数字 (0-9) で始まっていないものをいいます。

デフォルトでは、その値はシンボルの文字 (つまり、英大文字に変換されたもの) になります。シンボルに値がすでに割り当てられている場合には、それが変数の名前になり、その値がその変数の値になります。

単純記号を、以下に示します。

```
FRED
Whatagoodidea? /* Same as WHATAGOODIDEA? */
?12
<.D.A.T.E>
```

複合記号

複合シンボルでは、参照するときに、その名前の中の変数を置き換えることができます。

複合シンボルとは、ピリオドを少なくとも 1 つと、ピリオド以外の文字を少なくとも 2 つ含むものです。複合シンボルを数字またはピリオドで始めることはできず、複合シンボルの中にピリオドが 1 つしかない場合は、ピリオドを最後の文字とすることはできません。

名前は、語幹 (シンボルの最初のピリオドまで (ピリオドを含む) の部分) で始まります。この後に、語尾、定数シンボルである名前の部分 (ピリオドで区切ります)、単純シンボル、ヌルのいずれかが続きます。複合シンボルの派生名は、英大文字でのシンボルの語幹部分で、その後ろに、末尾部分が付いています。末尾部分では、すべての単純シンボルがそれぞれの値に置き換えられています。末尾そのものは、文字 A から Z、a から z、0 から 9、および @ # \$ % . ! ? および下線で構成されます。語尾の値はどの文字ストリングでもかまいません。これには、ヌル・ストリング、ブランクを含むストリングも含まれます。以下に例を示します。

```
taila='* ('
tailb=''
stem.taila=99
stem.tailb=stem.taila
say stem.tailb      /* Displays: 99 */
/* But the following instruction would cause an error */
/*          say stem.* ( */
```

語幹の後に、符号が埋め込まれた定数シンボル (例えば、12.3E+5) を使用することはできません。この場合、シンボル全体が無効なシンボルとなります。

複合記号を、以下に示します。

```
FRED.3
Array.I.J
AMESSY..One.2.
<.F.R.E.D>.<.A.B>
```


Before the symbol is used (that is, at the time of reference), the language processor substitutes the values of any simple symbols in the tail (I, J, and One in the examples), thus generating a new derived name. その後、この派生名は、単純シンボルと同様に使用されます。つまり、その値は、デフォルトで派生名であるか、あるいは (割り当てのターゲットとして使用されてきた場合には) 派生名によって指定された変数の値です。

シンボルに対する置き換えを行うことによって、共通の語幹を持つ変数の集合に対して、任意に指標付け (添え字付け) を行うことができます。置き換えられる値には、任意の文字 (ピリオドとブランクを含みません) を含めることができます。代入は 1 回だけ行われます。

要約すると、次に示す記号で参照される複合変数の派生名は、

```
s0.s1.s2. --- .sn
```

下記のようになります。

```
d0.v1.v2. --- .vn
```

ここで、d0 はシンボル s0 の上段シフト形式であり、v1 から vn までは、定数または単純シンボル s1 から sn までの値です。シンボル s1 から sn までの値はいずれもヌルにすることができます。値 v1 から vn もヌルにすることができ、任意の文字を含めることができます (具体的には、小文字は大文字に変換されず、ブランクは除去されず、ピリオドは特別な意味を持ちません)。

以下の一部の例は、REXX プログラムから一部を抜き出した形になっています。

```
a=3          /* assigns '3' to the variable A */
z=4          /* '4' to Z */
c='Fred'      /* 'Fred' to C */
a.z='Fred'    /* 'Fred' to A.4 */
a.fred=5      /* '5' to A.FRED */
a.c='Bill'    /* 'Bill' to A.Fred */
c.c=a.fred    /* '5' to C.Fred */
y.a.z='Annie' /* 'Annie' to Y.3.4 */

say a z c a.a a.z a.c c.a a.fred y.a.4
/* displays the string: */
/* "3 4 Fred A.3 Fred Bill C.3 5 Annie" */
```

複合シンボルを使用して、変数の配列およびリストをセットアップできます。この場合、添え字を数字にする必要はないので、プログラマーはさまざまな範囲を使用できます。便利なアプリケーションとしては、連想メモリー (内容アドレス可能) 形式の 効果をもたらす、1 つ以上の変数の値から添え字が取られる 配列をセットアップすることが挙げられます。

実際の最大値: 変数名の長さは、置換の前後とも、250 文字を超えることはできません。

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料](#)で使用されている表記規則および用語も参照してください。

語幹

語幹 は、最後の文字としてピリオドが 1 つだけ含まれているシンボルです。語幹を、数字またはピリオドで始めてはなりません。

語幹を、以下に示します。

```
FRED.
A.
<.A.B>.
```

語幹のデフォルトは、語幹の記号の文字から構成されるストリングです (つまり、大文字に変換された文字です)。記号に値をすでに割り当ててある場合には、記号が変数の名前になり、その値が変数の値になります。

語幹を割り当てのターゲットとして使用すると、その語幹で始まる名前のすべての複合変数が、既に値を割り振られているかどうかに関係なく、新しい値を受け取ります。この割り当て後は、その語幹または個

々の変数に別の値を割り当てるまでは、その語幹の付いた複合記号を参照すると、この新しい値が戻されます。

以下に例を示します。

```
hole. = "empty"
hole.9 = "full"

say hole.1 hole.mouse hole.9

/* says "empty empty full" */
```

したがって、変数の集合全体に、同じ値を割り当てることができます。以下に例を示します。

```
total. = 0
do forever
  say "Enter an amount and a name:"
  pull amount name
  if datatype(amount)='CHAR' then leave
  total.name = total.name + amount
end
```

注: 変数の集合全体に割り当てられている値は、語幹を使用することによっていつでも取得できます。ただし、派生名が語幹と同じ複合変数を使用している場合と同じ値にはなりません。以下に例を示します。

```
total. = 0
null = ""
total.null = total.null + 5
say total. total.null          /* says "0 5" */
```

変数の集合は、DROP 命令および PROCEDURE 命令を使用して、その語幹を参照することによって操作できます。DROP FRED. は、その語幹を持つすべての変数 ([172 ページの『DROP』](#)を参照) をドロップし、PROCEDURE EXPOSE FRED. は、その語幹を持つすべての可能な変数を公開します ([181 ページの『PROCEDURE』](#)を参照)。

注:

1. ARG 命令、PARSE 命令、PULL 命令、VALUE 組み込み関数、または変数プール・インターフェースで変数を変更した結果は、割り当ての場合と同じになります。値の割り当てが可能な場所で語幹を使用すると、変数の集合全体に値がセットされます。
2. 式には演算子 = を含めることができ、命令は純粹に式 ([156 ページの『外部環境に対するコマンド』](#)を参照) で構成されていてもよいので、あいまいさがあっても、以下の規則によって解決されます。すなわち、シンボルで始まり、2 番目のトークンが式 (またはキーワード命令) ではなく、等号 (=) である任意の文節が割り当てであることによって解決されます。これは制約にはなりません。文節は、最初の名前の前にヌル・ストリングを置いたり、式の最初の部分を括弧で囲んだりするなど、いくつかの方法でコマンドとして処理されるようにすることができるとのことです。

同様に、割り当てで不注意に REXX キーワードを変数名として使用しても、キーワードと変数名が混同されることはありません。例えば、以下の文節は割り当てであり、ADDRESS 命令ではありません。

```
Address='10 Downing Street';
```

3. SYMBOL 関数を使用して、シンボルに値が割り当てられているかどうかを調べられます ([216 ページの『SYMBOL』](#)を参照)。さらに、SIGNAL ON NOVALUE を設定して、初期設定されていない変数の使用をトラップできます (複合変数の末尾になっている場合を除きます。 [249 ページの『第 22 章 条件および条件トラップ』](#)を参照)。

外部環境に対するコマンド

周りの環境にコマンドを出すことは、REXX の不可欠な部分です。

環境

REXX プログラムが実行されるシステムには、コマンドを処理するための環境が少なくとも 1 つは備わっているものと想定されています。環境は、REXX プログラムに入るときにデフォルトで選択されます。こ

の環境は、ADDRESS 命令を使用することで変更できます。ADDRESS 組み込み関数を使用すると、現行環境の名前を知ることができます。使用しているオペレーティング・システムによって、REXX プログラムの外部環境が定義されます。REXX/CICS プログラムのデフォルトの環境は REXXCICS です。

コマンド

現在アドレスされている環境にコマンドを送る場合は、次の形式の文節を使用します。

```
expression;
```

式が計算されて、文字ストリング (ヌル・ストリングの場合もあります) が生成されます。次に、その文字ストリングが適切に整えられてから、使用しているシステムに実行依頼されます。式の中に計算しない部分があれば、それは引用符で囲んでください。

次に、環境がコマンドを処理します。2 次的な作用がある場合もあります。環境は、最終的に、戻りコードをセットしてから、言語処理プログラムに制御を戻します。戻りコードはストリングで、通常、数値です。処理されたコマンドに関する情報を戻します。戻りコードは、通常、コマンドが成功したかどうかを示しますが、他の情報を示すこともできます。言語処理プログラムは、この戻りコードを、REXX 特殊変数 RC に入れます。252 ページの『特殊変数』を参照してください。

戻りコードをセットする以外に、使用しているシステムは、エラーまたは障害が起こったかどうかを言語処理プログラムに示します。

- エラーとは、コマンドによって引き起こされる状態であり、通常、そのコマンドを使用するプログラムはエラーの発生に備えていることが期待されます。例えば、編集システムへ出した Locate コマンドから、requested string not found (要求されたストリングが見つからない) のエラーが報告されることがあります。
- 障害とは、コマンドによって引き起こされ、そのコマンドを使用したプログラムでは通常はリカバリーできない状態のことです (例えば、コマンドが実行可能でない場合や、見つからない場合など)。

コマンドにおけるエラー および障害は、ERROR または FAILURE の条件トラップが ON の場合には、REXX 処理に影響を与えることがあります (249 ページの『第 22 章 条件および条件トラップ』を参照してください)。TRACE E または TRACE F をセットしておく、エラーおよび障害が起きた場合にコマンドをトレースすることもできます。TRACE Normal は、TRACE F と同じであり、これがデフォルトです。187 ページの『TRACE』を参照してください。

以下に、デフォルトの REXX/CICS コマンド環境にコマンドをサブミットする例を示します。次のシーケンスでは、REXX/CICS にストリング EXECIO * READ TSQUEUE1 MYDATA. がサブミットされます。

```
TSQ1 = 'TSQUEUE1'  
"EXECIO * READ" TSQ1 "MYDATA."
```

CICS 一時記憶域キュー TSQUEUE1 が MYDATA 配列に正常に読み取られた場合、戻り時に、RC 内に置かれた戻りコードの値は 0 です。TSQUEUE1 が空の場合は、適切な戻りコードが RC に入れられます。

注: 式は、環境に渡される前に評価されることに注意してください。式の中の評価されない部分は、引用符で囲んでください。以下に例を示します。

```
"EXECIO * READ"          /* * does not mean "multiplied by" */
```

CICS で実行される REXX の基本構造

REXX/CICS のサポートにより、REXX/CICS Development System ではメイン・インターフェース・プログラム CICREXD を、REXX/CICS Runtime Facility では CICREXR を使用して、CICS 領域内に REXX EXEC をロードして実行することができます。

各 REXX EXEC は、別の CICS タスクのもとで実行されます。ネストされた REXX EXEC はいずれも、親 EXEC の CICS タスクの下で実行されます。

注: EXEC が Development System または Runtime Facility のどちらで実行されているかは、GETVERS コマンドを使用して調べることができます。390 ページの『GETVERS』を参照してください。

REXX EXEC 呼び出し

- 端末から開始された EXEC

CICS は、プログラムに関連付けられたトランザクション ID を使用して、実行するプログラムを判別します。REXX/CICS は、REXX/CICS コマンドによって作成されるテーブル DEFTRNID を使用して、CICS トランザクション ID を特定の REXX EXEC に関連付けます。REXX/CICS が提供する EXEC の CICRXTRY に関連付けられた CICS トランザクション ID は、デフォルトの REXX/CICS トランザクション ID として認識されます。提供されるデフォルトは REXX です。

REXX のみを CICS 画面から入力すると、CICRXTRY EXEC が開始されます。他のオペランドが指定された場合 (例えば REXX MYEXEC ABC)、EXEC MYEXEC が開始され、ABC が引数としてそれに渡されます。REXX/CICS のデフォルトのトランザクション ID 以外の CICS トランザクション ID の場合は、関連付けられた EXEC が開始され、その他のオペランドはすべて、その EXEC に引数として渡されます。例えば、EDIT TEST.EXEC により REXX/CICS のエディター EXEC である CICEDIT が開始され、引数 TEST.EXEC は編集または作成するファイルを指定します。REXX/CICS トランザクション ID はすべて、それらを REXX/CICS メイン・モジュール CICREXD に関連付ける CICS 定義を含んでいる必要があります。

- CICS START コマンドを使用して開始された EXEC

CICS START コマンドを使用して、REXX/CICS EXEC を開始することができます。START コマンドが、開始する CICS トランザクション ID を指定し、REXX/CICS DEFTRNID コマンドによって作成されたテーブルが、開始する EXEC を指定します。

この EXEC が CICRXTRY であり、CICS 開始データが存在する場合、最初オペランドは開始する EXEC を指定し、その他のオペランドはいずれも、引数としてその EXEC に渡されます。開始データが存在しない場合には、CICRXTRY が開始されます。

CICRXTRY 以外の EXEC の場合、開始データは引数としてその EXEC に渡されます。通常、REXX/CICS EXEC には、端末がそれらに関連付けられています。ただし、トランザクションに関連付けられた端末が REXX/CICS EXEC になく、いずれの端末出力も、廃棄されるか、または REXX/CICS SET TERMOUT コマンドで指定されたとおりに CICS 一時記憶域キューに送信されます。端末が関連付けられていないトランザクションに対して端末入力が必要されると、エラーが生成されます。

- CICS LINK コマンドまたは XCTL コマンドを使用して開始された EXEC

CICS LINK コマンドまたは XCTL コマンドを使用して、REXX/CICS インターフェース・プログラムである CICREXD または CICREXR を呼び出すことができます。この方法で呼び出された場合、インターフェース・プログラムに COMMAREA を渡す必要があります。

- CICS LINK を使用して REXX を呼び出す場合は、通信域の長さが 16 文字以上でなければなりません。先頭の 4 文字は、呼び出し側アプリケーションに完了コードを戻すために使用されます。この完了コードは、436 ページの『特定のコマンドに関連付けられていない戻りコード』にリストしている REXX/CICS 処理の戻りコードです。呼び出された EXEC の戻りコードではありません。次の 12 文字は、将来使用するために予約されています。この 16 文字の接頭部より後のデータを使用して、REXX EXEC 名とその EXEC に渡す引数をブランクで区切って渡すことができます。CICS LINK で開始する EXEC は、会話型モードで実行する必要があります。これは、呼び出し側の記憶域を保護して呼び出し元のアプリケーションに戻れるようにするために必要です。

- CICS XCTL を使用して REXX を呼び出す場合は、通信域に MVS SIB タイプ 1 の制御ブロックを含めることができます。SIB を使用しない場合、通信域の長さは 16 文字以上でなければなりません。この 16 文字の接頭部は、将来使用するために予約されています。この 16 文字の接頭部より後のデータを使用して、REXX EXEC 名とその EXEC に渡す引数をブランクで区切って渡すことができます。

- MVS SIB タイプ 1 の制御ブロックから実行される EXEC

Application Type Description (ATD) によって OfficeVision または MVS から REXX EXEC を呼び出すことができます。ATD を作成するときに、呼び出しに XCTL または START を使用します。XCTL を使用する場合は、アプリケーション・プログラムとして CICREXD または CICREXR を使用します。START を使用する場合は、アプリケーション・プログラム・フィールドに REXX/CICS のデフォルト・トランザクション ID (例: REXX) を入力します。

EXEC 名およびパラメーターを渡す方法は 2 つあります。

- MSG-TEXT フィールド - (MSG-REDEF = '1')

– Parameter Description Record (PDR) - 'REXXEXEC'

MSG-TEXT フィールドには、コマンド行で入力したデータが設定されます (ATD 名と 1 つ以上のパラメーターを指定した場合)。PDR REXXEXEC を使用する場合は、EXEC 名と必要なパラメーターを PDR のデータ域に設定する必要があります。MSG-TEXT と PDR REXXEXEC の両方を使用した場合は、MSG-TEXT 内のデータが優先されます。この 2 つのどちらも使用しない場合は、REXXTRY 対話式ユーティリティ (CICRTRY EXEC) が呼び出されます。

注: REXX 命令および REXX/CICS コマンドを対話的に実行できるようにする REXX/CICS とともに、REXXTRY (CICRTRY EXEC) というユーティリティが提供されます。このユーティリティを呼び出すには、オペランドを指定せずに、CICRTRY に関連付けられている REXX/CICS トランザクション ID を入力してください。

EXEC の実行場所

REXX/CICS EXEC は、CICS 領域内で、それらを発行した CICS タスクの一部として実行されます。REXX インタープリターは完全に再入可能であり、16 MB 境界より上で実行されます (AMODE=31、RMODE=ANY)。

EXEC の検索と読み込み

EXEC を開始できるように、EXEC を検出してストレージに読み込むのに、以下の規則が使用されます。

1. EXECLOAD を使用して記憶域に読み込まれた EXEC が検索され、該当する EXEC が見つかったら、EXECLOAD によってロードされたコピーが使用されます。
2. REXX/CICS CD コマンドによって定義されている、ユーザーの現行 RFS ディレクトリーが検索されます。見つかった場合、その EXEC が読み込まれ、開始されます。
3. REXX/CICS PATH コマンドによって定義されている、ユーザーのパスが検索されます。見つかった場合、その EXEC が読み込まれ、開始されます。
4. ユーザーが許可ユーザーである場合は、CICS 始動 JCL の CICAUTH の DD 名で指定されたデータ・セットが検索されます。該当の EXEC が見つかったら、それが読み込まれ、開始されます。
5. CICEXEC と CICUSER の DD 名で (この順序で) 指定されたデータ・セットが検索されます。EXEC が見つからない場合、エラー・コードが戻されます。

EXEC の編集

REXX/CICS Development System がインストールされている場合は、用意されている REXX/CICS エディターを使用して REXX EXEC を編集できます。255 ページの『[第 23 章 REXX/CICS テキスト・エディター](#)』を参照してください。また、REXX/CICS EXEC が MVS PDS にある場合は、TSO の ISPF/PDF エディター (または他の互換エディター) を使用してそれらを編集できます。

REXX/CICS EXEC の 73 桁目から 80 桁目でシーケンス番号を使用することはできません。

REXX ファイル・システム

EXEC は、REXX/CICS に付属する VSAM ベースの REXX File System (RFS) に、または MVS 区分データ・セットに、メンバーとして保存できます。283 ページの『[第 24 章 REXX/CICS ファイル・システム](#)』を参照してください。

注: EXEC を呼び出すために指定したファイル ID にファイル・タイプ拡張子が含まれていない場合、REXX EXEC の検出および発行を試みると、EXEC というファイル・タイプをもつ RFS ファイルのみが検索されます。ファイル ID にファイル・タイプ拡張子が含まれている場合には、該当の EXEC の検出および実行を試みると、一致するファイル・タイプをもつ RFS ファイルのみが検索されます。MVS PDS の検索では、ファイル名のみが PDS メンバー名と比較されます。

EXEC 実行検索順序の制御

EXEC を読み込もうとしたときにユーザー REXX ライブラリーが検索される順序は、CD コマンドおよび PATH コマンドによって定義されます。現行ディレクトリー (CD コマンドで設定) が検索された後、PATH コマンドで指定したすべてのディレクトリーが検索され、次に、CICS 領域の始動 JCL で指定したシステム

MVS PDS ライブラリーが検索されます。検索順序について詳しくは、[193 ページの『関数およびサブルーチン』](#)を参照してください。

ユーザー作成コマンドの追加

REXX/CICS コマンドとして呼び出せるように EXEC を定義することができます。REXX/CICS ユーザー・コマンドを定義するには、DEFCMD コマンドが使用されます。DEFCMD は、REXX のほか、標準 CICS 対応言語で作成されたコマンドをサポートします。[374 ページの『DEFCMD』](#)を参照してください。

標準 REXX 機能のサポート

SAY や TRACE ステートメント、PULL や PARSE EXTERNAL ステートメント、REXX スタッキング、REXX 関数など、標準の REXX 機能がサポートされています。

SAY ステートメントと TRACE ステートメント

REXX SAY 端末入出力および TRACE 端末入出力の出力ステートメントは、CICS 端末管理サポートを使用して、シミュレートされた行モード出力を提供します。また、SET TERMOUT コマンドを使用して、行モード出力を経路指定して一時記憶域キューに入れることもできます。[403 ページの『SET』](#)を参照してください。

PULL ステートメントと PARSE EXTERNAL ステートメント

REXX PULL および PARSE EXTERNAL の端末入出力の入力ステートメントは、CICS 端末管理サポートを使用して、シミュレートされた行モード入力を提供します。

1. PULL (または PARSE PULL) は、まずプログラム・スタックから 1 行プルしようと試み、その行が空の場合にのみ、端末への読み取りを発行します。
2. 非端末接続トランザクションの一部として実行している REXX EXEC から端末行モード入力を実行しようとする、エラーとなり、その EXEC は終了してエラー・メッセージが表示されます。

REXX スタック・サポート

各ユーザーは、REXX EXEC の複数世代間の共用プログラム・スタックをもっています。この単一自動プログラム・スタックには名前がありません。名前付きのプログラム・スタックが必要な場合は、RLS LPUSH コマンド、LQUEUE コマンド、および LPULL コマンドを使用してください。

REXX 関数サポート

REXX/CICS は、標準 SAA レベル 2 組み込み関数セットをサポートします。ただし、以下の例外があります。

- ストリーム入出力関数はサポートされません。
- USERID 関数は、ユーザーがサインオンしている場合、1 文字から 8 文字の CICS ユーザー ID を戻します。CICS ユーザーがサインオンしておらず、CICS 領域にデフォルトのユーザーが (CICS 始動パラメーターで DFLTUSER を指定する CICS システム・プログラマーによって) 指定されている場合は、その値が使用されます。

注：デフォルトのユーザーは、REXX ファイル・システム・ディレクトリーと REXX List System ディレクトリーを共用します。

- STORAGE 関数は、REXX ユーザーが CICS 領域の仮想記憶域を表示または変更することができます。この関数は、許可 EXEC から、あるいは許可ユーザーによってのみ、正常に呼び出すことができます。

REXX コマンド環境のサポート

現在使用可能な (REXX ADDRESS コマンドで使用する) REXX コマンド環境は、REXXCICS、CICS、EXEC SQL、EDITSVR、FLSTSVR、RFS、および RLS です。

REXX ホスト・コマンド環境の追加

新しい REXX/CICS コマンドおよびコマンド環境を動的に定義できるようにするために、サポートが提供されています。新しいコマンドは、REXX 対応言語または任意の REXX/CICS 対応言語 (例えば、Assembler、COBOL、C、PL/I) で作成される可能性があります。REXX/CICS に対するコマンドおよびコマンド環境の定義方法について詳しくは、[307 ページの『第 26 章 REXX/CICS コマンド定義』](#)を参照してください。

標準 CICS 機能のサポート

標準 CICS 機能またはファシリティーのサポートについて説明します。CICS マップ入出力サポート、データ・セット入出力サービス、CICS 機能およびサービスへのインターフェース、EXEC からのユーザー・アプリケーションの発行、CICS の一時記憶域キューへの REXX インターフェース、疑似会話型トランザクション・サポート、DBCS サポートに関する説明が含まれます。

CICS マップ入出力サポート

CICS 基本マッピング・サポート (BMS) 入出力のサポートは、CICS SEND MAP、RECEIVE MAP、および CONVERSE MAP の各コマンドと、REXX/CICS CONVTPMAP コマンドおよび COPYS2R コマンドによって提供されます。[473 ページの『第 36 章 基本マッピング・サポートの例』](#)を参照してください。

注: BMS マップは、通常の CICS プロシージャーを使用して事前定義しておく必要があります。

データ・セット入出力サービス

標準 CICS ファイル入出力コマンド (EXEC CICS READ や WRITE など) がサポートされます。提供されている RFS コマンドを使用して、EXEC から VSAM ベースの REXX ファイル・システム (RFS) への高水準入出力を実行できます。また、動的割り振りを使用して、標準の MVS 区分データ・セットを IMPORT と EXPORT コマンドや REXX/CICS エディターで使用することもできます。

CICS 機能およびサービスへのインターフェース

ADDRESS CICS コマンド環境から、ほとんどの CICS コマンド (アプリケーション開発のリファレンスで定義されています) がサポートされます。サポートされるコマンドに関して詳しくは、[349 ページの『第 30 章 REXX/CICS コマンド』](#)を参照してください。

EXEC からのユーザー・アプリケーションの発行

REXX/CICS は、CICS トランザクションを開始したり、REXX EXEC 内から CICS プログラムを起動したりできるようにする EXEC CICS START、LINK、および XCTL の各コマンドをサポートします。

CICS ストレージ・キューへの REXX インターフェース

CICS 一時記憶域および一時データ・キューの読み取り、書き込み、および REXX/CICS からの削除について、コマンド・サポートがあります。

疑似会話型トランザクション・サポート

REXX EXEC に対する CICS 疑似会話型サポートは、CICS RETURN TRANSID() コマンドを使用することで、REXX/CICS PSEUDO コマンド ([396 ページの『PSEUDO』](#)を参照) および SETSYS PSEUDO コマンド ([405 ページの『SETSYS』](#)を参照) によってサポートされます。

他のプログラミング言語へのインターフェース

REXX/CICS では、任意の REXX/CICS 対応言語で作成された CICS プログラムを呼び出すことができます (CICS LINK コマンドおよび CICS XCTL REXX/CICS コマンドのサポートによる)。DEFCMD コマンドおよび

DEFSCMD コマンドを使用して定義された新規 REXX コマンドのインプリメントに使用されるプログラムに対しても同じサポートが用意されています。EXEC CICS START を REXX EXEC から発行できるようにするサポートも用意されています。

DBCS サポート

REXX/CICS ユーザーは、SAA レベル 2 REXX に含まれているあらゆる種類の DBCS 関数と処理技法を使用することができます。

その他の機能

- CICS ユーザーの 4 文字の端末 ID を戻すために、TERMID コマンドが提供されています。
- REXXTRY 対話式ユーティリティ (CICRTRY EXEC) を使用中に、検索 PF キーを指定することにより、行モード入出力を使用して入力された最後の入力行を検索できます。SET RETRIEVE コマンド ([403 ページの『SET』](#)) を参照してください。
- SET TERMOUT コマンドは、(SAY または TRACE からの) 行モード端末出力を、端末の代わりに、あるいは端末に加えて、CICS 一時記憶域キューに送信できるようにします。
- PULL 命令は、PULL が端末からデータを読み取った場合、押された AID キーの名前を使用して REXX 変数 PULLKEY を設定します。
- 端末への行モード出力があると、フルスクリーンでは、画面の右下隅に MORE が表示されます。Clear を押すか、Enter を押して続行します。
- 端末からの行モード入力があると、画面の右下隅に READ が表示されます。

第 18 章 キーワード命令

キーワード命令は、命令を表すキーワードで最初の文節が始まる、1つ以上の文節です。制御の流れを左右するキーワード命令もあれば、プログラマーにサービスを提供するキーワード命令もあります。DO などのキーワード命令には、ネストした命令を組み込むことができます。

構文図では、大文字のシンボル(ワード)はキーワードまたはサブキーワードを示します。一部のワード(*expression* など)は、以前に定義されたトークンのコレクションを示すものがあります。ただし、キーワードおよびサブキーワードは大文字でも小文字でもかまいません。すなわち、if、If、および iF といった記号は、いずれも同じ結果をもたらします。また、行の終わりによって文節区切り文字(;)が暗黙指定されるので、通常は、それらの文節区切り文字の大半は省略することができます。

152 ページの『文節および命令』で説明したように、キーワード命令であると認識されるのは、そのキーワードが文節内の最初のトークンであり、しかも 2 番目のトークンが = 文字(割り当てを意味する)またはコロン(ラベルを表す)で始まっていない場合だけです。キーワード ELSE、END、OTHERWISE、THEN、および WHEN は、同じ状況で認識されます。REXX で定義されたキーワードで始まる文節は、コマンドではあり得ません。したがって、次のコードは ARG キーワード命令であり、ARG 組み込み関数の呼び出しで始まるコマンドではありません。

```
arg(fred) rest
```

DO、IF、または SELECT の各命令内のキーワードの位置が正しくないと、構文エラーになります。(キーワード THEN も、IF または WHEN 文節の本体部分として認識されます。)それ以外の文脈では、キーワードが予約されていないので、ラベルまたは変数の名前としてキーワードを使用することができます(ただし、一般的には、この方法はお勧めできません)。

その他の特定のキーワードで、サブキーワードと呼ばれるものは、個々の命令の文節内で予約されています。例えば、シンボルの VALUE および WITH は、それぞれ、ADDRESS 命令および PARSE 命令内のサブキーワードです。詳細については、個々の命令の説明をお読みください。予約されているキーワードについての一般的な説明は、469 ページの『予約キーワード』を参照してください。

キーワードに隣接する空白は、そのキーワードを後続のトークンから分離するためだけのものです。下記の例では、*expression* をサブキーワードから離すために、VALUE の後に 1 つまたは複数の空白が必要です。

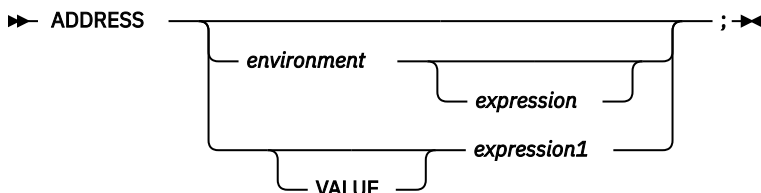
```
ADDRESS VALUE expression
```

ただし、下記の例では、VALUE サブキーワードの後に空白は必要ありませんが、空白を入れれば読みやすくはなります。

```
ADDRESS VALUE 'ENVIR' || number
```

ADDRESS

ADDRESS は、コマンドの宛先を一時的または永続的に変更します。コマンドとは、外部環境に送信されるストリングのことです。コマンドを送信するには、1 つの式だけからなる文節を指定するか、ADDRESS 命令を使用します。



代替サブコマンド環境の概念について、92 ページの『プログラムからのコマンドの発行』で説明しています。

指定した環境に単一のコマンドを送るには、リテラル・ストリングまたは単一記号 (定数と見なされる) で *environment* をコーディングし、その後に *expression* を続けます。(環境名は、コマンドの実行ができる外部プロシージャ または処理の名前です。) 環境名は、8 文字までに制限されています。 *expression* が計算され、その結果のストリングがコマンドとして処理されるべく、 *environment* あてに経路指定されます。(式の中に計算させたくない部分があれば、それを引用符で囲んでください。) コマンドを実行した後、 *environment* は、コマンド実行前の設定値に戻されます。単一のコマンドについては、このようにして一時的に宛先を変更します。特殊変数 RC は、他のコマンドの場合と同様にセットされます。(156 ページの『外部環境に対するコマンド』を参照。) このようにして処理されたコマンドのエラーおよび障害は、通常どおりにトラップまたはトレースされます。

例

```
ADDRESS CICS "READQ TSQ QUEUE('QUEUE1') INTO(VAR1)" /* CICS */
```

environment のみを指定した場合、宛先の永続的な変更が発生し、後に続くコマンド (REXX 命令でも割り当て命令でもない文節) はすべて、次の ADDRESS 命令が処理されるまで、指定されたコマンド環境に送られます。その前に選択されていた環境は、保管されます。

```
address cics
"READQ TSQ QUEUE('QUEUE1') INTO(VAR1)"
ADDRESS RFS
'COPY PROFILE.EXEC TEMP.EXEC'
```

同様に、VALUE 形式を使用すると、環境に対する永続的な変更を行うことができます。この場合は、*expression1* (単なる変数名でもかまいません) が計算され、その結果によって環境名が形成されます。*expression1* がリテラル・ストリングまたは記号で始まっていない (すなわち、演算子文字や括弧などの特殊文字で始まっている) 場合は、VALUE サブキーワードを省略することができます。

```
ADDRESS ('ENVIR'||number) /* Same as ADDRESS VALUE 'ENVIR'||number */
```

引数を指定しないと、環境が永続的に変更される前に選択されていた環境にコマンドの宛先が戻され、現行の環境名は保管されます。したがって、環境が変更された後で、ADDRESS だけを繰り返し実行すると、コマンドの宛先が 2 つの環境の間で交互に切り替わります。

2 つの環境名は、内部サブルーチン、外部サブルーチン、および関数呼び出しのたびに自動的に保管されます。165 ページの『CALL』を参照してください。

アドレスは、現在選択されている環境名にセットされます。ADDRESS 組み込み関数を使用して、現在のアドレス設定値を検索できます (197 ページの『ADDRESS』を参照)。

ARG

ARG は、プログラムまたは内部ルーチンに提供された引数ストリングを検索して、それらを変数に割り当てます。

```
➡ ARG _____ ;<
    |_____|
    template_list
```

ARG は、次の命令の短縮形です。

```
➡ PARSE UPPER ARG _____ ;<
    |_____|
    template_list
```

template_list は、多くの場合は単一のテンプレートですが、複数のテンプレートをコンマで区切って指定することもできます。テンプレートを指定する場合、各テンプレートは、ブランクまたはパターン (あるいはその両方) で区切ったシンボルのリストです。

サブルーチンまたは内部関数の処理中を除き、パラメーターとしてプログラムに渡されるストリングは、構文解析に関するセクション (227 ページの『第 20 章 構文解析』) で説明されている規則に従って変数に解析されます。

サブルーチンまたは内部関数の実行中に使用されるデータは、呼び出し元がそのルーチンに渡す引数ストリングです。

いずれの場合も、渡されたストリングは、大文字に (つまり、小文字の a-z が大文字の A-Z に) 変換されてから、処理されます。大文字変換を望まない場合には、PARSE ARG 命令を使用してください。

同じソース・ストリング (単数または複数) に対して、ARG 命令 および PARSE ARG 命令を繰り返し使用できます (通常、テンプレートは異なるものを使用します)。ソース・ストリングは変わりません。解析されるデータの長さまたは内容に関する制約事項は、呼び出し元によって課せられるもの以外はありません。

例

```
/* String passed is "Easy Rider" */
Arg adjective noun .

/* Now:  ADJECTIVE  contains 'EASY'          */
/*       NOUN       contains 'RIDER'         */
```

プログラムまたはルーチンで複数のストリングを使用したい場合、解析する *template_list* の中にコンマを使用して、各テンプレートが順番に選択されるようにすることができます。

```
/* Function is called by  FRED('data X',1,5) */
Fred: Arg string, num1, num2

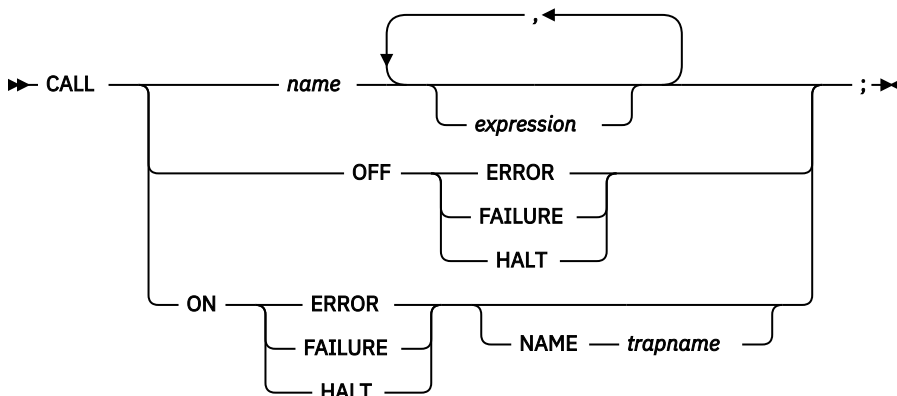
/* Now:  STRING  contains 'DATA X'          */
/*       NUM1    contains '1'               */
/*       NUM2    contains '5'               */
```

注:

1. ARG 組み込み関数は、REXX プログラムまたは内部ルーチンに対する引数ストリングの検索や検査も行うことができます。197 ページの『ARG (引数)』。
2. プログラムへ入る時点で、処理されているデータのソースも使用可能にされます。詳細については、PARSE 命令 (SOURCE オプション) 179 ページの『PARSE』を参照してください。

CALL

CALL は、ルーチン呼び出す (*name* を指定した場合) か、または特定の条件をトラップするかしないかを制御 (ON または OFF を指定した場合) します。



トラップingを制御するには、OFF または ON のほかに、トラップしたい条件を指定します。OFF は、指定された条件トラップをオフにします。ON は、指定された条件トラップをオンにします。条件トラップについては、249 ページの『第 22 章 条件および条件トラップ』を参照してください。

ルーチン呼び出すには、*name* (定数と見なされるリテラル・ストリングまたはシンボル) を指定します。*name* は、記号 (リテラルとして扱われます) またはリテラル・ストリングでなければなりません。呼び出し可能なルーチンは、以下のとおりです。

内部ルーチン

CALL 命令と同じプログラム、または CALL 命令を呼び出す関数呼び出しと同じプログラムの内部に存在する関数またはサブルーチン。

組み込みルーチン

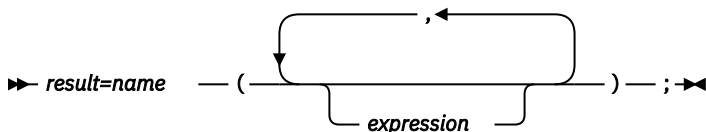
REXX 言語の一部として定義されている関数 (サブルーチンとして 呼び出すことができます)。

外部ルーチン

組み込まれたものでもなく、CALL 命令と同じプログラムまたは CALL 命令を呼び出す関数呼び出しと同じプログラムの内部にも存在しない関数またはサブルーチン。

name がストリングならば (つまり、引用符で囲んで *name* を指定した場合)、内部ルーチンの探索は迂回され、組み込み関数または外部ルーチンだけが呼び出されます。組み込み関数の名前は **大文字** です (通常は外部ルーチンの名前も **大文字** です)。したがって、リテラル・ストリングで名前を指定する場合も **大文字** にする必要があります。

呼び出されたルーチンは、オプションで結果を戻すことができます。その場合の CALL 命令は、以下の文節と機能的に同じです。



呼び出されたルーチンが結果を戻さない場合、(前述のように) それを関数として呼び出すと、エラーとなります。

REXX/CICS では、最大 20 個までの式 (expression) をコンマで区切って指定することができます。式は、ルーチンの実行時に左から右へ向かって評価され、引数ストリングが形成されます。CALL 命令に制御が戻されるまでの間、呼び出されたルーチン内の ARG 命令、PARSE ARG 命令、あるいは ARG 組み込み関数がアクセスするのは、これらのストリングであり、呼び出しプログラム内でその前にアクティブであったストリングではありません。必要に応じて余分にコンマを指定すれば、式を省略することができます。

CALL は、その後、関数呼び出しとまったく同じメカニズムを使用して、*name* という名前のルーチンにブランチします。[193 ページの『第 19 章 関数』](#)を参照してください。検索順序は、関数に関するセクションに記載されていますが、概略は次のとおりです。

内部ルーチン:

同じプログラム内の一連の命令であり、CALL 命令で指定する *name* と同一のラベルで始まります。ルーチン名を引用符で囲んで指定すると、その探索順序に内部ルーチンは含まれません。SIGNAL と CALL を一緒に使用すると、実行の時点で名前が決定される内部ルーチンを呼び出すことができます。これは、マルチウェイ呼び出しと呼ばれます ([186 ページの『SIGNAL』](#)を参照)。RETURN 命令が、内部ルーチンの実行を完了させます。

組み込みルーチン:

さまざまな関数を提供するために言語処理プログラムに組み込まれているルーチンのことです。これらは必ず、ルーチンの結果であるストリングを戻します。([196 ページの『組み込み関数』](#)を参照。)

外部ルーチン:

ユーザーは、言語処理プログラムおよび呼び出しプログラムにとって外部のルーチンを、作成または使用することができます。外部ルーチンは、REXX でコーディングする必要があります。REXX で作成された外部ルーチンを CALL 命令によってサブルーチンとして呼び出す場合には、ARG 命令、PARSE ARG 命令、あるいは ARG 組み込み関数を使用して、引数ストリングを検索することができます。

内部ルーチンの実行時には、認識済みのすべての変数を正常にアクセスすることができます。ただし、PROCEDURE 命令は、ローカル変数環境をセットアップして、サブルーチンおよび呼び出し元を互いに保護することができます。PROCEDURE 命令の EXPOSE オプションを使用すれば、選択した変数をルーチンに公開することができます。

サブルーチンとしての外部プログラムの呼び出しは、内部ルーチンの呼び出しと似ています。ただし、呼び出し元のすべての変数が常に隠されているという点において、外部ルーチンは暗黙の PROCEDURE であるといえます。内部値 (NUMERIC 設定値など) の状況は、(呼び出し元から引き継いだ値ではなく) それらのデフォルトで始まります。さらに、EXIT を使用してルーチンから戻ることができます。

制御が内部ルーチンに到達すると、CALL 命令の行番号が SIGL 変数 (呼び出し元の変数環境内にあります) で使用可能になります。これは、そのまま、デバッグ補助機能として使用できるため、制御がルーチンにどのように到達したかが分かります。内部ルーチンで PROCEDURE 命令を使用する場合は、CALL の行番号にアクセスするために EXPOSE SIGL が必要になります。

最終的には、サブルーチンは RETURN 命令を処理する必要があり、制御は、その地点から、元の CALL の後の文節に戻ります。RETURN 命令で式を指定した場合、変数 RESULT には、その式の値が設定されます。そうでない場合には、RESULT 変数はドロップされます (初期設定されません)。

内部ルーチンには、他の内部ルーチンの呼び出しだけでなく、それ自身に対する再帰呼び出しも含めることができます。

実際の最大値: 内部ルーチン呼び出しを含む制御構造の合計ネスト数は、使用可能なストレージによって異なります。

例

```
/* Recursive subroutine execution... */
arg z
call factorial z
say z'!' =' result
exit

factorial: procedure      /* Calculate factorial by */
  arg n                  /* recursive invocation. */
  if n=0 then return 1
  call factorial n-1
  return result * n
```

内部サブルーチン (および関数) の実行時には、重要な情報がすべて自動的に保管され、後でルーチンから戻るときに復元されます。該当する HTTP ヘッダーは、以下のとおりです。

- DO ループおよびその他の構造の状況。

サブルーチン内で SIGNAL を実行すれば安全です。そのサブルーチンが呼び出されたときにアクティブであった DO ループなどが終了していないためです。 (ただし、サブルーチン内の現在アクティブなものは、終了します。)

- トレース活動。

サブルーチンがデバッグされていれば、TRACE Off をサブルーチンの先頭に挿入することができます。これは、呼び出し元のトレースには影響を与えません。逆に、サブルーチンをデバッグしたい場合は、先頭に TRACE Results を挿入しておけば、戻り時に、サブルーチンに入った時点の条件 (例えば、Off) に自動的に復元されます。同様に、? (対話式デバッグ) および ! (コマンド禁止) は、複数のルーチンにまたがって保管されます。

- NUMERIC 設定値。

算術演算の DIGITS、FUZZ、および FORM (176 ページの『NUMERIC』を参照) は、保管され、その後、戻るときに復元されます。したがって、サブルーチンは、呼び出し元に影響を与えずに、必要な精度などを設定することができます。

- ADDRESS 設定値。

コマンドの現行の宛先および以前の宛先 (163 ページの『ADDRESS』を参照) は、保管され、その後、戻るときに復元されます。

- 条件トラップ。

条件トラップ (CALL ON および SIGNAL ON) は、保管され、戻り時には復元されます。つまり、CALL ON、CALL OFF、SIGNAL ON、および SIGNAL OFF をサブルーチン内で使用しても、呼び出し元がセットアップした条件には影響を与えません。

- 条件情報。

この情報は、現在トラップされている条件の状態および発生点について記述したものです。CONDITION 組み込み関数がこの情報を戻します。200 ページの『CONDITION』を参照してください。

- 経過時間クロック。

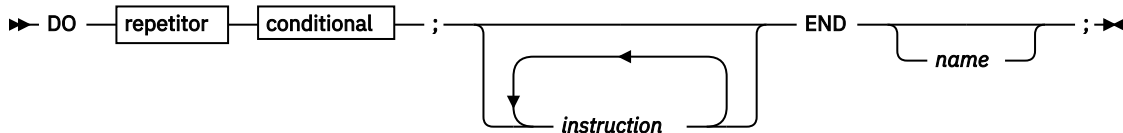
サブルーチンは、その呼び出し元から経過時間クロック (216 ページの『TIME』を参照) を継承しますが、時間クロックは複数のルーチン呼び出しにわたって保管されるため、その呼び出し元に影響を与えずに、サブルーチンまたは内部関数を独立して再始動し、クロックを使用することができます。おなじ理由のため、内部ルーチンの中で始動されたクロックは、その呼び出し元では使用できません。

- OPTIONS 設定値。

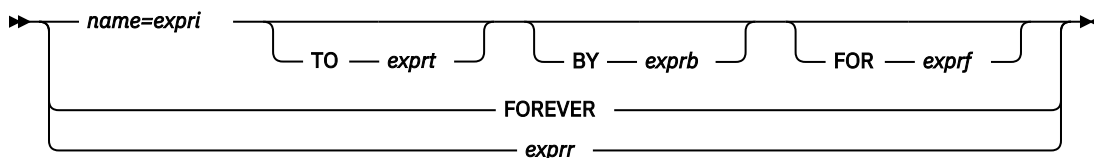
ETMODE および EXMODE は、保管され、戻る時に復元されます。詳しくは、177 ページの『OPTIONS』を参照してください。

DO

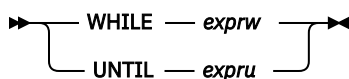
DO は、命令をまとめてグループ化し、オプションで、それらを繰り返し処理します。反復実行の間に、制御変数 (*name*) を、ある値の範囲で段階的に変更できます。



repetitor



conditional



DO は、命令をまとめてグループ化し、オプションで、それらを繰り返し処理します。反復実行の間に、制御変数 (*name*) を、ある値の範囲で段階的に変更できます。

構文に関する注意事項

- *exprr*, *expri*, *expri*, *expri*, および *expri* オプション (指定する場合) は、計算結果が数値になる任意の式にします。 *exprr* オプションおよび *expri* オプションの場合は、さらに、その計算結果が負でない整数 (正またはゼロ) にならなければなりません。数値は、必要に応じて、NUMERIC DIGITS の設定値に従って丸められます。
- *expri* オプションまたは *expri* オプション (指定する場合) は、計算結果が 1 または 0 になる任意の式にすることができます。
- TO, BY, および FOR 句を使用する場合は、どのような順序で指定してもかまいませんが、それらの計算は、句を書いた順に行われます。
- *instruction* は、割り当て、コマンド、およびキーワード命令 (IF, SELECT, および DO 命令自身などのより複雑な構造の命令も含みます) などの、任意の命令にすることができます。
- サブキーワードの WHILE および UNTIL は DO 命令内で予約されているため、DO 命令内ではどの式内の記号としても使用できません。同様に、TO, BY, および FOR を、*expri*, *expri*, *expri*, または *expri* の中で使用することもできません。FOREVER も予約されていますが、これは DO キーワードの直後に続いていて、その後には符号が続いていない場合に限りです。
- *expri* オプションのデフォルトは、1 (該当する場合) です。

単純 DO グループ

repetitor も *conditional* も指定しない場合は、いくつかの命令を単にまとめただけの構成になります。これらの命令は、1 回実行されます。

下記の例では、命令が 1 回だけ実行されます。

```
/* The two instructions between DO and END are both */
/* processed if A has the value "3". */
If a=3 then Do
    a=a+2
    Say 'Smile!'
End
```

反復 DO ループ

DO 命令に反復子 (repetitor) 句または条件 (conditional) 句、あるいはその両方を指定した場合、その命令のグループは反復 DO ループを形成します。命令は反復子句にしたがって処理されますが、オプションで、条件付き句によって変更することができます。[170 ページの『条件句 \(WHILE および UNTIL\)』](#)を参照してください。

• 単純反復ループ

単純反復ループは、反復句が反復回数を求める式になっている反復 DO ループです。

repetitor を省略して *conditional* を指定した場合、あるいは *repetitor* が FOREVER の場合には、命令のグループが名目上"永久的に"実行されます。つまり、条件が満たされるか、またはループを終了させる REXX 命令 (例えば、LEAVE) が実行されるまで、実行されます。

注: 条件付き句については、[170 ページの『条件句 \(WHILE および UNTIL\)』](#)を参照してください。

単純形式の反復ループでは、*exprr* が即座に計算され (計算結果は、正の整数またはゼロでなければなりません)、ループがその回数だけ実行されます。以下に例を示します。

```
/* This displays "Hello" five times */
Do 5
  say 'Hello'
end
```

コマンドと割り当てを区別する場合と同様、*exprr* の最初のトークンがシンボルであり、2 番目のトークンが = である (または = で始まる) 場合は、制御付きの形式の *repetitor* が必要です。

• 制御付き反復ループ

制御付き形式では、*name* (制御変数) を指定します。この制御変数には、命令リストの最初の実行の前に、初期値 (*expri* の結果を、それに 0 を加算したことにして形式設定したもの) が割り当てられます。次に、この変数は、命令リストの 2 回目以降の実行の前に (*exprb* の結果を加算して) 処理されます。

命令リストは、終了条件 (*exprt* の結果によって決まります) が満たされない間は、繰り返し実行されます。*exprb* が正または 0 ならば、*name* が *exprt* より大きくなったときにループは終了します。負であれば、*name* が *exprt* よりも小さくなったときにループは終了します。

expri、*exprt*、および *exprb* オプションの計算結果は、数値にならなければなりません。これらのオプションは、ループが開始される前および制御変数が初期値にセットされる前に、1 回だけ評価されます。*exprb* のデフォルト値は 1 です。*exprt* を省略した場合、他の何らかの条件によって終了させられない限り、ループは無期限に実行されます。以下に例を示します。

```
Do I=3 to -2 by -1      /* Displays: */
  say i                 /*      3      */
end                     /*      2      */
                        /*      1      */
                        /*      0      */
                        /*     -1      */
                        /*     -2      */
```

数値は、以下のように、整数である必要はありません。

```
I=0.3
Do Y=I to I+4 by 0.7    /* Displays: */
  say Y                 /*      0.3    */
end                     /*      1.0    */
                        /*      1.7    */
                        /*      2.4    */
```

```

/*      3.1      */
/*      3.8      */

```

ループ内で制御変数を変更することはできますが、変更した場合、ループの反復に影響を与える可能性があります。制御変数の値の変更は、特定の状況下では適切な場合もありますが、通常、好ましいプログラミング手法とは見なされません。

終了条件は、反復が開始されるたびにテストされます (2 回目以降の反復時には、制御変数の処理後にもテストされます)。したがって、終了条件が即座に合致した場合には、命令のグループ全体をスキップすることになります。制御変数は、名前で参照されることに注意してください。例えば、複合名 A.I を制御変数として使用した場合、ループ内で I を変更すると、制御変数の内容も変更されます。

制御付きループの実行には、FOR 句によってさらに制限を加えることができます。この場合には、*exprf* を指定する必要があります、その計算結果は、正の整数またはゼロでなければなりません。このオプションは、単純な反復ループの反復回数と同じ働きをし、その他の条件でループが終了しないときに、ループの反復回数の限度を設定します。TO 式や BY 式と同様に、反復回数は、初めて DO 命令が実行され、制御変数に初期値が入れられる前に、1 回だけ評価されます。TO 条件と同じように、FOR 条件も、反復が開始されるたびにチェックされます。以下に例を示します。

```

Do Y=0.3 to 4.3 by 0.7 for 3 /* Displays: */
  say Y                      /*      0.3      */
end                          /*      1.0      */
                           /*      1.7      */

```

制御付きループでは、制御変数を記述する *name* を、END 文節に指定できます。この *name* は、大文字か小文字以外のすべての点で、DO 文節内の *name* と一致する必要があります (複合変数の置換は行われませんので、ご注意ください)。一致していない場合には、構文エラーになります。このようにしておけば、最小のオーバーヘッドで、ループのネストを自動的にチェックすることができます。以下に例を示します。

```

Do K=1 to 10
  ...
  ...
End k /* Checks that this is the END for K loop */

```

注: NUMERIC 設定は、制御変数の連続値に影響する可能性があります。制御変数のステップ実行の計算に、REXX の算術規則が適用されるためです。

条件句 (WHILE および UNTIL)

条件 (conditional) 句は、反復 DO ループの繰り返しを変更することができます。これによって、ループが終了することもあります。これは、どのような形式の *repetitor* (なし、FOREVER、単純、または制御付き) の後にも続けることができます。WHILE または UNTIL を指定すると、ループするたびに、すべての変数の最新値を使用して、それぞれ *exprw* または *expru* が計算されます (計算結果は、0 または 1 にならなければなりません)。このとき、*exprw* の計算結果が 0 あるいは *expru* の計算結果が 1 になると、ループは終了します。

WHILE ループの場合の条件は、命令グループの先頭でチェックされます。UNTIL ループの場合、条件は命令グループの最後 (制御変数が加算される前) でチェックされます。例えば、次のようになります。

```

Do I=1 to 10 by 2 until i>6
  say i
end
/* Displays: "1" "3" "5" "7" */

```

注: LEAVE 命令または ITERATE 命令を使用して、反復ループの実行を変更することもできます。

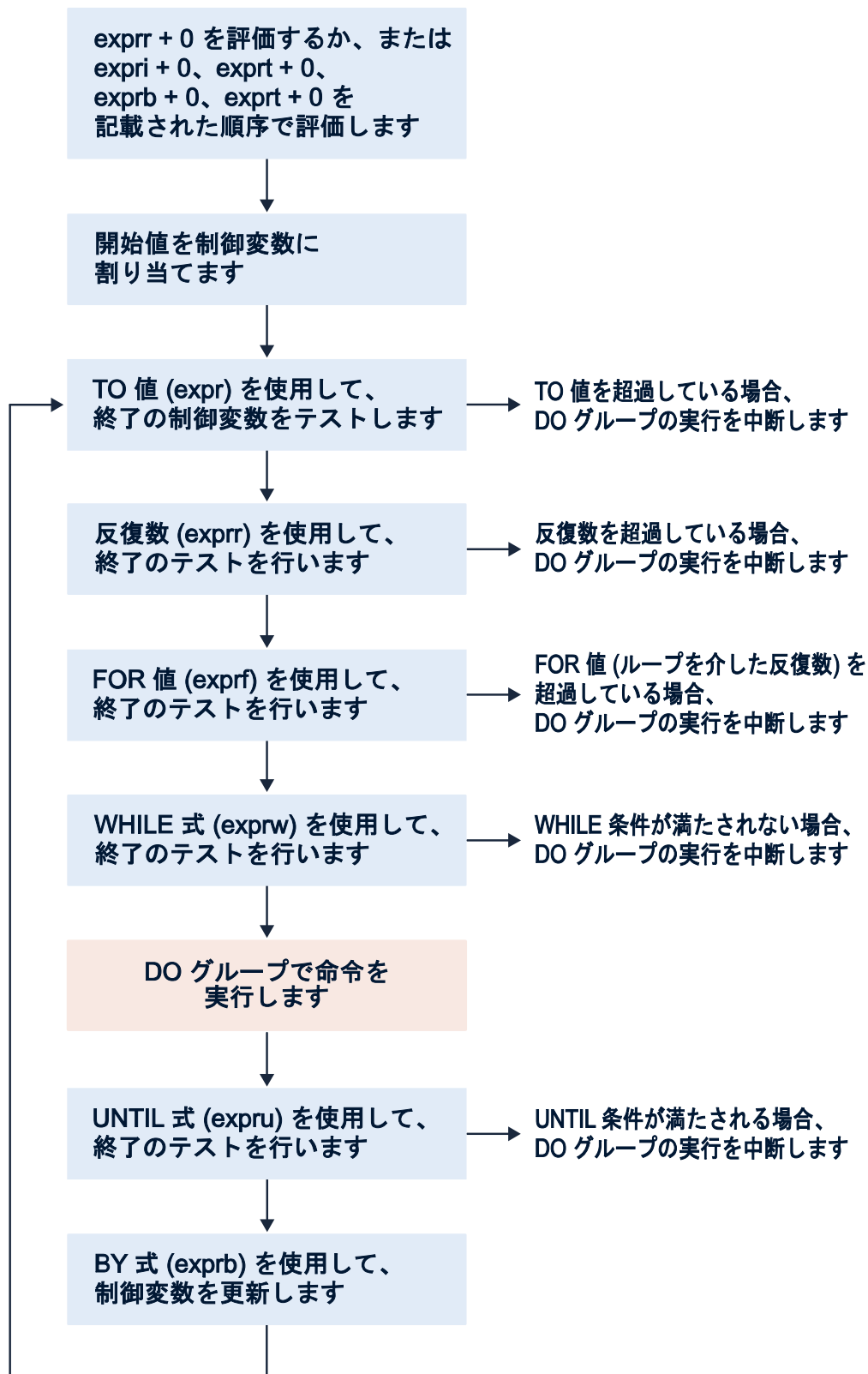
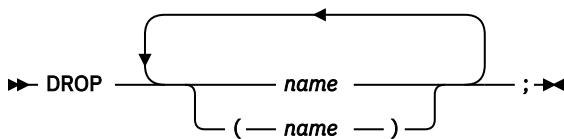


図 47. DO ループの概念

DROP

DROP は、変数を割り当て解除します。つまり、変数を、元の未初期化状態に復元します。



name は、括弧で囲まれていない場合、ユーザーがドロップする変数を識別するため、1つ以上の空白またはコメントで他の *name* から区切った、有効な変数名であるシンボルでなければなりません。

括弧で1つの *name* を囲むと、その値がドロップする変数の補助リストとして使用されます。(空白は、括弧の内側でも外側でも必須ではありませんが、追加することは可能です。)この補助リストは、括弧が使用できないこと以外は、元のリストと同じ規則(すなわち、有効な変数名を空白で区切ったリストであること)に従う必要があります。

変数は、左から右へ順番にドロップされていきます。1つの名前を複数回指定しても、また、認識されていない変数をドロップしても、エラーにはなりません。公開された変数が指定される ([181 ページの『PROCEDURE』](#)を参照) と、前の世代の変数はドロップされます。

例

```
j=4
Drop a z.3 z.j
/* Drops the variables: A, Z.3, and Z.4          */
/* so that reference to them returns their names. */
```

次の例では、変数名を括弧に入れて、補助リストとして使用しています。

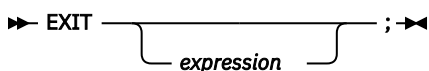
```
mylist='c d e'
drop (mylist) f
/* Drops the variables C, D, E, and F          */
/* Does not drop MYLIST                       */
```

語幹(すなわち、最後の文字として、1つのピリオドのみを含むシンボル)を指定すると、その語幹で始まるすべての変数をドロップします。

```
Drop z.
/* Drops all variables with names starting with Z. */
```

EXIT

EXIT は、プログラムを無条件に中止します。



オプションで、EXIT は呼び出し元に文字ストリングを戻します。プログラムは、内部ルーチンが現在実行中であっても、即時に停止されます。内部ルーチンがアクティブでない場合、RETURN ([184 ページの『RETURN』](#)を参照) と EXIT は、実行されているプログラムに対する効果は同じです。

式 *expression* を指定した場合、その式は評価され、プログラムが停止したときに、その評価結果のストリングが呼び出し元に戻されます。例えば、次のようになります。

```
j=3
Exit j*4
/* Would exit with the string '12' */
```

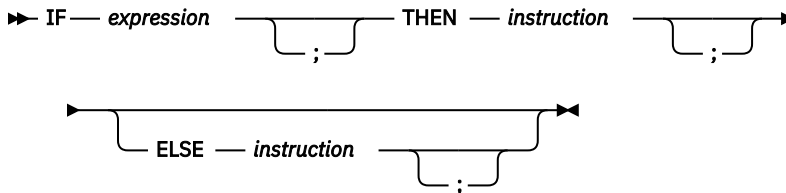
expression を指定しない場合には、呼び出し元にデータは戻されません。プログラムが外部関数として呼び出された場合は、直ちにエラーとして検出されるか (RETURN が使用された場合)、または呼び出し元に戻る際にエラーとして検出されます (EXIT が使用された場合)。

プログラムの「終わりからの離脱」は、プログラム全体を終了し、結果のストリングは戻さないという点で、常に、EXIT 命令と同等です。

注：プログラムがコマンド・インターフェースを介して呼び出されていた場合は、戻された値を、基礎オペレーティング・システムで受け入れ可能な戻りコードにするための変換が試みられます。変換に失敗しても、基礎となるオペレーティング・システムが原因の失敗と見なされるため、SIGNAL ON SYNTAX によるトラップの対象にはなりません。戻されたストリングは、値が汎用レジスターに収まる整数 (つまり、 -2^{31} から $2^{31}-1$ の範囲) である必要があります。

IF

IF は、*expression* の評価に応じて、1 つの命令または命令のグループを条件付きで処理します。*expression* は、評価された結果が 0 または 1 である必要があります。



THEN の後の命令は、結果が 1 (真) の場合のみ処理されます。ELSE を指定した場合、ELSE の後の命令は、評価の結果が 0 (偽) である場合にのみ処理されます。

例

```
if answer='YES' then say 'OK!'
                    else say 'Why not?'
```

THEN 部分の最後の文節と同じ行に ELSE 文節を入れる場合には、その ELSE の前にセミコロンを書く必要があります。

```
if answer='YES' then say 'OK!'; else say 'Why not?'
```

ELSE は、同じレベルにある、最も近い IF と結合します。下記に示す例のように NOP 命令を使用すると、IF 構造をネストしたときのエラーおよび混同を防ぐことができます。

```
If answer = 'YES' Then
  If name = 'FRED' Then
    say 'OK, Fred.'
  Else
    nop
Else
  say 'Why not?'
```

注：

1. *instruction* には、任意の割り当て、コマンド、またはキーワード命令を使用することができ、これには、DO、SELECT、または IF 命令そのものなど、より複雑な構成も含まれます。ヌル文節は命令ではないので、THEN または ELSE の後にセミコロン (またはラベル) を余分に入れても、(PL/I の場合のように) ダミー命令を入れることと同じにはなりません。この目的のためには、NOP 命令が提供されています。
2. THEN というシンボルを *expression* の中で使用することはできません。これは、キーワード THEN は、文節を開始する必要があるという点で、処理方法が異なるためです。これにより、IF 文節上の式は、; を必要とせずに、THEN で終了できます。このようになっていないと、他のコンピューター言語に慣れている人は大きな困難に直面することになります。

INTERPRET

INTERPRET は、*expression* を評価することで動的に作成された命令を処理します。

►► INTERPRET — *expression* — ;►►

expression の計算とその後の処理 (解釈) は、 スtring が計算されて (DO; と END; で囲まれて) 1 つの行としてプログラムに 挿入される場合と同様に行われます。

(INTERPRET 命令を含む) 任意の命令が使えますが、DO...END や SELECT...END などの構造は 完全な構造でなければなりません。例えば、解釈される命令の String に LEAVE 命令または ITERATE 命令 (反復 DO ループ内でのみ 有効) を含めることはできません。ただし、反復 DO...END 構造全体も含まれている場合は除きます。

セミコロンが指定されていない場合は、式の実行時に、その式の終わりにセミコロンが暗黙指定されます。

例

1.

```
data='FRED'
interpret data '= 4'
/* Builds the string  "FRED = 4" and          */
/* Processes:  FRED = 4;                      */
/* Thus the variable FRED is set to "4"      */
```
2.

```
data='do 3; say "Hello there!"; end'
interpret data          /* Displays:          */
                        /* Hello there!      */
                        /* Hello there!      */
                        /* Hello there!      */
```

注:

1. ラベル文節は、解釈される文字 String 内では使用できません。
2. INTERPRET 命令の概念に不慣れなため、理解できない結果が出た場合には、TRACE R または TRACE I をセットして INTERPRET 命令を実行すると、理解する手助けとなります。以下に例を示します。

```
/* Here is a small REXX program. */
Trace Int
name='Kitty'
indirect='name'
interpret 'say "Hello" indirect'!"!"
```

これを実行すると、以下のようなトレースになります。

```
kitty
3 *-* name='Kitty'
  >L>  "Kitty"
4 *-* indirect='name'
  >L>  "name"
5 *-* interpret 'say "Hello" indirect'!"!"
  >L>  "say "Hello""
  >V>  "name"
  >O>  "say "Hello" name"
  >L>  "!"!"
  >O>  "say "Hello" name"!"!"
  *-* say "Hello" name"!"
  >L>  "Hello"
  >V>  "Kitty"
  >O>  "Hello Kitty"
  >L>  "!"
  >O>  "Hello Kitty!"
Hello Kitty!
```

この例の 3 行目および 4 行目は、5 行目で使用する変数をセットしています。この後 5 行目の実行は、2 段階で進みます。まず初めに、解釈すべき String が、リテラル・String、変数 (INDIRECT)、および 別のリテラル・String を使用して、作成されます。次に、作成された純文字 String が、元のプログラムの一部であるかのように解釈されます。この文字 String は新しい文節であるため、新しい文節としてトレースされてから (5 行目の下にある 2 番目の *-* トレース・フラグ)、実行されま

す。再びリテラル・ストリングが変数 (NAME) と別のリテラル の値に連結され、次に、最終結果 (Hello Kitty!) が表示されます。

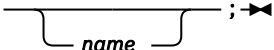
3. VALUE 関数 (219 ページの『VALUE』を参照) は、INTERPRET 命令の 代わりに、多くの目的に使用できます。したがって、最後の例の 5 行目を、次の行に置き換えることもできます。

```
say "Hello" value(indirect)"!"
```

INTERPRET が必要になるのは、通常、複数のステートメントを一緒に解釈したり、式を動的に計算するといった、特殊な場合だけです。

ITERATE

ITERATE は、反復 DO ループ (つまり、単純 DO を使用していないすべての DO 構造) 内の流れを変更します。

➡ ITERATE  ; ➡

END 文節が検出された場合と同様に、命令のグループの実行が停止し、制御権が DO 命令に渡されます。制御変数 (存在する場合) は、通常どおり、増分処理されてからテストされ、DO 命令でループが終了されない限り、命令のグループが再び実行されます。

name は、定数の値をとるシンボルです。 *name* が指定されていない場合、ITERATE は、最も内側のアクティブな反復ループに進みます。 *name* を指定する場合には、現在アクティブなループ (最も内側のループの場合もあります) の制御変数の名前を指定する必要があり、これが、対象のループになります。反復させる対象として選択されたループの内側にアクティブ・ループがあると、それらはすべて、(LEAVE 命令による終了と同様に) 終了します。

例

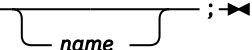
```
do i=1 to 4
  if i=2 then iterate
  say i
end
/* Displays the numbers:  "1" "3" "4" */
```

注:

1. *name* を指定する場合は、大/小文字の違いという側面を除き、DO 文節内の制御変数を表すシンボルと完全に一致させる必要があります。比較の際には、複合変数の置換は行われません。
2. 現在処理中である場合に、そのループはアクティブとなります。ループの実行中にサブルーチンが呼び出された場合 (または INTERPRET 命令が処理された場合)、そのループは、サブルーチンが戻るか、または INTERPRET 命令が完了するまで、非アクティブになります。ITERATE を使用して、非アクティブなループに進むことはできません。
3. 複数のアクティブなループが同じ制御変数を使用している場合、ITERATE は最も内側のループを選択します。

LEAVE

LEAVE により、1 つ以上の反復 DO ループ (つまり、単純 DO 以外のすべての DO 構成) が直ちに終了します。

➡ LEAVE  ; ➡

END 文節が検出され、終了条件が満足された場合と同様に、命令のグループの処理が終了し、END 文節の次の命令に制御が渡されます。ただし、終了時には、制御変数 (存在する場合) に、LEAVE 命令が処理されたときに入っていた値が入ります。

name は、定数の値をとるシンボルです。 *name* が指定されない場合、LEAVE は、最も内側のアクティブな反復ループを終了させます。 *name* を指定する場合は、現在アクティブなループ (最も内側のものである可能性があります) の制御変数の名前でなければならず、指定により、このループが終了します。 その後、制御は、選択されたループの DO 文節に一致する、END に続く文節に渡されます。

例

```
do i=1 to 5
  say i
  if i=3 then leave
end
/* Displays the numbers:  "1" "2" "3" */
```

注:

1. *name* を指定する場合は、大/小文字の違いという側面を除き、DO 文節内の制御変数を表すシンボルと完全に一致させる必要があります。 比較の際には、複合変数の置換は行われません。
2. 現在処理中である場合に、そのループはアクティブとなります。 ループの実行中にサブルーチンが呼び出された場合 (または INTERPRET 命令が処理された場合)、そのループは、サブルーチンが戻るか、または INTERPRET 命令が完了するまで、非アクティブになります。 LEAVE でもって非アクティブなループを終了させることはできません。
3. 複数のアクティブなループが同じ制御変数を使用している場合、LEAVE は最も内側のループを選択します。

NOP

NOP は効力を持たないダミー命令です。 この命令は、THEN 文節または ELSE 文節のターゲットとして便利です。

➡ NOP — ;➡

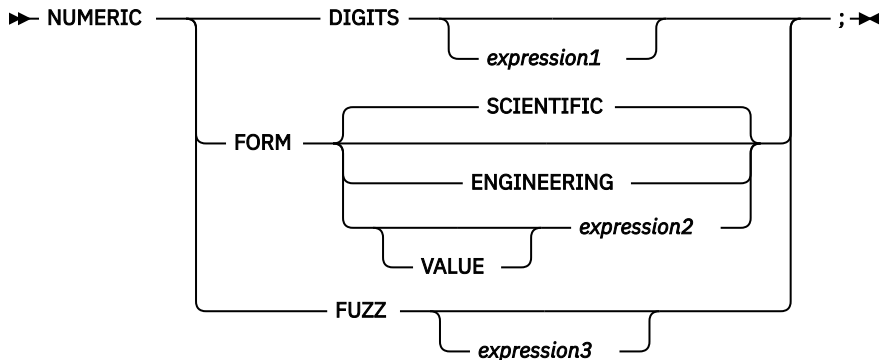
例

```
Select
  when a=c then nop          /* Do nothing */
  when a>c then say 'A > C'
  otherwise      say 'A < C'
end
```

注: NOP の代わりに余分にセミコロンを入れても、単にヌル文節を挿入したことにしかならないので、そのセミコロンは無視されます。 2 番目の WHEN 文節は、THEN の後に続く最初の命令と見なされてしまうため、構文エラーとして処理されます。 ただし、NOP は実際に命令であるため、THEN 文節の有効なターゲットになります。

NUMERIC

NUMERIC は、プログラムが算術演算を行う方法を変更します。



この命令のオプションについては、[241 ページの『第 21 章 数値と算術演算』](#)および [248 ページの『エラー』](#)で説明されていますが、要約を以下に示します。

NUMERIC DIGITS

このオプションは、算術演算および算術組み込み関数の計算結果の精度を制御します。*expression1* を省略すると、デフォルトの精度は 9 桁になります。省略しない場合、*expression1* の結果は、正の整数でなければならず、現行の NUMERIC FUZZ 設定値よりも大きくなければなりません。

DIGITS の値には制限がありません (使用可能なストレージを除きます。詳しくは、[141 ページの『第 17 章 REXX の一般的な概念』](#)を参照) が、精度を高くすると、多くの処理時間を必要とする可能性があります。ことに注意してください。可能なかぎり、デフォルト値を使用することをお勧めします。

DIGITS 組み込み関数を使用して、現行の NUMERIC DIGITS 設定値を取り出すことができます。[205 ページの『DIGITS』](#)を参照してください。

NUMERIC FORM

このオプションは、算術演算および算術組み込み関数の結果に REXX が使用する指数表記の形式を制御します。これには、SCIENTIFIC (この場合、小数点の前に 1 桁だけの非ゼロの数字が付きます) または ENGINEERING (この場合、10 の累乗が常に 3 の倍数になります) のいずれかがあります。デフォルトは SCIENTIFIC です。サブキーワード SCIENTIFIC または ENGINEERING は、FORM に直接設定するか、VALUE の後の式 (*expression2*) の評価結果から取得されます。この場合の結果は、SCIENTIFIC または ENGINEERING のいずれかでなければなりません。*expression2* が記号またはリテラル・ストリングで始まっていない (すなわち、演算子文字や括弧などの特殊文字で始まっている) 場合は、VALUE サブキーワードを省略することができます。

FORM 組み込み関数を使用して、現行の NUMERIC FORM 設定値を取り出すことができます。[207 ページの『FORM』](#)を参照してください。

NUMERIC FUZZ

このオプションは、数値比較演算時に無視する、完全精度での桁数を制御します。[245 ページの『数値の比較』](#)を参照してください。*expression3* を省略すると、0 桁がデフォルトとなります。省略しない場合、*expression3* の結果は、0 または正の整数 (必要に応じて現行の NUMERIC DIGITS 設定値に従って丸められます) でなければならず、現行の NUMERIC DIGITS 設定値よりも小さくなければなりません。

NUMERIC FUZZ は、数値比較のたびに、NUMERIC DIGITS の値を NUMERIC FUZZ 値の分だけ一時的に減らします。比較時には、DIGITS の桁数から FUZZ の桁数を引いた桁数の精度のもとで 2 つの数値の減算を行い、その結果を 0 と比較します。

FUZZ 組み込み関数を使用して、現行の NUMERIC FUZZ 設定値を取り出すことができます。[209 ページの『FUZZ』](#)を参照してください。

注：この 3 つの数値設定は、内部サブルーチン、外部サブルーチン、および関数呼び出し全体で自動的に保管されます。詳しくは、[CALL 命令 \(165 ページの『CALL』 ページ\)](#)を参照してください。

OPTIONS

OPTIONS は、特殊な要求またはパラメーターを言語処理プログラムに渡します。これらは、例えば、言語処理プログラム・オプションであるか、または特殊文字セットを定義する場合があります。

►► OPTIONS — *expression* — ;◄◄

expression が評価されると、結果が、一度に 1 ワードずつ検査されます。言語処理プログラムは、ワードを大文字に変換します。言語処理プログラムはワードを認識すると、それらのワードに従って処理します。認識されないワードは無視され、別のプロセッサに対する命令であると見なされます。

言語処理プログラムは、以下のワードを認識します。

ETMODE

DBCS 文字を含むリテラル・ストリング、シンボル、およびコメントが有効な DBCS ストリングであるかどうかを検査することを指定します。このオプションを使用する場合は、これをプログラムの最初の命令にしなければなりません。

expression が、例えば `OPTIONS 'GETETMOD'()` という外部機能呼び出しであり、しかもプログラムに DBCS リテラル・ストリングが含まれている場合は、オプションが有効になる前にプログラム全体がスキャンされることがないように、関数の名前を引用符で囲んでください。内部関数呼び出しを使用して ETMODE をセットすることは、プログラム内の DBCS リテラル・ストリングを解釈する際にエラーの発生する可能性があるので、お勧めできません。

NOETMODE

DBCS 文字を含むリテラル・ストリング、シンボル、およびコメントが有効な DBCS ストリングであるかどうかを検査しないことを指定します。NOETMODE がデフォルトです。言語処理プログラムは、このオプションがプログラム内の最初の命令でない限り、このオプションを無視します。

EXMODE

命令、演算子、および関数が混合ストリング内の DBCS データを論理文字単位で扱うことを指定します。DBCS データの完全性が保たれます。

NOEXMODE

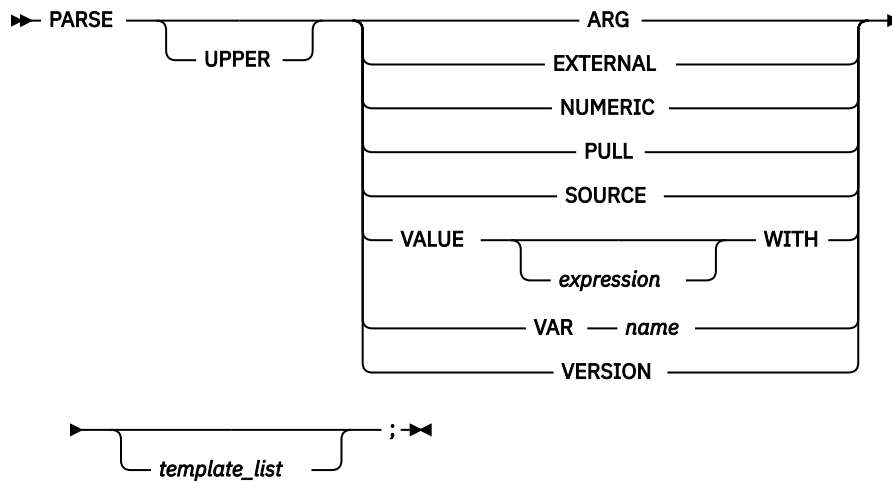
ストリング内のすべてのデータをバイト単位で処理することを指定します。DBCS 文字 (存在する場合) の完全性は、失われる可能性があります。NOEXMODE がデフォルトです。

注:

1. 言語プロセッサのスキャン手順の都合により、`OPTIONS 'ETMODE'` 命令は、リテラル・ストリング、シンボル、またはコメントに DBCS 文字を含むプログラムの最初の命令として指定する必要があります。`OPTIONS 'ETMODE'` を最初の命令として入れておらずに、それをプログラム内で後で使用した場合は、エラー・メッセージ `CICREX488E` を受け取ることになります。プログラムの最初の命令として置いた場合、後で同じ命令を使用しても無視されます。式にラベル探索を開始するようなものが含まれていると、ラベル探索の間にトークン化されるすべての文節は、現行の ETMODE の設定値でトークン化されます。このため、これがプログラム内の最初のステートメントであると、デフォルトは NOETMODE になります。
2. DBCS リテラルおよび DBCS コメントが含まれているプログラムを適切にスキャンするようにするためには、ETMODE、NOETMODE、EXMODE、および NOEXMODE という語は、`OPTIONS` 命令の中にリテラル・ストリングとして (つまり、引用符で囲んで) 入力してください。
3. EXMODE の設定値は、サブルーチンおよび機能呼び出しにまたがって、保管および復元が行われます。
4. DBCS 文字を 1 バイトの EBCDIC 文字と区別するために、DBCS ストリングは、シフト・アウト (SO) 文字とシフト・イン (SI) 文字で囲まれています。SO 文字および SI 文字の 16 進値は、それぞれ、`X'0E'` および `X'0F'` です。
5. `OPTIONS 'ETMODE'` を指定した場合、リテラル・ストリングの中の DBCS 文字は、リテラル・ストリング内の閉じ引用符の探索からは除外されます。
6. ETMODE、NOETMODE、EXMODE、および NOEXMODE の語は、結果内に複数回あってもかまいません。どれが有効になるかは、ETMODE と NOETMODE および EXMODE と NOEXMODE のペアで指定された中の、最後に有効なものによって判別されます。

PARSE

PARSE は、解析規則に従って、(各種ソースからの) データを 1 つまたは複数の変数に割り当てます。



227 ページの『第 20 章 構文解析』を参照してください。

`template_list` は、多くの場合は単一のテンプレートですが、複数のテンプレートをコンマで区切った形にすることもできます。テンプレートを指定した場合、各テンプレートは、ブランクまたはパターン (または、この両方) で他のテンプレートから区切られた、記号のリストとなります。

各テンプレートは、単一のソース・ストリングに適用されます。複数のテンプレートを指定しても構文エラーにはなりませんが、非ヌルのソース・ストリングを複数個指定できるのは、PARSE ARG バリエーションだけです。236 ページの『複数のストリングの構文解析』を参照してください。

テンプレートを指定しない場合、変数に値はセットされませんが、必要に応じて、解析用にデータを準備するための処置が取られます。PARSE EXTERNAL および PARSE PULL の場合には、キューからデータ・ストリングが解析されます。PARSE LINEIN (および、キューが空のときの PARSE PULL) の場合には、デフォルト入力ストリームから行が読み取られます。PARSE VALUE の場合には、*expression* が計算されます。PARSE VAR の場合は、指定した変数がアクセスされます。値がない場合には、NOVALUE 条件 (使用可能な場合) が生じます。

UPPER オプションを指定すると、解析されるデータが、まず、大文字に (つまり、小文字の a~z が大文字の A~Z に) 変換されます。指定しなければ、解析中に大文字変換は行われません。

各種の PARSE 命令ごとに、使用されるデータについて以下で説明します。

PARSE ARG

入力引数としてプログラムまたは内部ルーチンに渡された、1 つ以上のストリングを解析します。(詳細と例については、164 ページの『ARG』を参照してください。)

注: また、ARG 組み込み関数を使用して、REXX プログラムまたは内部ルーチンに対する引数ストリングを検索したり、検査したりすることができます (197 ページの『ARG (引数)』を参照)。

PARSE EXTERNAL

これは、REXX/CICS に用意されている非 SAA サブキーワードです。端末入力バッファにある次のストリングが解析されます。このキューには、外部非同期イベント (ユーザー・コンソール入力またはメッセージなど) の結果であるデータが入ります。このキューが空であれば、コンソール読み取りが行われます。ただし、この方法は、標準的なコンソール入力には使用しないでください。標準的なコンソール入力には PULL を使用の方が一般的であり、この方法は、プログラム・スタックを阻害できない特殊なアプリケーション (デバッグなど) に使用してください。

PARSE NUMERIC

これは、REXX/CICS に用意されている非 SAA サブキーワードです。現行の数値制御 (NUMERIC 命令で設定されているもの。) が使用可能です。これらの制御は、DIGITS FUZZ FORM の順になっています。例えば、次のようになります。

```
Parse Numeric Var1
```

この命令の後、Var1 は、9 0 SCIENTIFIC という値に等しくなります。176 ページの『NUMERIC』、205 ページの『DIGITS』、207 ページの『FORM』、および 209 ページの『FUZZ』の組み込み関数を参照してください。

PARSE PULL

外部データ・キューにある次のストリングを解析します。外部データ・キューが空の場合は、PARSE PULL は、デフォルト入力ストリーム (ユーザーの端末) から行の読み取りを行い、必要であれば、行が完了するまでプログラムは一時停止します。PUSH 命令および QUEUE 命令を使用すると、キューの先頭または末尾にデータを追加することができます。QUEUED 組み込み関数を使用して、キューに現在入っている行数を知ることができます。212 ページの『QUEUED』を参照してください。システム内の他のプログラムは、このキューを変更し、それを、REXX で作成されたプログラムとの通信の手段として使用することができます。PULL 命令 (182 ページの『PULL』) も参照してください。

注: PULL および PARSE PULL は、プログラム・スタックからの読み取りを行います。プログラム・スタックが空であれば、端末入力バッファから読み取ります。また、端末入力バッファも空であれば、コンソールから読み取ります。

PARSE SOURCE

実行中のプログラムのソースを記述しているデータを解析します。言語処理プログラムは、プログラムが実行している間固定である (変更されない) ストリングを戻します。PARSE SOURCE 命令は、以下のトークンが入っているソース・ストリングを戻します。

1. 文字 CICS。
2. COMMAND、FUNCTION、または SUBROUTINE のいずれかのストリング。プログラムがホスト・コマンドとして呼び出されたのか、式内の関数呼び出しから呼び出されたのか、CALL 命令によって呼び出されたのか、あるいはサーバー・プロセスとして呼び出されたのかによって異なります。
3. 大文字での EXEC の名前。EXEC の最初のロード元であったファイル (RFS)、あるいは MVS 区分データ・セットまたは DDNAME/メンバーの名前。形式は、次の 3 とおりです。
 - DDNAME - メンバー
 - RFS の完全修飾ファイル ID
 - データ・セット名 (メンバー)
4. 常に REXXCICS である初期 (デフォルト) ホスト・コマンド環境。
5. 特定の CICS 環境の ID。この場合は CICS-TS です。

PARSE VALUE

expression の計算結果であるデータを解析します。*expression* を指定しない場合、ヌル・ストリングが使用されます。WITH は、この文脈内ではサブキーワードなので、*expression* の中で記号として使用することはできません。

したがって、次の例は、現在時刻を取得して構成要素の部分ごとに分けています。

```
PARSE VALUE time() WITH hours ':' mins ':' secs
```

PARSE VAR *name*

変数の *name* の値を解析します。*name* は、変数名として有効な記号でなければなりません (つまり、ピリオドまたは数字で始まってはなりません)。変数 *name* は、テンプレートにも指定しない限り変更されません。例えば、次の命令は、*string* から最初のワードを除去し、そのワードを *word1* 変数に入れて、残りを *string* に再び割り当てます。

```
PARSE VAR string word1 string
```

同様に、次の例は、さらに `string` のデータを大文字に変換してから解析します。

```
PARSE UPPER VAR string word1 string
```

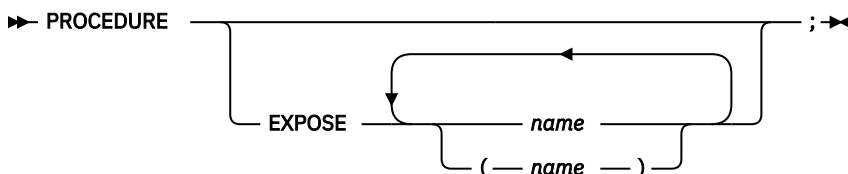
PARSE VERSION

言語処理プログラムの言語レベルおよび日付を記述している情報を解析します。この情報は5つのワードで構成され、それぞれブランクで区切られています。

1. REXX370 というストリング。370 実装環境を意味します。
2. 言語レベル記述 (例えば、3.48)。
3. 言語処理プログラムのリリース日付 (例えば、05 April 2000)。

PROCEDURE

PROCEDURE は、内部ルーチン (サブルーチンまたは関数) 内で、変数を保護して、後続の命令からは見えないようにします。



RETURN 命令が実行された後、元の変数環境が復元され、ルーチン内で使用された (公開されなかった) 変数はすべてドロップされます。(公開された変数は、PROCEDURE 命令が公開したルーチンの呼び出し元に属するものです。このルーチンが変数を参照または変更する際に、変数の元の (呼び出し元の) コピーが使用されます。) 内部ルーチンに PROCEDURE 命令を組み込む必要はありません。この場合、ルーチンが処理する変数は呼び出し元が「所有」しているものです。PROCEDURE 命令を使用する場合は、その命令が、CALL または関数呼び出しの後で最初に処理される命令でなければなりません。すなわち、ら別の後続く、最初の命令でなければなりません。

EXPOSE オプションを使用すると、*name* で指定された変数はすべて公開されます。この変数 (設定およびドロップを含む) への参照はいずれも、呼び出し元が所有する変数環境を参照します。したがって、既存の変数の値はアクセス可能であり、ルーチンからの RETURN でも、行われた変更はすべて持続されます。*name* は、括弧で囲まれていない場合、ユーザーが公開する変数を指示するため、1 つ以上のブランクで他の *name* から区切った、有効な変数名であるシンボルでなければなりません。

1つの *name* を括弧で囲むと、変数 *name* の公開後にすぐに *name* の値が変数補助リストとして使用されます。(ブランクは、括弧の内側でも外側でも必須ではありませんが、追加することは可能です。)この補助リストは、括弧が使用できないこと以外は、元のリストと同じ規則(すなわち、有効な変数名をブランクで区切ったリストであること)に従う必要があります。

変数は、左から右へ順番に提示されていきます。1つの名前を2回以上指定しても、呼び出し元が変数として使用していない名前を指定しても、エラーにはなりません。

主プログラム内の変数で、公開されないものはすべて、それまでと同様に保護されます。したがって、呼び出し元の変数のうちのいくつかを限定してアクセス可能にした上で、これらの変数を変更することができます(あるいは、これらの変数の他に新しい変数を追加することもできます)。ルーチンから戻った時点で、これらの変更はすべて呼び出し元にも分かる形で示されます。

例

```
/* This is the main REXX program */
j=1; z.1='a'
call toft
say j k m          /* Displays "1 7 M" */
exit

/* This is a subroutine */
toft: procedure expose j k z.j
      say j k z.j /* Displays "1 K a" */
```

```
k=7; m=3      /* Note: M is not exposed */
return
```

EXPOSE リスト内で Z.J が J より前に置かれた場合、J の呼び出し元の値がその時点で可視でないため、Z.1 は公開されません。

補助リスト内の変数も、左から右へ公開されます。

```
/* This is the main REXX program */
j=1;k=6;m=9
a='j k m'
call test
exit

/* This is a subroutine */
test: procedure expose (a) /* Exposes A, J, K, and M */
    say a j k m           /* Displays "j k m 1 6 9" */
    return
```

補助リストを使用すれば、より簡単に多くの変数を一度に公開できるようになり、また、VALUE 組み込み関数を用いれば、動的に指定した変数の処理もできるようになります。

```
/* This is the main REXX program */
c=11; d=12; e=13
Showlist='c d' /* but not E */
call Playvars
say c d e f    /* Displays "11 New 13 9" */
exit

/* This is a subroutine */
Playvars: procedure expose (showlist) f
    say word(showlist,2) /* Displays "d" */
    say value(word(showlist,2),'New') /* Displays "12" and sets new value */
    say value(word(showlist,2)) /* Displays "New" */
    e=8 /* E is not exposed */
    f=9 /* F was explicitly exposed */
    return
```

語幹を *name* として指定すると、この語幹の他に、その語幹で名前が始まるすべての複合変数が提示されます。155 ページの『語幹』を参照してください。

```
/* This is the main REXX program */
a.=11; i=13; j=15
i = i + 1
C.5 = 'FRED'
call lucky7
say a. a.1 i j c. c.5
say 'You should see 11 7 14 15 C. FRED'
exit
lucky7:Procedure Expose i j a. c.
/* This exposes I, J, and all variables whose */
/* names start with A. or C. */
A.1='7' /* This sets A.1 in the caller's */
        /* environment, even if it did not */
        /* previously exist. */
return
```

変数が確実にすべての中間 PROCEDURE 命令に組み込まれるようにすることにより、複数の世代のルーチンに公開することができます。


ルーチンの呼び出し方法に関する詳細と例については、CALL 命令 (165 ページの『CALL』) および関数の説明 (193 ページの『第 19 章 関数』) を参照してください。

PULL

PULL は、プログラム・スタックからストリングを 1 つ読み取ります。プログラム・スタックが空の場合、PULL は、現行の端末入力装置から 1 行読み取ろうとします。

```
➡ PULL _____ ; ➡
      |         |
      | template_list |
      |         |
```

PULL は、次の命令の短縮形です。

➡ PARSE UPPER PULL  ; ➡

キューの先頭に現在あるものが、1つのストリングとして読み取られます。*template_list* を指定しない場合は、それ以上のアクションは実行されません(つまり、ストリングは事実上破棄されます)。*template_list* を指定する場合は、通常テンプレートを1つ指定します。すなわち、ブランクまたはパターン(またはその両方)で分けた記号のリストです。*(template_list* には、複数のテンプレートをコンマで区切って指定することもできますが、PULL が解析するソース・ストリングは1つだけです。したがって、複数のテンプレートをコンマで区切って指定した場合は、最初のテンプレート以外のテンプレート内の変数にはヌル・ストリングが割り当てられます。)ストリングは、大文字に変換(つまり、小文字 a から z が大文字 A から Z に変換)されてから、構文解析に関するセクション(227 ページの『第 20 章 構文解析』)で説明している規則に従って変数に解析されます。大文字変換を望まない場合には、PARSE PULL 命令を使用してください。

注:

1. 外部データ・キューの REXX/CICS 実行環境は、このプログラム・スタックです。言語処理プログラムは、プログラム・スタックから1行読み取ります。プログラム・スタックが空の場合は、端末読み取りが行われます。ユーザー用のプログラム・スタックは \SYSTEM\userid*PROGSTACK* という名前の RLS キューに入っています。
2. PULL により端末から読み取りが行われた場合、PULL コマンドの完了時に 変数 PULLKEY が設定されます。これには、PULL コマンドへの応答で押された AID キーの名前(例えば、ENTER、PFKEY 1、PAKEY 1、MSR、PEN、または CLEAR)が含まれています。

指定されるキューについては、REXX List System の LPULL コマンド 301 ページの『LPULL』を参照してください。

例

```
Say 'Do you want to erase the file? Answer Yes or No:'  
Pull answer .  
if answer='NO' then say 'The file will not be erased.'
```

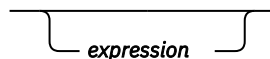
この例では、ユーザーが入力した最初のワードを区別するために、テンプレートでダミーのプレースホルダーである ピリオド(.) を使用しています。

外部データ・キューが空の場合は、コンソール読み取りが出され、プログラムは、必要であれば、1行が完了するまで一時停止します。

QUEUED 組み込み関数(212 ページの『QUEUED』を参照)は、プログラム・スタックに現在入っている行数を戻します。

PUSH

PUSH は、式 *expression* の評価結果であるストリングを、LIFO (後入れ先出し) で外部データ・キューにスタックします。

➡ PUSH  ; ➡

expression を指定しなければ、ヌル・ストリングがスタックされます。

注: 外部データ・キューの REXX/CICS 実行環境は、このプログラム・スタックです。言語処理プログラムは、プログラム・スタックから1行読み取ります。プログラム・スタックが空の場合は、端末読み取りが行われます。ユーザー用のプログラム・スタックは \SYSTEM\userid*PROGSTACK* という名前の RLS キューに入っています。

指定されるキューについては、REXX List System の LPUSH コマンドの 302 ページの『LPUSH』を参照してください。

例

```
a='Fred'
push      /* Puts a null line onto the queue */
push a 2  /* Puts "Fred 2"      onto the queue */
```

QUEUED 組み込み関数 (212 ページの『[QUEUED](#)』で説明) は、外部データ・キューに現在入っている行数を戻します。

QUEUE

QUEUE は、*expression* の結果であるストリングを、外部データ・キューの末尾に追加します。つまり、FIFO (先入れ先出し) で追加されます。

➡ QUEUE _____ ; ➡
 └─ *expression* ─┘

expression を指定しなければ、ヌル・ストリングがキューに入ります。

注：外部データ・キューの REXX/CICS 実行環境は、このプログラム・スタックです。言語処理プログラムは、プログラム・スタックから 1 行読み取ります。プログラム・スタックが空の場合は、端末読み取りが行われます。ユーザー用のプログラム・スタックは \SYSTEM\userid*PROGSTACK* という名前の RLS キューに入っています。

例

```
a='Toft'
queue a 2 /* Enqueues "Toft 2" */
queue     /* Enqueues a null line behind the last */
```

QUEUED 組み込み関数 (212 ページの『[QUEUED](#)』で説明) は、外部データ・キューに現在入っている行数を戻します。

RETURN

RETURN は、REXX プログラムまたは内部ルーチンからその呼び出し点へ制御を (可能であれば結果も) 戻します。

➡ RETURN _____ ; ➡
 └─ *expression* ─┘

内部ルーチン (サブルーチンまたは関数) がアクティブでない場合、実行されているプログラムに対する RETURN と EXIT の効果は同じです。172 ページの『[EXIT](#)』を参照してください。

サブルーチンを実行していた場合は (CALL 命令を参照してください)、*expression* (指定した場合) が計算されて、制御が呼び出し元に戻り、REXX 特殊変数である RESULT に *expression* の値がセットされます。*expression* を省略すると、RESULT 特殊変数がドロップされます (初期値は与えられません)。CALL の時点で保管されたさまざまな設定値 (トレースやアドレスなど) も復元されます。165 ページの『[CALL](#)』を参照してください。

関数を処理していた場合も、実行されるアクションは同じです。ただし、RETURN 命令に *expression* を指定しなければなりません。この *expression* の結果は、元の式の関数が呼び出されたところで使用されます。詳しくは、193 ページの『[第 19 章 関数](#)』の関数の説明を参照してください。

ルーチン (サブルーチンまたは内部関数) 内で PROCEDURE 命令を実行していた場合、*expression* が計算された後、しかも結果が使用されるかまたは RESULT に割り当てられる前に、現行世代のすべての変数がドロップされます (そして、前の世代の変数が公開されます)。

SAY

SAY は、デフォルト出力ストリーム (端末) に対して 1 行書き出して、それをユーザーに表示します。

► SAY — *expression* ; ►

expression の結果の長さに制限はありません。 *expression* を省略すると、ヌル・ストリングが書き出されます。

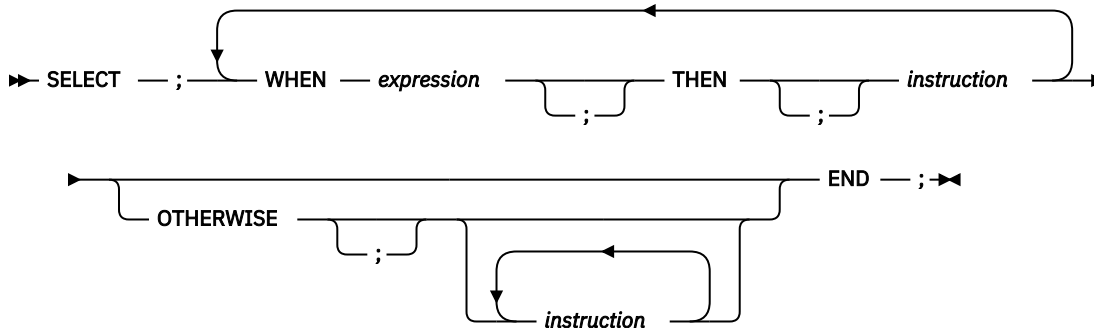
SET TERMOUT コマンドを使用して、SAY 出力をリダイレクトすることができます。

例

```
data=100
Say data 'divided by 4 =>' data/4
/* Displays: "100 divided by 4 => 25" */
```

SELECT

SELECT は、いくつかの択一命令から 1 つを条件付きで実行します。



WHEN の後の各 *expression* は順に評価され、結果は 0 または 1 でなければなりません。結果が 1 である場合、対応する THEN の後の命令 (IF、DO、または SELECT などの複合命令である場合があります) が処理され、その後、制御が END に渡されます。結果が 0 であれば、制御は次の WHEN 文節に移ります。

1 と評価される WHEN 式がない場合、制御は、OTHERWISE の後の命令 (存在する場合) に渡されます。このような状態の場合、OTHERWISE がないとエラーになります (ただし、OTHERWISE の後の命令のリストは省略できます)。

例

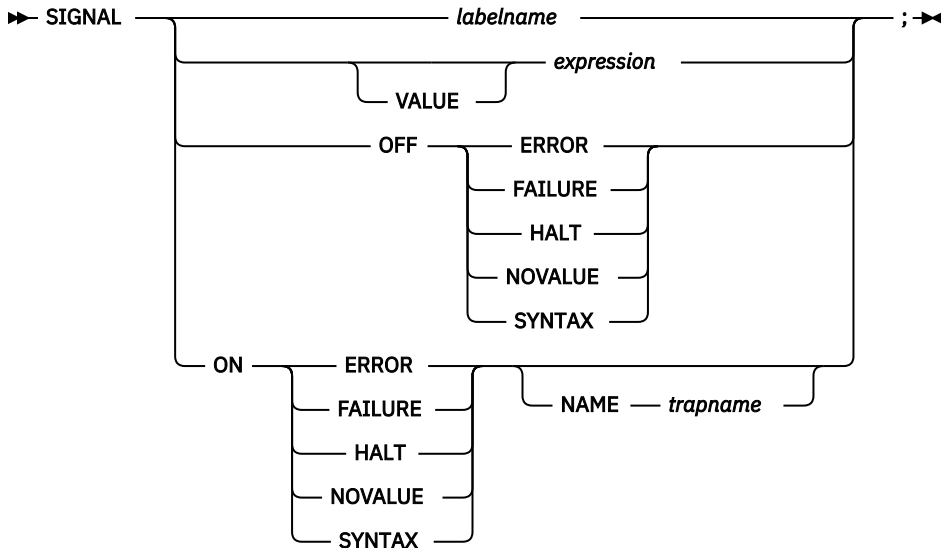
```
balance=100
check=50
balance = balance - check
Select
  when balance > 0 then
    say 'Congratulations! You still have' balance 'dollars left.'
  when balance = 0 then do
    say 'Warning, Balance is now zero! STOP all spending.'
    say "You cut it close this month! Hope you do not have any"
    say "checks left outstanding."
  end
  Otherwise
    say "You have just overdrawn your account."
    say "Your balance now shows" balance "dollars."
    say "Oops! Hope the bank does not close your account."
end /* Select */
```

注:

1. *instruction* は、任意の割り当て、コマンド、またはキーワード命令にすることができます。これには、DO、IF、または SELECT の命令そのものなど、より複雑な構成のものも含まれます。
2. ヌル文節は命令ではないため、THEN 文節の後に余分なセミコロン (またはラベル) を付けても、ダミー命令を置いたのと同じことにはなりません。この目的のためには、NOP 命令が提供されています。
3. THEN というシンボルを *expression* の中で使用することはできません。これは、キーワード THEN は、文節を開始する必要があるという点で、処理方法が異なるためです。これにより、WHEN 文節上の式は、; (区切り文字) を必要とせずに THEN で終了できます。

SIGNAL

SIGNAL を使用すると、制御の流れの変則的な変更を可能にしたり (*labelname* または *VALUE expression* を指定)、特定の条件のトラッピングを制御したりできます (ON または OFF を指定)。



トラッピングを制御するには、OFF または ON のほかに、トラップしたい条件を指定します。OFF は、指定された条件トラップをオフにします。ON は、指定された条件トラップをオンにします。249 ページの『第 22 章 条件および条件トラップ』を参照してください。

制御の流れを変更する場合、ラベル名には、*labelname* を指定するか、あるいは VALUE の後に *expression* を指定してその計算結果を用います。指定する *labelname* は、定数と見なされるリテラル・ストリングまたはシンボルでなければなりません。 *labelname* にシンボルを使用した場合、探索は、英字の大/小文字に関係なく行われます。リテラル・ストリングを使用する場合には、文字は英大文字である必要があります。これは、言語処理プログラムは、プログラムでどのように入力されているかに関係なく、すべてのラベルを英大文字に変換するからです。同様に、SIGNAL VALUE の場合、*expression* は大文字のストリングに評価される必要があります。そうでないと、言語処理プログラムはそのラベルを見つけれられません。 *expression* がシンボルまたはリテラル・ストリングで始まっていない (つまり、演算子文字や括弧などの特殊文字で始まっている) 場合には、サブキーワード VALUE を省略できます。現行ルーチン内で保留されているすべてのアクティブな DO、IF、SELECT、および INTERPRET の各命令は、この後、終了します (つまり、再開できません)。次に、プログラム内のラベルのうち、指定された名前と一致する最初のラベル (検索がプログラムの先頭から開始されたものとして) に制御が渡されます。

例

```
Signal fred; /* Transfer control to label FRED below */
....
....
Fred: say 'Hi!'
```

検索は、実質的にプログラムの先頭から開始されるため、プログラム内でラベルが重複している場合、制御は常に最初のオカレンスに渡されます。

指定されたラベルに制御が渡ると、**SIGNAL** 命令の行番号が特殊変数 **SIGL** に割り当てられます。ラベルへの制御権の移動元を判別するために **SIGL** を使用できるため、これは、デバッグに役立ちます。

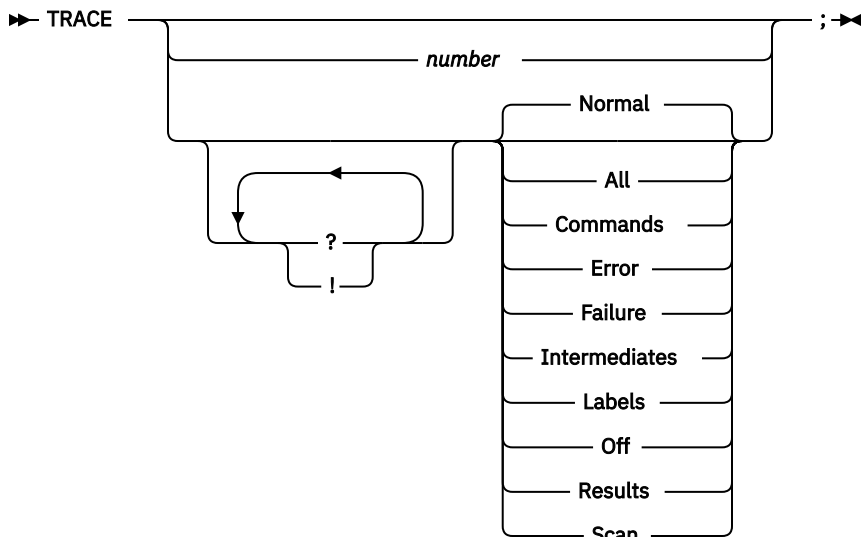
SIGNAL VALUE の使用

SIGNAL 命令の **VALUE** 形式は、実行時に名前が決定されるラベルにブランチできます。これにより、内部ルーチンに対するマルチウェイ呼び出し (または関数呼び出し) は安全に行われます。呼び出しルーチン内の **DO** ループなどはいずれも、呼び出しメカニズムによる終了から保護されるためです。例えば、次のようになります。

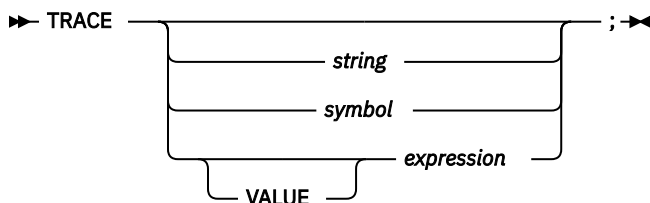
```
fred='PETE'  
call multiway fred, 7  
....  
....  
exit  
Multiway: procedure  
  arg label .          /* One word, uppercase          */  
                        /* Can add checks for valid labels here */  
  signal value label    /* Transfer control to wherever      */  
  ....  
Pete: say arg(1) '!' arg(2) /* Displays: "PETE ! 7"          */  
  return
```

TRACE

TRACE は、REXX プログラムの処理中のトレース・アクション (つまり、どれだけを表示するか) を制御します。



または、以下を参照してください。



トレースは、文節が処理されるにつれて文節の記述を作成することにより、プログラム内の一部またはすべての文節を記述します。TRACE は、主にデバッグのために使用されます。TRACE は、通常、対話式のデバッグ中に手作業で入力するため、その構文は、他の REXX 命令より簡潔です。(これは、プログラムの実行中にユーザーが言語処理プログラムと対話できるトレースの形式の 1 つです。) 使用にあたっては、キーを打つ手間が少ないのが特に便利です。

number を指定する場合は、整数を指定しなければなりません。

string または *expression* は、計算結果が以下のものになります。

- 数字オプション
- 有効な接頭部オプションまたは英字 (ワード) オプションの 1 つ (後述します)
- スル

symbol は、定数として扱われるため、下記のいずれかになります。

- 数字オプション
- 有効な接頭部オプションまたは英字 (ワード) オプションの 1 つ (後述します)

TRACE の後にあるオプションまたは *expression* の計算結果によって、トレース動作が決まります。
expression が記号またはリテラル・ストリングで始まっていない (すなわち、演算子や括弧などの特殊文字で始まっている) 場合は、VALUE サブキーワードを省略することができます。

英字 (ワード) オプション

ワードは、全体を入力することもできますが、大文字で強調表示された文字だけが必要であり、後続の文字はすべて無視されます。これが、英字オプションと呼ばれる理由です。ERROR および FAILURE の定義については、[156 ページの『外部環境に対するコマンド』](#)を参照してください。

TRACE アクションは、以下の英字オプションに対応しています。

All (すべて)

すべての文節を実行前にトレース (すなわち、表示) します。

コマンド

すべてのコマンドを実行前にトレースします。コマンドの結果がエラーまたは障害であった場合、トレースは、コマンドからの戻りコードも表示します。

Error

実行後、エラーまたは障害を起こしたすべてのコマンドを、そのコマンドの戻りコードと一緒にトレースします。

失敗

実行後、障害を起こしたすべてのコマンドを、そのコマンドの戻りコードと一緒にトレースします。これは Normal オプションと同じです。

Intermediates

すべての文節を実行前にトレースします。式の計算中の中間結果および置換された名前もトレースされます。

ラベル

実行中に渡されたラベルのみをトレースします。ラベルごとに言語処理プログラムが一時停止する場合には、このオプションとデバッグ・モードを併用すると特に便利です。これはまた、すべての内部サブルーチン呼び出しおよび SIGNAL 命令による制御の移動について、ユーザーが把握したい場合にも役立ちます。

正常

負の戻りコードを戻したコマンドがあれば、実行後に、コマンドからの戻りコードも一緒にトレースします。これがデフォルト設定です。

Off

トレースは何も行わず、特殊な接頭部オプション (後述します) を OFF にリセットします。

結果

すべての文節を実行前にトレースします。式の計算の最終結果 (前述の Intermediates と対比してください) を表示します。さらに、PULL、ARG、および PARSE 命令の実行時に割り当てられた値も表示します。一般的なデバッグの場合には、この設定値をお勧めします。

スキャン

データ内に残っているすべての文節を、実行せずにトレースします。基本検査 (END の脱落の検査など) が実行され、通常どおりにトレース結果がフォーマットされます。このオプションが有効なのは、

他の命令 (INTERPRET または対話式デバッグを含みます) あるいは 内部ルーチン内で、TRACE S 文節自身がネストされていない場合だけです。

接頭部オプション

接頭部の ! および ? が、単独で使用するか、または 英字オプションの 1 つと一緒に使用した場合に有効です。両方の接頭部を、任意の順序で、1 つの TRACE 命令に指定できます。1 つの接頭部を、複数回指定することができます。1 つの命令上に接頭部が現れるたびに、前の同じ接頭部のアクションが反転します。接頭部 (1 つ以上) は、オプションの直前に (ブランクを 間に入れずに) 指定する必要があります。

接頭部の ! および ? は、以下のように、トレースと実行を修正します。

?

対話式デバッグを制御します。通常の実行では、? の接頭部を指定した TRACE オプションにより、対話式デバッグがオンに切り替えられます。471 ページの『プログラムの対話式デバッグ』を参照してください。対話式デバッグがオンであると、ほとんどの文節がトレースされた後、解釈は停止します。例えば、TRACE ?E という命令は、エラー (つまり非ゼロの戻りコード) を戻したすべてのコマンドの実行後、言語プロセッサが入力を待って休止するようにします。

トレースされるプログラムの中に TRACE 命令があっても無視されます。(これは、予期せず対話式デバッグが終了しないようにするためです。)

対話式デバッグは、下記のとおり、いくつかの方法でオフに切り替えることができます。

- TRACE 0 を入力すると、すべてのトレースがオフにされます。
- オプションを指定せずに TRACE を入力すると、デフォルトが復元されます。対話式デバッグはオフになりますが、TRACE Normal (実行の結果、失敗したコマンドがあればトレースする) を有効にしてトレースが続行されます。
- TRACE ? を入力すると、対話式デバッグがオフになり、現行オプションを用いてトレースは続行されます。
- TRACE 命令でオプションを指定し、その前に ? 接頭部を付けて入力すると、対話式デバッグがオフになり、新しいオプションを用いてトレースが続行されます。

したがって、? 接頭部を使用すると、対話式デバッグに入ったり、出たり、切り替えることができます。(いったん対話式デバッグに入ってしまうと、言語処理プログラムはユーザー・プログラム内のそれ以降の TRACE ステートメントを無視するので、対話式デバッグをオフにする場合には、CALL TRACE '?' を使用してください。)

!

ホスト・コマンドの実行を禁止するために使用します。通常の実行中は、TRACE 命令の接頭部に ! を付けると、後続のホスト・コマンドすべての実行が延期されます。例えば、TRACE !C を使用すると、コマンドはトレースされますが、実行はされません。コマンドを迂回するたびに、REXX の特殊変数 RC は 0 にセットされます。この処置は、破壊的な影響を与える可能性のあるプログラムをデバッグする際に使用できます。対話式デバッグの間に手作業で入力されたコマンドが、このために禁止されることはありません。手作業で入力されたコマンドは、必ず実行されます。

コマンド禁止になっている場合は、接頭部 ! を付けた TRACE 命令を出せば、それをオフに切り換えることができます。したがって、! 接頭部を繰り返し使用すると、コマンド禁止モードに入ったり、出たり、交互に切り替わることになります。あるいは、TRACE 0 を出すか、オプションを指定しない TRACE を出せば、コマンド禁止モードをいつでもオフにできます。

数値オプション

対話式デバッグがアクティブであり、しかも指定したオプションが正の整数 (または、正の整数として評価される式) である場合、その数値は、スキップすべきデバッグ停止の数を示します。(471 ページの『プログラムの対話式デバッグ』を参照。) ただし、オプションが負の整数 (または、計算結果が負の整数になる式) の場合には、指定した数の文節だけ、デバッグ停止を含むすべてのトレースが一時的に禁止されます。例えば、TRACE -100 は、通常であればトレースされる、後続の 100 個の文節が、実際には表示されないという意味です。その後は、前のとおりトレースが再開します。

トレースに関するヒント

1. ループがトレースされる場合、DO 文節は、ループの反復のたびに トレースされます。
2. TRACE 組み込み関数を使用して、現在有効なトレース・アクションを検索できます ([218 ページの『TRACE』](#)参照)。
3. TRACE A、R、I、または S を指定した場合は、トレースされる文節に関連するコメントも、実行時に入手可能であれば、トレース結果に含まれます。ヌル文節内のコメントも同様です。
4. 実行前にトレースされるコマンドに入っているのは、常にコマンドの最終値(つまり、環境に渡される文字列)であり、その値を生成した文節がトレース出力内に作成されています。
5. トレース動作は、サブルーチンおよび関数の呼び出しのたびに、自動的に保管されます。 [165 ページの『CALL』](#)を参照してください。

例

最も一般的に使用するトレースの 1 つを以下に示します。

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/* and tracing Results of expressions.          */
```

TRACE 出力のフォーマット

トレースされる各文節は、ネストの論理レベルの数などに応じて、自動的にフォーマット (字下げ) されて表示されます。言語プロセッサは、データのエンコードの中の制御コード (例えば、'40'x より小さい EBCDIC 値) を疑問符 (?) に置き換えて、コンソールを妨害しないようにする場合があります。結果 (要求された場合) は、スペース 2 個分だけ余計に字下げされ、先行ブランクと末尾ブランクがはっきり分かるよう、二重引用符で囲まれます。

いずれの行においてもトレースされる最初の文節の前に、その行番号が示されます。行番号が 99999 より大きいと、言語プロセッサは左側を切り捨て、? 接頭部によって、切り捨てが行われたことを示します。例えば、行番号 100354 は、?00354 と表示されます。トレース中に表示される行のすべてに、3 文字の接頭部が付いて、トレースしている データのタイプが示されます。これらの接頭部は、以下のとおりです。

単一文節のソース・コード、つまり、プログラム内に実際にあるデータを表します。

+++

トレース・メッセージを表します。これには、コマンドからの非ゼロの戻りコード、対話式デバッグに入ったときのプロンプト・メッセージ、対話式デバッグ中における構文エラーの指示、またはプログラムの構文エラーの後のトレースバック文節場合があります。

>>>

式の結果 (TRACE R の場合)、解析中に変数に割り当てられた値、あるいは サブルーチン呼び出しから戻された値を表します。

>.>

構文解析中にプレースホルダーに「割り当てられる」値を識別します ([228 ページの『ピリオドはプレースホルダー』](#)を参照)。

以下の接頭部は、TRACE Intermediates が有効な 場合にのみ使用されます。

>C>

トレース・データは複合変数の名前で、置換後かつ使用前にトレースされたものです。ただし、変数の値が置換されてその名前に組み込まれている場合に限りです。

>F>

トレース・データは、関数呼び出しの結果です。

>L>

トレース・データは、リテラル (文字列、初期設定されていない変数、または定数記号) です。

>O>

トレース・データは、2つの項の演算結果です。

>P>

トレース・データは、接頭部演算の結果です。

>V>

トレース・データは、変数の内容です。

TRACE 命令にオプションを指定しない場合、あるいは式の計算結果がヌルの場合には、デフォルトのトレース動作が復元されます。デフォルトは、TRACE N、コマンド禁止 (!) オフ、および対話式デバッグ (?) オフです。

SIGNAL ON SYNTAX によってトラップされない構文エラーの後で、エラーのある文節が必ずトレースされます。エラーの発生時にアクティブな CALL または INTERPRET、あるいは関数の呼び出しもトレースされます。検出できなかったラベルへの制御の転送を試みたためにこのエラーが発生した場合は、そのラベルもトレースされます。特別のトレース接頭部 +++ は、これらのトレースバックの行を識別します。

UPPER

UPPER は、1つまたは複数の変数の内容を大文字に変換します。変数は、左から右へ順番に変換されていきます。

これは、REXX/CICS で提供される非 SAA 命令です。

➡ UPPER  variable ; ➡

variable はシンボルであり、1つ以上のブランクまたはコメントで他の *variable* と区切ります。単純シンボルおよび複合シンボルのみを指定してください。154 ページの『単純記号』を参照してください。

TRANSLATE 組み込み関数を繰り返し呼び出すよりも、この命令を使用する方が便利です。

例

```
a1='Hello'; b1='there'
Upper a1 b1
say a1 b1      /* Displays "HELLO THERE" */
```

定数シンボルまたは語幹が検出されると、エラーが通知されます。初期設定されていない変数を使用してもエラーにはならず、影響はありません (ただし、NOVALUE 条件 (SIGNAL ON NOVALUE) が有効になっている場合は、トラップされます)。

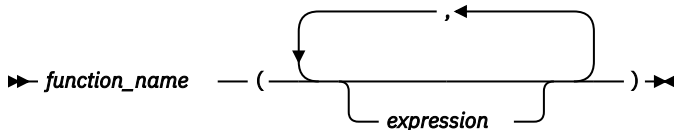
第 19 章 関数

関数とは、単一の結果ストリングを返す内部ルーチン、組み込みルーチン、または外部ルーチンです。(サブルーチンとは、CALL 命令で呼び出され、結果を返すことも、返さないこともある、内部ルーチン、組み込みルーチン、または外部ルーチンである関数です。)

構文

関数呼び出しは、式の中の項目であり、何らかの手順を実行してストリングを返すルーチンを呼び出します。戻されたストリングが、引き続き行われる式の計算の中で、関数呼び出しを置き換えます。

下記の表記法を使用すると、データ項 (ストリングなど) が有効である場所であれば、式の中に内部ルーチンおよび外部ルーチンを呼び出す関数呼び出しを組み込むことができます。



`function_name` は、リテラル・ストリングまたは単一記号であり、定数として扱われます。

括弧の中には、実装で定義されている最大数までの式をコンマで分けて書くことができます。REXX/CICS の場合、組み込むことのできる式の最大数は 20 個です。これらの式のことを、関数に対する引数と呼びます。引数であるそれぞれの式の中に、さらに関数呼び出しを含めることができます。

左括弧は関数の名前に隣接している必要があり、その間にブランクを入れてはなりません。これに違反した構造では、関数呼び出しとして認識されません。(この場合は代わりに、ブランク演算子と見なされます。) 名前と左括弧の間に入れることができるのは、コメント (なんの効力もありません) だけです。

引数が左から右に順に計算された後で、結果のストリングがすべて関数に渡されます。関数は、何らかの演算 (引数の指定は必須ではありませんが、通常は、渡された引数ストリングによって決まります) を実行し、最終的には単一の文字ストリングを返します。そして、このストリングが元の式に組み込まれます。つまり、関数参照全体を変数の名前でも置き換えて考えてみると、戻されたデータがその変数の値として入れられる、という場合と同様の仕組みであることが見てとれるでしょう。

例えば、関数 SUBSTR は言語処理プログラム (セクション 215 ページの『SUBSTR (サブストリング)』を参照) に組み込まれており、次のように使用できます。

```
N1='abcdefghijk'  
Z1='Part of N1 is: 'substr(N1,2,7)  
/* Sets Z1 to 'Part of N1 is: bcdefgh' */
```

関数が受け取る引数の数は可変です。用途に合わせて必要な引数だけを指定するようにしてください。例えば、SUBSTR('ABCDEF',4) と指定すると、DEF を返します。

関数およびサブルーチン

関数呼び出しの仕組みは、サブルーチンの呼び出しの仕組みと同じです。関数とサブルーチンの唯一の違いは、関数はデータを返す必要があるのに対し、サブルーチンにはその必要がないという点です。

下記のタイプのルーチンを、関数として呼び出すことができます。

内部

ルーチン名がプログラム内でラベルとして使われている場合には、現行の処理状況が保管されるため、後で呼び出し点に戻って実行を再開することができます。制御は、その後、この名前に一致するプログラム内の最初のラベルに渡されます。CALL 命令で呼び出されるルーチンと同様に、他のさまざまな状況情報 (TRACE 設定値や NUMERIC 設定値など) も保管されます。詳しくは、『CALL 命令』(165 ページの『CALL』) を参照してください。SIGNAL と CALL を一緒に使用すると、実行の時点で名前が決定される内部ルーチンを呼び出すことができます。これは、マルチウェイ呼び出しと呼ばれます (186 ページの『SIGNAL』を参照)。

内部ルーチンを関数として呼び出す場合は、それから戻るために RETURN 命令内で式を指定しなければなりません。サブルーチンとして呼び出す場合には、この指定は必要ありません。例えば、次のようになります。

```
/* Recursive internal function execution... */
arg x
say x'!' = ' factorial(x)
exit

factorial: procedure /* Calculate factorial by */
arg n /* recursive invocation. */
if n=0 then return 1
return factorial(n-1) * n
```

内部ラベルの探索中に、構文検査が行われ、EXEC がトークン化されます。詳細については、[パフォーマンスの考慮事項](#)を参照してください。FACTORIAL は、自分自身を呼び出す (これは再帰呼び出しです) という点で通常とは異なります。PROCEDURE 命令は、呼び出しのたびに新しい変数 n が必ず作成されるようにします。

注: ルーチンを探索する場合、言語処理プログラムは現在、REXX プログラム内のステートメントをスキャンして、内部ラベルを探し出します。この探索中に、言語処理プログラムは構文エラーを検出する場合があります。その結果、処理中の元の行とは別のステートメントで構文エラーが起こる可能性もあります。

組み込み

これらの関数は常に使用可能であり、[196 ページの『組み込み関数』](#)で規定されています。

外部

使用するプログラムおよび言語処理プログラムにとって外部の関数を作成したり使用したりすることができます。外部ルーチンは REXX で作成する必要があります。REXX プログラムを関数として呼び出すことができますが、この場合は、複数の引数ストリングを渡すことができます。ARG 命令や PARSE ARG 命令、または ARG 組み込み関数は、これらの引数ストリングを検索することができます。関数として呼び出された場合、プログラムは、データを呼び出し元に戻す必要があります。

注:

1. 外部 REXX 関数は、CICS がサポートする任意の言語で作成されたプログラムに対して EXEC CICS LINK を容易に実行することができます。アセンブラで REXX/CICS コマンド・ルーチンを作成することもできます。
2. 関数としての外部 REXX プログラムの呼び出しは、内部ルーチンの呼び出しと類似しています。ただし、外部ルーチンは、呼び出し元のすべての変数が常に隠され、内部値 (NUMERIC 設定値など) の状況が (呼び出し元の値を継承するのではなく) デフォルトで始まるという点で、暗黙的な PROCEDURE です。
3. その他の REXX プログラムは、関数として呼び出すことができます。呼び出した REXX プログラムから戻る場合は、EXIT または RETURN を使用することができますが、いずれの場合も、式を指定しなければなりません。
4. 注意しながら INTERPRET 命令を使用して、可変の関数名を持つ関数を実行することができます。ただし、プログラムの明確さが損なわれるので、可能であれば避けてください。

検索順序

関数の検索は、内部ルーチン、組み込み関数、外部関数の順に行われます。

関数名がリテラル・ストリングとして指定された (つまり、引用符で囲んで指定された) 場合、内部ルーチンは使用されません。この場合、関数は組み込み関数または外部関数でなければなりません。これにより、例えば、組み込み関数の名前を利用して、その機能を拡張することができます。しかも、その組み込み関数は、必要なときには呼び出すことができます。例えば、次のようになります。

```
/* This internal DATE function modifies the */
/* default for the DATE function to standard date. */
date: procedure
arg in
if in='' then in='Standard'
return 'DATE'(in)
```


組み込み関数は、大文字の名前を持っています。したがって、例に示すように、探索を成功させるには、リテラル・ストリング内の名前は大文字でなければなりません。通常は、外部関数についても同じことがいえます。

外部関数およびサブルーチンの探索順序は、システムで定義されています。

REXX で作成された EXEC、コマンド、外部関数、またはサブルーチンが REXX/CICS から呼び出されたときには (例えば、CICS コマンド行、CALL 命令、EXEC コマンド、または EXEC 内のコマンドから)、必ず、以下の検索順序でターゲット EXEC が見つかります。

- DEFCMD で定義されたコマンド EXEC またはその他の呼び出された EXEC の場合の検索順序。

許可コマンドを実行しようとするものである場合は、許可ユーザーかどうか、あるいは、DD 名 CICEXEC または CICAUTH からロードされた EXEC 内にそのコマンドがあるかどうかをチェックされます。それらのいずれも該当しない場合、コマンドは、戻りコード -4 で失敗します。

1. 内部関数またはサブルーチンの現在の EXEC 内。

この検索は、関数 (またはサブルーチン) の名前を引用符で囲むことによってそれを外部のものとして明示的に識別することで迂回できます。

2. ストレージ内の EXEC (以前の EXECLOAD からのもの)。

3. 現在の RFS ディレクトリー。

4. 現在の PATH。PATH コマンドが実行された場合、RFS ディレクトリーおよび MVS 区分データ・セットが、最新の PATH コマンドでリストされた順序で検索されます。[394 ページの『PATH』](#)を参照してください。

5. DD 名 CICAUTH に割り振られた (または連結された) PDS データ・セット。

ユーザーが許可ユーザーである場合、または現行 EXEC が CICEXEC からロードされていて、検索が許可コマンドのプログラムを対象としている場合は、CICAUTH DD 名がチェックされます。

6. DD 名 CICEXEC に割り振られた (または連結された) PDS データ・セット。

7. DD 名 CICUSER に割り振られた (または連結された) PDS データ・セット。

DEFCMD コマンドについて詳しくは、[374 ページの『DEFCMD』](#)を参照してください。

注:

1. REXX/CICS EXEC が呼び出される MVS データ・セット・タイプは、MVS 区分編成 (DSORG=PO) のみです。MVS 順次データ・セット (DSORG=PS) はサポートされていません。
2. MVS 区分データ・セットにある REXX EXEC には、73 桁目から 80 桁目までのシーケンス番号がない場合があります。
3. (PATH リストを左から右へ処理して) ターゲット EXEC が見つからないままデータ・セット名に達した場合は、最後の PATH ステートメントで指定されていた MVS 区分データ・セットを動的に (SVC 99 によって) 割り振ろうとします。
4. DEFCMD AUTH パラメーターを使用して、REXX/CICS コマンドが許可コマンドであることを指定できます。
5. EXECLOAD および EXECDROP コマンドは許可コマンドであるため、REXX/CICS 許可ユーザーまたは CICEXEC からロードされた EXEC しか、これらのコマンドは実行できません。

実行中にエラーが発生した

外部関数または組み込み関数がエラーを検出すると、言語処理プログラムに通知され、構文エラーになります。したがって、関数呼び出しを含んでいた文節の実行は終了します。同様に、外部関数がデータを正しく戻さなかった場合は、言語処理プログラムがこのことを検出し、エラーとして報告します。

内部関数の実行中に発生した構文エラーはトラップできる (SIGNAL ON SYNTAX を使用) ため、リカバリーが可能な場合もあります。エラーがトラップされないと、プログラムは終了します。


```
...
otherwise nop;
end;
```

ABS (絶対値)

ABS 関数は、*number* の絶対値を戻します。結果は、符号なしで、NUMERIC の現行設定値に応じて形式設定されます。

➡ ABS(— *number* —) ➡

例

```
ABS('12.3')      ->    12.3
ABS(' -0.307')   ->    0.307
```

ADDRESS

ADDRESS 関数は、コマンドが現在実行依頼されている対象の環境の名前を戻します。この環境は、サブコマンド環境の名前である場合があります。

➡ ADDRESS(—) ➡

詳細については、ADDRESS 命令 ([163 ページの『ADDRESS』](#)) を参照してください。末尾ブランクは結果からは除かれます。

例

```
ADDRESS()      ->    'CICS'      /* default under CICS */
ADDRESS()      ->    'EDITSVR'   /* default under CICS editor */
```

ARG (引数)

ARG 関数は、プログラムまたは内部ルーチンに対する引数ストリングに関する情報、あるいは引数ストリングを 戻します。

➡ ARG(—————) ➡

└─ *n* ─┘

└─ , — *option* ─┘

n を指定しないと、プログラムまたは内部ルーチンに渡される引数の数が返ります。

n だけを指定した場合は、*n* 番目の引数ストリングが戻されます。引数ストリングが存在しない場合には、ヌル・ストリングが戻されます。*n* は、正の整数でなければなりません。

option を指定すると、ARG は *n* 番目の引数ストリングがあるかどうか調べます。有効なオプションを以下に示します。(大文字で強調表示された文字だけが必要であり、後続の文字はすべて無視されます。)

Exists

n 番目の引数があれば、1 が返ります。(つまり このルーチンが呼ばれたときに明示的に指定されていれば。) それ以外の場合は、0 が返ります。

Omitted

n 番目の引数が省略されていれば、1 が返ります。(つまり このルーチンが呼ばれたときに明示的に指定されていない場合。) それ以外の場合は、0 が返ります。

例

```
/* following "Call name;" (no arguments) */
ARG()      ->    0
ARG(1)     ->    ''
ARG(2)     ->    ''
```

```

ARG(1,'e')    -> 0
ARG(1,'0')    -> 1

/* following "Call name 'a',,'b';" */
ARG()         -> 3
ARG(1)        -> 'a'
ARG(2)        -> ''
ARG(3)        -> 'b'
ARG(n)        -> ''      /* for n>=4 */
ARG(1,'e')    -> 1
ARG(2,'E')    -> 0
ARG(2,'0')    -> 1
ARG(3,'o')    -> 0
ARG(4,'o')    -> 1

```

注:

1. 引数ストリングの数は、明示的な引数ストリングがない場合に ARG(*n*, 'e') が 1 または 0 を返す最大数 *n* です。つまり、これは、明示的に指定された最後の引数ストリングの位置です。
2. コマンドとして呼び出されるプログラムの場合、引数ストリングは 1 つまたは無しです。名前だけで呼び出されるプログラムの場合は、引数ストリングは無しで、コマンドにその他 (ブランクを含む) が付く場合は、引数ストリングは 1 個になります。
3. ユーザーは、ARG または PARSE ARG 命令を用いて、引数ストリングを検索して、プログラムまたは内部ルーチンに対してそれらを直接分析することもできます。(164 ページの『ARG』、179 ページの『PARSE』、227 ページの『第 20 章 構文解析』を参照してください。)

BITAND (ビットごとの AND)

BITAND 関数は、ビットごとに論理的に AND 演算された 2 つの入力ストリングから成るストリングを返します。

➡ BITAND(— *string1* —————) ➡

, —————

string2 —————

, — *pad* —————

論理演算においては、ストリングをエンコードしたものが使用されます。結果の長さは、2 つのストリングのうち長い方のストリングの長さになります。 *pad* 文字が指定されていない場合、2 つのストリングのうち短い方が終わると AND 演算は停止し、長い方のストリングの未処理部分が、そこまでの部分結果に付加されます。 *pad* が指定された場合には、これが短い方のストリングの右側に付加されてから、論理演算が実行されます。 *string2* のデフォルトは、長さが 0 の (ヌル) ストリングです。

例

```

BITAND('12'x)          -> '12'x
BITAND('73'x,'27'x)     -> '23'x
BITAND('13'x,'5555'x)   -> '1155'x
BITAND('13'x,'5555'x,'74'x) -> '1154'x
BITAND('pQrS',,'BF'x)   -> 'pqrs'      /* EBCDIC */

```

BITOR (ビットごとの OR)

BITOR 関数は、ビットごとに論理的に包含 OR 演算された 2 つの入力ストリングから成るストリングを返します。

➡ BITOR(— *string1* —————) ➡

, —————

string2 —————

, — *pad* —————

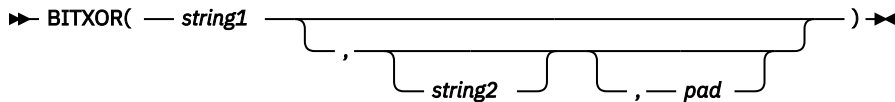
論理演算においては、ストリングをエンコードしたものが使用されます。結果の長さは、2 つのストリングのうち長い方のストリングの長さになります。 *pad* 文字が指定されていない場合、2 つのストリングのうち短い方が終わると OR 演算は停止し、長い方のストリングの未処理部分が、そこまでの部分結果に付加されます。 *pad* が指定された場合には、これが短い方のストリングの右側に付加されてから、論理演算が実行されます。 *string2* のデフォルトは、長さが 0 の (ヌル) ストリングです。

例

```
BITOR('12'x)          -> '12'x
BITOR('15'x,'24'x)     -> '35'x
BITOR('15'x,'2456'x)   -> '3556'x
BITOR('15'x,'2456'x,'F0'x) -> '35F6'x
BITOR('1111'x,'4D'x)   -> '5D5D'x
BITOR('pQrS'','','40'x) -> 'PQRS' /* EBCDIC */
```

BITXOR (ビットごとの排他 OR)

BITXOR 関数は、2つの入力字符串をビット単位の排他 OR で論理演算して合成された字符串を戻します。



論理演算においては、字符串をエンコードしたものが使用されます。結果の長さは、2つの字符串のうち長い方の字符串の長さになります。*pad* 文字が指定されていない場合、2つの字符串のうち短い方が終わると XOR 演算は停止し、長い方の字符串の未処理部分が、そこまでの部分結果に付加されます。*pad* が指定された場合には、これが短い方の字符串の右側に付加されてから、論理演算が実行されます。*string2* のデフォルトは、長さが 0 の (ヌル) 字符串です。

例

```
BITXOR('12'x)          -> '12'x
BITXOR('12'x,'22'x)     -> '30'x
BITXOR('1211'x,'22'x)   -> '3011'x
BITXOR('1111'x,'444444'x) -> '555544'x
BITXOR('1111'x,'444444'x,'40'x) -> '555504'x
BITXOR('1111'x,'4D'x)   -> '5C5C'x
BITXOR('C711'x,'222222'x,' ') -> 'E53362'x /* EBCDIC */
```

B2X (2 進数から 16 進数へ)

B2X 関数は、文字形式の字符串を戻します。この字符串は、*binary_string* を 16 進数へ変換したものです。

➡ B2X(*binary_string*) ➡

binary_string は、2 進数字 (0 または 1) の字符串です。この長さは任意です。オプションで、*binary_string* に空白を入れて (先行や末尾でなく、4 桁境界でのみ) 読みやすくすることができます。これらの空白は無視されます。

戻される字符串には、A から F までの値の大文字の英字が使用され、空白は含まれません。

binary_string がヌル・字符串の場合、B2X はヌル・字符串を戻します。*binary_string* 内の 2 進数の数が 4 の倍数でない場合には、合計で 4 の倍数になるように、左側に数字の 0 を最大 3 つ追加してから変換されます。

例

```
B2X('11000011') -> 'C3'
B2X('10111')     -> '17'
B2X('101')        -> '5'
B2X('1 1111 0000') -> '1F0'
```

B2X を関数 X2D および X2C と組み合わせると、2 進数を他の形式に変換することができます。以下に例を示します。

```
X2D(B2X('10111')) -> '23' /* decimal 23 */
```


Condition name

現在トラップされている条件の名前を返します。

説明

現行のトラップ条件に関して記述したストリングを返します。記述が存在しない場合は、ヌル・ストリングを返します。

命令

CALL または SIGNAL、つまり、現行条件がトラップされたときに処理された命令のキーワードを返します。option を省略すると、このオプションがデフォルトになります。

状況

現在トラップされている条件の状況を返します。これは、処理中に変わることがあります。以下のいずれかになります。

ON - 条件は、使用可能です。

OFF - 条件は、使用不可です。

DELAY - 条件の新たな発生はいずれも、遅延されるか、無視されます。

トラップされた条件がない場合、CONDITION 関数は 4 つのケースのすべてにおいてヌル・ストリングを返します。

例

```
CONDITION()      -> 'CALL'      /* perhaps */
CONDITION('C')   -> 'FAILURE'
CONDITION('I')   -> 'CALL'
CONDITION('D')   -> 'FailureTest'
CONDITION('S')   -> 'OFF'        /* perhaps */
```

注: CONDITION 関数は、複数のサブルーチン呼び出し (CALL ON 条件トラップが原因のものを含む) にまたがって保管および復元される条件情報を返します。したがって、CALL ON *trapname* で呼び出されたサブルーチンが戻った後で、現在トラップされている条件は、CALL が行われる前に現行であった条件 (何もない場合もあります) に逆戻りします。CONDITION は、条件がトラップされる前に戻した値を返します。

COPIES

COPIES *string* のコピーを *n* 個連結したものを返します。*n* は、正の整数またはゼロでなければなりません。

➡ COPIES(— *string* — , — *n* —) ➡

例

```
COPIES('abc',3)   -> 'abcabcabc'
COPIES('abc',0)   -> ''
```

C2D (文字から 10 進数へ)

C2D 関数は、文字表記の *string* を、10 進数値で返します。

➡ C2D(— *string* — , — *n* —) ➡

結果を整数として表せない場合には、エラーになります。つまり、結果の桁数が、NUMERIC DIGITS の現行設定値を超えてはなりません。*n* を指定すると、この値が、戻される結果の長さになります。*n* を指定しない場合、*string* は、無符号の 2 進数として処理されます。

string がヌルの場合は、0 を返します。

実際の最大値: 入力ストリングに、最終結果を形成する上での有効な文字数 250 を超える文字を入れることはできません。先行符号文字 ('00'x および 'FF'x) は、この合計数にはカウントされません。

例

```
C2D('09'X)      ->      9
C2D('81'X)      ->     129
C2D('FF81'X)    ->    65409
C2D('')         ->      0
C2D('a')        ->     129    /* EBCDIC */
```

n を指定すると、string は、*n* 個の文字で表された符号付きの数として処理されます。この数は、左端のビットがオフであれば正であり、その同じビットがオンであれば、2 の補数表記の負です。いずれにしても、これは整数に変換されます。したがって、負になる場合もあります。string は、その左側に '00'x 文字が埋め込まれるか (「符号で拡張される」のではありません)、あるいは左側が切り捨てられて、*n* 個の文字にされます。この埋め込みまたは切り捨ては、RIGHT (string、*n*、'00'x) が実行された場合と同様に行われます。*n* が 0 の場合には、C2D は、必ず 0 を返します。

```
C2D('81'X,1)    ->    -127
C2D('81'X,2)    ->     129
C2D('FF81'X,2)  ->    -127
C2D('FF81'X,1)  ->    -127
C2D('FF7F'X,1)  ->     127
C2D('F081'X,2)  ->   -3967
C2D('F081'X,1)  ->    -127
C2D('0031'X,0)  ->      0
```

C2X (文字から 16 進数へ)

C2X 関数は、文字形式の string を返します。この string は、string を 16 進数に変換したものです。

➡ C2X(— string —) ➡

戻される string のバイト数は、該当の入力 string の 2 倍です。例えば、EBCDIC システムでは、文字 1 の EBCDIC 表現が 'F1'X であるため、C2X(1) は F1 を返します。

戻される string は、A から F の値に対して英大文字を使用しますが、空白は含まれません。string の長さは任意です。string がヌルの場合は、ヌル・string を返します。

例

```
C2X('72s')      ->    'F7F2A2' /* 'C6F7C6F2C1F2'X in EBCDIC */
C2X('0123'X)    ->    '0123'   /* 'F0F1F2F3'X   in EBCDIC */
```

DATATYPE

string のみが指定され、string がエラーなしで 0 に加算できる有効な REXX 数値である場合、DATATYPE 関数は NUM を返します。string が有効な数値でない場合、DATATYPE は CHAR を返します。

➡ DATATYPE(— string — , — type —) ➡

type を指定する場合、string が type に一致する場合は 1 を、そうでない場合は 0 を返します。string がヌルの場合、この関数は 0 を返します (ただし、type が X の場合には、ヌル・string に対して 1 を返すので、例外となります)。有効な type は、以下のとおりです。大文字だけが必要であり、後続の文字はすべて無視されます。hexadecimal オプションの場合、オプションの名前を指定する string は、h ではなく、x で始まる必要があります。)

Alphanumeric

string 内の文字が、a~z、A~Z、および 0~9 の範囲内の文字のみであれば、1 を返します。

Binary

string 内の文字が、0 または 1 (あるいはその両方) だけであれば、1 を返します。

C

string が大文字小文字混合の SBCS/DBCS string であるときは、1 を返します。

Dbcs

string が SO バイトと SI バイトで囲まれている DBCS 専用ストリングであれば、1 を返します。

Lowercase

string 内の文字が、a～z の範囲内の文字のみであれば、1 を返します。

大/小文字混合

string 内の文字が、a～z および A～Z の範囲内の文字のみであれば、1 を返します。

Number

string が有効な REXX 数であれば、1 を返します。

記号

string 内の文字が、REXX 記号で使える文字だけであれば、1 を返します。(143 ページの『トークン』を参照。) 英字の大文字小文字の両方とも許されることに注意してください。

Uppercase

string 内の文字が、A から Z の範囲内の文字だけであれば、1 を返します。

Whole number

string が NUMERIC DIGITS の現行設定値のもとで有効な REXX 整数であれば、1 を返します。

hexadecimal

string 内の文字が、a から f、A から F、0 から 9、およびブランクの範囲内の文字だけであれば、1 を返します (ただし、ブランクは、一對の 16 進文字の間だけにしか認められません)。 *string* がヌル・ストリングである場合にも、1 を返します。ヌル・ストリングは有効な 16 進ストリングだからです。

注: DATATYPE 関数は、ストリング内の文字のエンコード (例えば、ASCII や EBCDIC など) に関係なく、それらの文字の意味またはタイプをテストします。

例

```
DATATYPE(' 12 ') -> 'NUM'
DATATYPE('') -> 'CHAR'
DATATYPE('123*') -> 'CHAR'
DATATYPE('12.3', 'N') -> 1
DATATYPE('12.3', 'W') -> 0
DATATYPE('Fred', 'M') -> 1
DATATYPE('', 'M') -> 0
DATATYPE('Fred', 'L') -> 0
DATATYPE('?20K', 'S') -> 1
DATATYPE('BCd3', 'X') -> 1
DATATYPE('BC d3', 'X') -> 1
```

DATE

DATE 関数は、デフォルトでは形式 *dd mon yyyy* (日 月 年 (例えば、13 mar 1992)) でローカルの日付を返します。日の前にゼロやブランクは付けられません。

➡ DATE(option) ➡

使用中の言語に月の名前の省略形があれば使用されます (例えば、Jan や Feb)。

以下のオプションを使用すると、特定の形式を取得できます。大文字だけが必要であり、後続の文字はすべて無視されます。

Base

基本日付 (0001 年 1 月 1 日) から数えた経過日数 (つまり、現在日は含みません) を返します。形式は、*dddddd* です (先行ゼロやブランクはありません)。式 DATE('B')//7 は、現在の曜日 (0 が月曜日で、6 が日曜日) に対応する、0 から 6 までの範囲の数値を返します。

したがって、この関数を使用して、各国語に関係なく、曜日を判別できます。

注: 基本日付の 0001 年 1 月 1 日は、現在のグレゴリオ暦から逆算して決められます (1 年を 365 日とします。ただし、4 で割りきれ年であれば 1 日足しますが、400 で割り切れない年は例外となります)。グレゴリオ暦を最初に作成したときの暦の体系におけるエラーは、考慮されていません。

Century

100 の倍数である直前の年の 1 月 1 日から数えた、現在日を含む日数を戻します。形式は、*dddddd* です (先行ゼロはありません)。例えば、1992 年 3 月 13 日の DATE(C) の呼び出しでは、1900 年 1 月 1 日から 1992 年 3 月 13 日までの日数 33675 が戻されます。同様に、2000 年 1 月 2 の DATE(C) の呼び出しでは、2000 年 1 月 1 日から 2000 年 1 月 2 日までの日数 2 が戻されます。

Days

今年の、現在日を含む日数を戻します。形式は *ddd* です (先行ゼロやブランクはありません)。

European

日付を形式 *dd/mm/yy* で戻します。

Julian

日付を形式 *yyddd* で戻します。

Month

現在の月の完全な英語名を戻します (例: August)。

正常

日付を形式 *dd mon yyyy* で戻します。これがデフォルトです。

Ordered

日付を形式 *yy/mm/dd* で戻します (ソートなどに適しています)。

標準

日付を形式 *yyyymmdd* で戻します (ソートなどに適しています)。

Usa

日付を形式 *mm/dd/yy* で戻します。

Weekday

曜日を大/小文字混合の英語名で戻します (例: Tuesday)。

注: ある文節内で DATE または TIME を初めて呼び出したときに、タイム・スタンプが作成されます。それ以降、このタイム・スタンプが、その文節内のこれらすべての関数呼び出しに使用されます。したがって、DATE 関数と TIME 関数のいずれか、あるいはその両方を 1 つの式または文節内で複数回呼び出しても、相互に必ず整合性が保たれます。

例

次の例では、現在の日付が 1992 年 3 月 13 日であると仮定しています。

```
DATE()      -> '13 Mar 1992'
DATE('B')   -> 727269
DATE('C')   -> 33675
DATE('D')   -> 73
DATE('E')   -> '13/03/92'
DATE('J')   -> 92073
DATE('M')   -> 'March'
DATE('N')   -> '13 Mar 1992'
DATE('O')   -> '92/03/13'
DATE('S')   -> '19920313'
DATE('U')   -> '03/13/92'
DATE('W')   -> 'Friday'
```

DBCS (2 バイト文字セット関数)

DBCS 処理関数の一部である関数をリストしています。

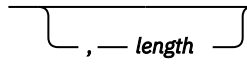
- DBADJUST
- DBBRACKET
- DBCENTER
- DBCJUSTIFY
- DBLEFT
- DBRIGHT
- DBRLEFT

- DBRRIGHT
- DBTODBCS
- DBTOSBCS
- DBUNBRACKET
- DBVALIDATE
- DBWIDTH

453 ページの『第 33 章 2 バイト文字セット (DBCS) サポート』を参照してください。

DELSTR (ストリングの削除)

DELSTR 関数は、*n* 番目の文字で始まり、*length* 個の文字から成るサブストリングを削除後に、*string* を戻します。

➡ DELSTR(— *string* — , — *n* —) ➡


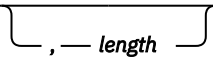
length を省略した場合、または *length* が *n* から *string* の終わりまでの文字数より大きい場合、この関数は *string* の残り (*n* 番目の文字を含む) を削除します。 *length* は、正の整数またはゼロでなければなりません。 *n* は、正の整数でなければなりません。 *n* が *string* の長さを超えている場合、この関数は *string* を変更せずに戻します。

例

```
DELSTR('abcd',3)      -> 'ab'
DELSTR('abcde',3,2)   -> 'abe'
DELSTR('abcde',6)     -> 'abcde'
```

DELWORD (ワードの削除)

DELWORD 関数は、*n* 番目のワードで始まり、空白で区切られた、*length* 個のワードから成るサブストリングを削除後に、*string* を戻します。

➡ DELWORD(— *string* — , — *n* —) ➡


length を省略した場合、または *length* が *n* から *string* の終わりまでのワード数より大きい場合、この関数は *string* 内の残りのワード (*n* 番目のワードを含む) を削除します。 *length* は、正の整数またはゼロでなければなりません。 *n* は、正の整数でなければなりません。 *n* が、*string* のワード数より大きい場合は、この関数は、*string* を変更せずに戻します。 削除されるストリングには、最後のワードの後にある空白も含まれますが、最初のワードより前の空白は含まれません。

例

```
DELWORD('Now is the time',2,2) -> 'Now time'
DELWORD('Now is the time ',3)  -> 'Now is '
DELWORD('Now is the time',5)   -> 'Now is the time'
DELWORD('Now is the time',3,1) -> 'Now is time'
```

DIGITS

DIGITS 関数は、NUMERIC DIGITS の現行設定値を戻します。

➡ DIGITS(—) ➡

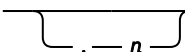
詳しくは、NUMERIC 命令 [176 ページの『NUMERIC』](#) を参照してください。

例

```
DIGITS()    ->    9    /* by default */
```

D2C (10 進数から文字へ)

D2C 関数は、10 進数で指定した *wholenumber* を 2 進に変換したものが表す、文字フォーマットのストリングを戻します。

➡ D2C(— *wholenumber* ) ➡

n を指定すると、それが最終結果の文字の長さとなります。この場合、入力ストリングは、変換後に、要求された長さまで符号で拡張されます。数値が大きすぎて *n* 文字に収まらない場合、結果は左側が切り捨てられます。*n* は、正の整数またはゼロでなければなりません。

n を省略する場合、*wholenumber* は正の整数またはゼロでなければならず、結果の長さは、必要に応じた長さになります。したがって、戻される結果に先行の '00'x 文字は付きません。

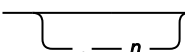
実際の最大値: 出力ストリングは、有効文字数の 250 字を超えてはなりません。ただし、先行符号文字 ('00'x および 'FF'x) が付いた結果として、これより長くなることはあります。

例

```
D2C(9)          -> ' '    /* '09'x is unprintable in EBCDIC */
D2C(129)         -> 'a'    /* '81'x is an EBCDIC 'a' */
D2C(129,1)       -> 'a'    /* '81'x is an EBCDIC 'a' */
D2C(129,2)       -> ' a'   /* '0081'x is EBCDIC ' a' */
D2C(257,1)       -> ' '    /* '01'x is unprintable in EBCDIC */
D2C(-127,1)      -> 'a'    /* '81'x is EBCDIC 'a' */
D2C(-127,2)      -> ' a'   /* 'FF'x is unprintable EBCDIC; */
                  /* '81'x is EBCDIC 'a' */
D2C(-1,4)        -> ' '    /* 'FFFFFFF'x is unprintable in EBCDIC */
D2C(12,0)        -> ''     /* '' is a null string */
```

D2X (10 進数から 16 進数へ)

D2X 関数は、10 進数で指定した *wholenumber* を 16 進に変換したものを表す、文字フォーマットのストリングを戻します。

➡ D2X(— *wholenumber* ) ➡

戻されるストリングでは、値に大文字の英字 A から F を使用し、ブランクは含まれません。

n を指定すると、それが最終結果の文字の長さとなります。この場合、入力ストリングは、変換後、要求された長さまで符号で拡張されます。数値が大きすぎて *n* 文字に収まらない場合は、左側が切り捨てられます。*n* は、正の整数またはゼロでなければなりません。

n を省略する場合、*wholenumber* は正の整数またはゼロでなければならず、戻される結果に先行ゼロは付きません。

実際の最大値: 出力ストリングに、500 個を超える有効な 16 進文字を入れることはできません。ただし、追加の先行符号文字 (0 および F) がある場合には、結果はそれより長くなる場合があります。

例

```
D2X(9)          -> '9'
D2X(129)         -> '81'
D2X(129,1)       -> '1'
D2X(129,2)       -> '81'
D2X(129,4)       -> '0081'
D2X(257,2)       -> '01'
D2X(-127,2)      -> '81'
```

```
D2X(-127,4)  ->  'FF81'  
D2X(12,0)    ->  ''
```

ERRORTXT

ERRORTXT 関数は、エラー番号 *n* に関連付けられた REXX エラー・メッセージを返します。

➡ ERRORTXT(— *n* —) ➡

n は、0～99 の範囲内の値でなければならず、それ以外の値はエラーになります。*n* が指定可能な範囲内の値であっても、定義済みの REXX エラー番号でなければ、ヌル・ストリングを返します。エラー番号およびエラー・メッセージの詳細については、[411 ページの『第 31 章 エラー番号とメッセージ』](#)を参照してください。

例

```
ERRORTXT(16)  ->  'Label not found'  
ERRORTXT(60)  ->  ''
```

EXTERNALS

EXTERNALS 関数は、端末入力バッファ内 (システム外部イベント・キュー) のエレメント数を返します。

➡ EXTERNALS(—) ➡

CICS には、同等のバッファはありません。したがって、REXX の CICS 実装環境では、EXTERNALS 関数は常に 0 を返します。以下に例を示します。

```
EXTERNALS()  ->  0    /* Always */
```

FIND

FIND 関数は、*string* で見つかった *phrase* の最初のワードのワード番号を返します。*phrase* が見つからない場合、または *phrase* にワードがない場合は、0 を返します。

WORDPOS は、このタイプのワード探索のための優先組み込み関数です。[222 ページの『WORDPOS \(ワード位置\)』](#)を参照してください。

➡ FIND(— *string* — , — *phrase* —) ➡

phrase は、ブランクで区切られた一連のワードです。*phrase* または *string* 内のワードの間にある複数のブランクは、比較のときには 1 個のブランクとして扱われます。

例

```
FIND('now is the time','is the time')  ->  2  
FIND('now is the time','is the')       ->  2  
FIND('now is the time','is time ')     ->  0
```

FORM

FORM 関数は、NUMERIC FORM の現行設定値を返します。

➡ FORM(—) ➡

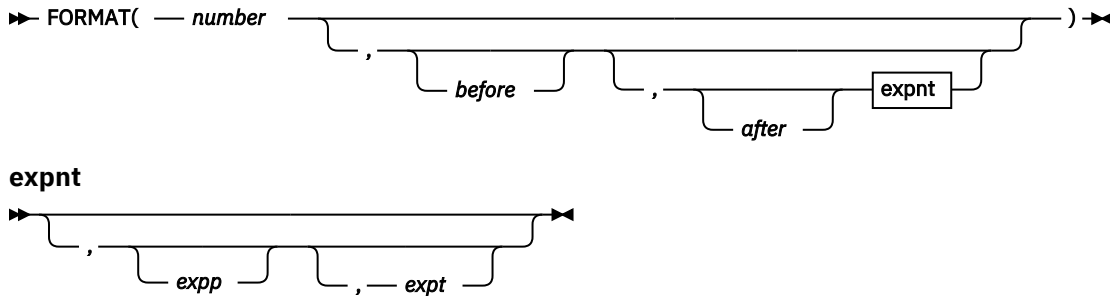
NUMERIC 命令 [176 ページの『NUMERIC』](#)を参照してください。

例

```
FORM()  ->  'SCIENTIFIC' /* by default */
```

FORMAT

FORMAT 関数は、*number* を丸めてから形式設定して戻します。



number+0 演算が実行される場合と同様に、REXX の標準規則に従って、まず最初に *number* が丸められます。*number* だけを指定すると、この演算とまったく同じ結果になります。他のオプションを指定した場合、*number* は以下のように形式設定されます。

before と *after* のオプションは、それぞれ、結果の整数部および小数部の文字数を指定します。これらのオプションのいずれか、あるいは両方を省略すると、該当部分の必要に応じた文字数になります。

before が、数値の整数部 (負数の場合は、符号をプラス) を表すのに十分な大きさでない場合、エラーになります。*before* が該当部分に必要な数よりも大きい場合は、数値の左側にブランクが埋め込まれます。*after* が、この数値の小数部と同じサイズでない場合、この数値は、適合するように丸められます (あるいはゼロで拡張されます)。0 を指定すると、数値は整数に丸められます。

例

```
FORMAT('3',4)      -> ' 3'
FORMAT('1.73',4,0)  -> ' 2'
FORMAT('1.73',4,3)  -> ' 1.730'
FORMAT('-.76',4,1)  -> ' -0.8'
FORMAT('3.03',4)    -> ' 3.03'
FORMAT(' -12.73',,4) -> ' -12.7300'
FORMAT(' -12.73')   -> ' -12.73'
FORMAT('0.000')     -> '0'
```

最初の 3 つの引数は、すでに説明したとおりです。さらに、*expp* および *expt* は結果の指数部分を制御します。この指数部分は、デフォルトで、DIGITS および FORM という 現行の NUMERIC 設定値に従って形式設定されます。*expp* は、指数部分の桁数をセットします。デフォルトでは、必要なだけの桁数が使用されます (ゼロの場合もあります)。*expt* は、指数表記に使用するトリガー・ポイントをセットします。このデフォルトは、NUMERIC DIGITS の現行設定値です。

expp が 0 のときは、指数が与えられないため、この数値は、必要に応じて、ゼロが追加されて単純形式で表されます。*expp* が、指数を表すのに十分な大きさでなければ、エラーになります。

整数部または小数部に必要な桁数は、それぞれ *expt* を超えるか または *expt* の 2 倍を超えると、指数表記されます。*expt* が 0 ならば、指数が 0 でない限り、常に指数表記されます。(*expp* が 0 であれば、その値が *expt* の 0 値を指定変更します。) ゼロ以外の *expp* が指定されている場合に、指数が 0 になると、結果の指数部に対して *expp*+2 個のブランクが与えられます。指数が 0 の場合は *expp* を指定していないと、単純表記が使用されます。*expp* は 10 未満でなければなりません、その他の引数に関する制限はありません。

例:

```
FORMAT('12345.73',,,2,2) -> '1.234573E+04'
FORMAT('12345.73',,3,0)  -> '1.235E+4'
FORMAT('1.234573',,3,0)  -> '1.235'
FORMAT('12345.73',,,3,6) -> '12345.73'
FORMAT('1234567e5',,3,0) -> '123456700000.000'
```

FUZZ

FUZZ 関数は、NUMERIC FUZZ の現行設定値を戻します。

➡ FUZZ(—) ➡

NUMERIC 命令 [176 ページ](#)の『NUMERIC』を参照してください。

例

```
FUZZ()      ->    0      /* by default */
```

INDEX

INDEX 関数は、ある文字列 *needle* の、別の文字列 *haystack* 内での文字位置を戻します。あるいは、文字列 *needle* が見つからないか、ヌル・文字列である場合は、0 を戻します。

POS は、ある文字列が別の文字列内でどの位置に存在しているかを知るための優先組み込み関数です。 [212 ページ](#)の『POS (位置)』を参照してください。

➡ INDEX(— *haystack* — , — *needle* — , — *start* —) ➡

デフォルトでは、この検索は、*haystack* の最初の文字から開始されます (*start* の値は 1)。これは、異なる *start* 地点 (正の整数でなければなりません) を指定することによって、オーバーライドできます。

例

```
INDEX('abcdef', 'cd')      ->    3
INDEX('abcdef', 'xd')      ->    0
INDEX('abcdef', 'bc', 3)    ->    0
INDEX('abcabc', 'bc', 3)    ->    5
INDEX('abcabc', 'bc', 6)    ->    0
```

INSERT

INSERT 関数は、文字列 *new* が *length* の長さになるように埋め込みまたは切り捨てを行い、文字列 *target* の *n* 番目の文字の後に挿入します。

➡ INSERT(— *new* — , — *target* — , — *n* — , — *length* — , — *pad* —) ➡

➡ INSERT(— *new* — , — *target* — , — *n* — , — *length* — , — *pad* —) ➡

n のデフォルト値は 0 です。つまり、文字列の先頭の前に挿入されます。 *n* および *length* を指定する場合は、正の整数またはゼロにしなければなりません。 *n* がターゲット・文字列の長さより大きい場合は、文字列 *new* の前にも埋め込み文字が追加されます。 *length* のデフォルト値は *new* の長さです。 *length* が、文字列 *new* の長さよりも小さい場合、INSERT は、*new* を *length* の長さに切り捨てます。デフォルトの *pad* 文字は空白です。

例

```
INSERT(' ', 'abcdef', 3)      ->    'abc def'
INSERT('123', 'abc', 5, 6)     ->    'abc 123 '
INSERT('123', 'abc', 5, 6, '+') ->    'abc++123+++'
INSERT('123', 'abc')           ->    '123abc'
INSERT('123', 'abc', 5, '-')   ->    '123--abc'
```

JUSTIFY

JUSTIFY 関数は、両方のマージンに行末揃えするために、ブランクで区切られたワード間に *pad* 文字を追加することによって形式設定された *string* を戻します。

➡ JUSTIFY(— *string* — , — *length* — , — *pad* —) ➡

これは、幅 *length* に対して行われます (*length* は正の整数またはゼロでなければなりません)。デフォルトの *pad* 文字は空白です。

まず最初に行うのは、`SPACE(string)` が実行された場合と同様に余分な空白を除去する (すなわち、複数の空白が単一の空白に変換され、先行空白と末尾空白は除去されます) ことです。`length` が、変更されたストリングの幅よりも小さい場合、そのストリングは右側で切り捨てが行われ、末尾空白はすべて除去されます。次に、必要な長さになるように、追加の *pad* 文字が左から右に均等に追加され、*pad* 文字がワード間の空白に取って代わります。

例

```
JUSTIFY('The blue sky',14)      -> 'The blue sky'
JUSTIFY('The blue sky',8)       -> 'The blue'
JUSTIFY('The blue sky',9)       -> 'The blue'
JUSTIFY('The blue sky',9,'+')   -> 'The++blue'
```

LASTPOS (最後の位置)

LASTPOS 関数は、ある文字列 *needle* の最後のオカレンスの位置を、別の文字列 *haystack* で戻します。

➡ LASTPOS(— needle — , — haystack —) ➡

LASTPOS は、*needle* がヌル・ストリングであるか、見つからない場合は、0 を戻します。デフォルトでは、検索は *haystack* の最後の文字から始まり、逆方向にスキャンされます。これは、逆方向のスキャンが始まる地点である *start* を指定すると、指定変更できます。*start* は正の整数でなければなりません、*haystack* の長さの値より大きいかまたは省略されていると、LENGTH(*haystack*) がデフォルトになります。また、POS 関数 (212 ページの『POS (位置)』) も参照してください。

例

```
LASTPOS(' ','abc def ghi')      -> 8
LASTPOS(' ','abcdefghi')         -> 0
LASTPOS('xy','efgxyz')           -> 4
LASTPOS(' ','abc def ghi',7)     -> 4
```

LEFT

LEFT 関数は、*string* の左端の *length* 個の文字を含む、長さ *length* のストリングを戻します。

➤ LEFT(— *string* — , — *length* —) ➤

戻される文字列は、必要に応じて、右側で `pad` 文字が埋め込まれます (または切り捨てられます)。デフォルトの `pad` 文字は空白です。 `length` は、正の整数またはゼロでなければなりません。 `LEFT` 関数は、以下のものとまったく同じ働きをします。

➡ SUBSTR — (— *string* — , — 1 — , — *length* —) ➡

例

```
LEFT('abc d',8)      -> 'abc d  '
```

```
LEFT('abc d',8,'.')  -> 'abc d...'
```

```
LEFT('abc def',7)    -> 'abc de'
```

LENGTH

LENGTH 関数は、*string* の長さを戻します。

➡ LENGTH(— *string* —) ➡

例

```
LENGTH('abcdefgh')  -> 8
```

```
LENGTH('abc defg')   -> 8
```

```
LENGTH('')           -> 0
```

LINESIZE

LINESIZE 関数は、現行の端末行の長さ (SAY 命令を使用して表示される行を、言語処理プログラムが区切る点) を戻します。

➡ LINESIZE(—) ➡

LINESIZE は、以下の場合にデフォルト値 80 を戻します。

- 端末が接続されていない場合。出力がリダイレクトされている場合。

注: 端末が接続されているかどうかを調べるには、REXX/CICS コマンド `SCRNINFO` を指定してください (402 ページの『[SCRNINFO](#)』)。接続されている端末がない場合、画面の高さおよび幅の値は 0 です。

MAX (最大)

MAX 関数は、指定されたリストのうち最大の数値を、現行の NUMERIC 設定値に従って形式設定して戻します。

➡ MAX(— *number* —) ➡

実際の最大値: 最大 20 個の数値を指定でき、さらに多くの引数が必要な場合は、MAX への呼び出しをネストすることができます。

例

```
MAX(12,6,7,9)        -> 12
```

```
MAX(17.3,19,17.03)    -> 19
```

```
MAX(-7,-3,-4.3)       -> -3
```

```
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(20,21)) -> 21
```

MIN (最小)

MIN 関数は、指定されたリストのうち最小の数値を、現行の NUMERIC 設定値に従って形式設定して戻します。

➡ MIN(— *number* —) ➡

実際の最大値: 最大 20 個の数値を指定でき、さらに多くの引数が必要な場合は、MIN への呼び出しをネストすることができます。

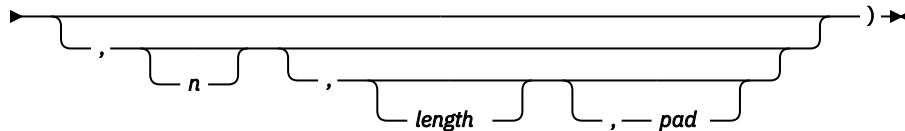
例

```
MIN(12,6,7,9)                -> 6
MIN(17.3,19,17.03)            -> 17.03
MIN(-7,-3,-4.3)               -> -7
MIN(21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,MIN(2,1)) -> 1
```

OVERLAY

OVERLAY 関数は、ストリング *target* を戻します。これは、*n* 番目の文字で始まり、ストリング *new* でオーバーレイされて、長さ *length* になるように埋め込みまたは切り捨てが行われます。

➡ OVERLAY(— *new* — , — *target* →



オーバーレイは、元の *target* ストリングの終わりを超えて拡張することがあります。*length* を指定する場合は、正の整数またはゼロでなければなりません。*length* のデフォルト値は *new* の長さです。*n* がターゲット・ストリングの長さを超えている場合、*new* ストリングの前に埋め込みが行われます。デフォルトの *pad* 文字は空白で、*n* のデフォルト値は 1 です。*n* を指定する場合は、正の整数でなければなりません。

例

```
OVERLAY(' ', 'abcdef', 3)      -> 'ab def'
OVERLAY(' ', 'abcdef', 3, 2)   -> 'ab. ef'
OVERLAY('qq', 'abcd')          -> 'qqcd'
OVERLAY('qq', 'abcd', 4)       -> 'abcqq'
OVERLAY('123', 'abc', 5, 6, '+') -> 'abc+123+++'
```

POS (位置)

POS 関数は、あるストリング *needle* が別のストリング *haystack* に出現する位置を戻します。

➡ POS(— *needle* — , — *haystack* — , — *start* —) ➡

POS は、*needle* がヌル・ストリングであるか、見つからない場合、あるいは *start* が *haystack* の長さより大きい場合は、0 を戻します。デフォルトでは、この検索は、*haystack* の最初の文字から開始されます (つまり、*start* の値が 1 になります)。これは、検索の開始地点である *start* (正の整数でなければなりません) を指定すると、指定変更できます。INDEX および LASTPOS 関数 (209 ページの『INDEX』 および 210 ページの『LASTPOS (最後の位置)』) も参照してください。

例

```
POS('day', 'Saturday')        -> 6
POS('x', 'abc def ghi')       -> 0
POS(' ', 'abc def ghi')       -> 4
POS(' ', 'abc def ghi', 5)     -> 8
```

QUEUED

QUEUED 関数は、この関数が呼び出された時点で外部データ・キューに残っている行数を戻します。行が残っていない場合には、PULL または PARSE PULL は端末入力バッファを読み取ります。待機中の端末入力がない場合には、コンソール読み取りになります。

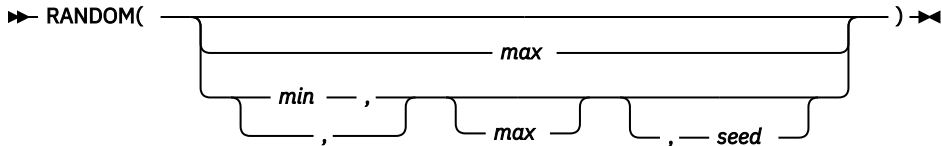
➡ QUEUED(—) ➡

例

```
QUEUED()    ->    5    /* Perhaps */
```

RANDOM

RANDOM 関数は、*min* から *max* までの範囲 (両方を含む) の、負でない整数の疑似乱数を戻します。



max または *min* (あるいはその両方) を指定する場合、*max* から *min* を引いた結果が 100000 を超えてはなりません。 *min* および *max* のデフォルトは、それぞれ、0 および 999 です。 結果が反復可能なシーケンスになるようにするには、3 番目の引数として特別な *seed* を使用します (注 213 ページの『1』で説明)。この *seed* は、0 から 999999999 までの正の整数でなければなりません。

例

```
RANDOM()      ->    305
RANDOM(5,8)    ->     7
RANDOM(2)      ->     0 /* 0 to 2 */
RANDOM(,1983)  ->   123 /* reproducible */
```

注:

1. 予測可能な一連の疑似乱数を得るには、RANDOM を何回か使用することになりますが、*seed* は初回にのみ指定します。例えば、6 面の無作為のさいころを 40 回投げるシミュレーションを行う場合は、以下のようにします。

```
sequence = RANDOM(1,6,12345) /* any number would */
                               /* do for a seed    */
do 39
  sequence = sequence RANDOM(1,6)
end
say sequence
```

数値は、可能な限り、行き当たりばったりに見えるように、最初の *seed* を使用して数学的に生成されます。プログラムを再び実行すると、同じシーケンスが生成されます。異なる初期 *seed* を使用すれば、ほぼ確実に異なるシーケンスが生成されます。 *seed* を指定しない場合は、RANDOM が最初に呼び出されたときに、時刻機構のマイクロ秒フィールドが *seed* として使用されます。したがって、プログラムを実行するたびに、ほぼ必ず、違う結果が戻される ことになります。

2. 乱数発生ルーチンは、プログラム全体にわたってグローバルです。 現行の *seed* が、内部ルーチン呼び出しをまたがって保管されることはありません。

REVERSE

REVERSE 関数は、*string* を逆の順序にして戻します。

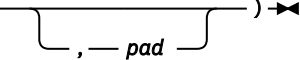
➡ REVERSE(— *string* —) ➡

例

```
REVERSE('ABC.') -> '.cBA'
REVERSE('XYZ ') -> ' ZYX'
```

RIGHT

RIGHT 関数は、*string* の右端の *length* 個の文字を含む長さ *length* のストリングを返します。

➡ RIGHT(— *string* — , — *length* —) ➡


戻されるストリングの左側には、必要に応じて、*pad* 文字が埋め込まれます (あるいは切り捨てられます)。デフォルトの *pad* 文字は空白です。 *length* は、正の整数またはゼロでなければなりません。

例

```
RIGHT('abc d',8)    -> ' abc d'
RIGHT('abc def',5)  -> 'c def'
RIGHT('12',5,'0')   -> '00012'
```

SIGN

SIGN 関数は、*number* の符号を示す数値を返します。 *number*+0 演算が実行される場合と同様に、REXX の標準規則に従って、まず最初に *number* が丸められます。

➡ SIGN(— *number* —) ➡

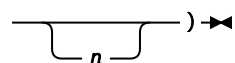
SIGN は、*number* が 0 未満であれば -1 を返し、0 であれば 0 を返し、0 より大きければ 1 を返します。

例

```
SIGN('12.3')      -> 1
SIGN(' -0.307')   -> -1
SIGN(0.0)          -> 0
```

SOURCELINE

SOURCELINE 関数は、*n* を省略した場合はプログラム内の最終行の行番号を返します。 *n* を指定した場合は、プログラム内の *n* 番目の行を返します。

➡ SOURCELINE(— *n* —) ➡



n を指定する場合は、正の整数でなければならず、プログラム内の最終行の番号を超えてはなりません。

例

```
SOURCELINE()      -> 10
SOURCELINE(1)      -> '/* This is a 10-line REXX program */'
```

SPACE

SPACE 関数は、空白で区切られたワードを、各ワード間に *n pad* 文字を入れて *string* に返します。

➡ SPACE(— *string* — , — *n* — , — *pad* —) ➡


n を指定する場合は、正の整数またはゼロでなければなりません。 0 の場合、空白はすべて削除されます。 先行空白および末尾空白は常に削除されます。 *n* のデフォルトは 1 で、デフォルトの *pad* 文字は空白です。

例

```
SPACE('abc def ') -> 'abc def'
SPACE(' abc def',3) -> 'abc def'
SPACE('abc def ',1) -> 'abc def'
SPACE('abc def ',0) -> 'abcdef'
SPACE('abc def ',2,'+') -> 'abc++def'
```

STORAGE

224 ページの『REXX/CICS で提供される外部関数』を参照してください。

STRIP

STRIP 関数は、指定された *option* に基づいて、*string* から先行文字または末尾の文字 (あるいはその両方) を除去した結果を返します。

➡ STRIP(— *string* — , — *option* — , — *char* —) ➡

有効なオプションを以下に示します。大文字だけが必要であり、後続の文字はすべて無視されます。

両方

先行文字および後続文字の両方を、*string* から除去します。これはデフォルトです。

Leading

先行文字を *string* から除去します。

Trailing

後続文字を *string* から除去します。

3 番目の引数 *char* は、削除される文字を指定するもので、デフォルトはブランクです。 *char* を指定する場合は、正確に 1 文字の長さになければなりません。

例

```
STRIP(' ab c ') -> 'ab c'
STRIP(' ab c ', 'L') -> 'ab c'
STRIP(' ab c ', 'T') -> ' ab c'
STRIP('12.7000', 0) -> '12.7'
STRIP('0012.700', 0) -> '12.7'
```

SUBSTR (サブストリング)

SUBSTR 関数は、*n* 番目の文字から始まる *string* のサブストリングを返します。このサブストリングは、長さが *length* で、必要に応じて *pad* が埋め込まれています。

➡ SUBSTR(— *string* — , — *n* — , — *length* — , — *pad* —) ➡

n は、正の整数でなければなりません。 *n* が LENGTH(*string*) より大きい場合は、埋め込み文字だけが返されます。

length を省略すると、ストリングの残りの部分が返されます。デフォルトの *pad* 文字はブランクです。

例

```
SUBSTR('abc',2) -> 'bc'
SUBSTR('abc',2,4) -> 'bc '
SUBSTR('abc',2,6,'.') -> 'bc...'
```

注: 状況により、特に、2 つ以上のサブストリングを 1 つのストリングから抽出する場合は、サブストリングを選択するのに、解析テンプレートの定位置 (数字) パターンを使用した方が便利な場合もあります。LEFT および RIGHT 関数も参照。

SUBWORD

SUBWORD 関数は、 n 番目のワードで始まる *string* のサブストリングを戻します。このサブストリングは、
 ブランクで区切られた最大 *length* 個のワードからなります。

SUBWORD(— *string* — , — *n* — , — *length* —)

`n` は、正の整数でなければなりません。 `length` を省略すると、デフォルトで `string` 内の残りのワード数に設定されます。 戻されるストリングには、先行空白も末尾空白もありますが、選択されたワードの間の空白はすべて含まれています。

例

```
SUBWORD('Now is the time',2,2)  -> 'is the'
SUBWORD('Now is the time',3)    -> 'the time'
SUBWORD('Now is the time',5)    -> ''
```

SYMBOL

SYMBOL 関数は、*name* で指定されたシンボルの状態を戻します。

►► **SYMBOL(— *name* —)** ◄◄

SYMBOL は、*name* が有効な REXX シンボルでない場合は、BAD を戻します。SYMBOL は、それが変数の名前 (つまり、値が割り当てられたシンボル) である場合は、VAR を戻します。それ以外の場合には、LIT を戻します。これは、それが定数シンボルであるか、またはまだ値が割り当てられていないシンボル (つまり、リテラル) のどちらかであることを示します。

REXX 式内のシンボルと同様、*name* 内の小文字が大文字に変換され、可能であれば、複合名内で置換が行われます。

注: `name` が関数に渡される前に置換が行われなくするために、それをリテラル・ストリングとして指定します (あるいはそれが式から導出される必要があります)。

例

```

/* following: Drop A.3; J=3 */
SYMBOL('J')      -> 'VAR'
SYMBOL(J)         -> 'LIT' /* has tested "3" */
SYMBOL('a.j')    -> 'LIT' /* has tested A.3 */
SYMBOL(2)         -> 'LIT' /* a constant symbol */
SYMBOL('*')      -> 'BAD' /* not a valid symbol */

```

TIME

TIME 関数は、現地時間を 24 時間表示形式で戻します。デフォルトの形式は、hh:mm:ss (時、分、秒) で、例えば、04:41:37 となります。

► TIME() ◄

option

以下のオプションを使用して、別の形式で時刻を表示したり、経過時間クロックにアクセスできるようにしたりすることができます。太文字だけが必要であり、後続の文字はすべて無視されます。

Civil

常用フォーマットの時刻: hh:mmxx を戻します。時は 1 から 12 までの値、分は 00 から 59 までの値になります。分は、直後に、am または pm という文字が続きます。これにより、午前中の時刻 (深夜 12 時から午前 11:59 までが 12:00am から 11:59am と表示) と、正午および午後の時刻 (正午 12 時から午後 11:59 までは、12:00pm から 11:59pm と表示) が区別されます。時間には、先行ゼロがありません。分のフィールドは、他の TIME 結果との整合性のために、現在の分 (最も近い分ではなく) を示します。

Elapsed

ssssssss.uuuuuu を戻します。この値は、経過時間クロック (後述します) が開始またはリセットされて以降経過した「秒.マイクロ秒」を示す数です。この数値には、先行ゼロやブランクは含まれず、NUMERIC DIGITS の設定値の影響は受けません。小数部の桁数は、常に 6 桁です。

Hours

深夜 12 時以降の時間数を表す最大 2 桁の文字を hh (結果が 0 のとき以外は先行ゼロまたはブランクなし) というフォーマットで戻します。

Long

時間を hh:mm:ss.uuuuuu のフォーマット (uuuuuu は、秒の端数で、マイクロ秒単位です)。結果の最初の 8 文字は Normal フォーマットの場合と同じ規則に従い、小数部は常に 6 桁です。

Minutes

深夜 12 時以降の分数を表す最大 4 文字を mmmm のフォーマットで (結果が 0 の場合を除いて、先行ゼロまたはブランクなし)。

正常

前述のように、デフォルトの hh:mm:ss というフォーマットで時刻を戻します。時間の値は 00 から 23 までの範囲であり、分および秒の値は 00 から 59 までの範囲です。これらの値はすべて、常に、2 桁です。秒数の端数は無視されます (時間は切り上げられません)。これがデフォルトです。

リセット

ssssssss.uuuuuu を戻します。これは、経過時間クロック (後述します) が開始またはリセットされて以降経過した「秒.マイクロ秒」を示す数であり、経過時間クロックを 0 にリセットします。この数値には、先行ゼロやブランクは含まれず、NUMERIC DIGITS の設定値の影響は受けません。小数部の桁数は、常に 6 桁です。

Seconds

深夜 12 時以降の秒数を示す最大 5 桁の文字を sssss というフォーマットで戻します (結果が 0 の場合を除いて、先行ゼロまたはブランクなし)。

注: ある文節内で DATE または TIME を初めて呼び出したときに、タイム・スタンプが作成されます。それ以降、このタイム・スタンプが、その文節内のこれらすべての関数呼び出しに使用されます。したがって、DATE 関数と TIME 関数のいずれか、あるいはその両方を 1 つの式または文節内で複数回呼び出しても、相互に必ず整合性が保たれます。

実際の最大値: 経過時間の秒数が 9 桁を超える (31.6 年を超えたことになる) と、エラーになります。

例

以下の例では、時刻を 4:54 p.m. としています。

```
TIME()      -> '16:54:22'
TIME('C')   -> '4:54pm'
TIME('H')   -> '16'
TIME('L')   -> '16:54:22.123456' /* Perhaps */
TIME('M')   -> '1014'           /* 54 + 60*16 */
TIME('N')   -> '16:54:22'
TIME('S')   -> '60862' /* 22 + 60*(54+60*16) */
```

経過時間クロック

TIME 関数を使用して、実 (経過) 時間間隔を測定することができます。TIME('E') または TIME('R') へのプログラムにおける最初の呼び出しで、経過時間クロックが開始され、どちらの呼び出しに対しても 0

を戻します。そのあと、TIME('E') および TIME('R') への呼び出しがあると、最初の呼び出し以降、または TIME('R') への最後の呼び出し以降の経過時間が戻されます。

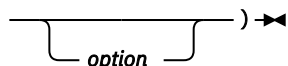
クロックは、内部ルーチン呼び出し全体で保管されます。つまり、内部ルーチンは、その呼び出し側が開始した時間クロックを継承します。内部ルーチンがクロックをリセットしても、呼び出し側が実行するタイミングには影響ありません。経過時間クロックの例を、以下に示します。

```
time('E')    ->    0          /* The first call */
/* pause of one second here */
time('E')    ->    1.002345  /* or thereabouts */
/* pause of one second here */
time('R')    ->    2.004690  /* or thereabouts */
/* pause of one second here */
time('R')    ->    1.002345  /* or thereabouts */
```

注：単一文節内での時刻の整合性については、前述の注を参照してください。経過時間クロックは、TIME および DATE に対する他の呼び出しと同期されるため、単一文節内で経過時間クロックを複数回呼び出ししても、必ず、同じ結果が戻されます。同じ理由により、通常の TIME/DATE の 2 つの結果の時間間隔を、経過時間クロックを使用して正確に計算することができます。

TRACE

TRACE 関数は、現在有効になっているトレース・アクションを戻し、オプションで、設定値を更新します。

➡ TRACE() ➡

option を指定すると、トレース設定値が選択されます。これは有効な接頭部である？か！、あるいは TRACE 命令に関連した英字オプション (すなわち、A、C、E、F、I、L、N、O、R、あるいは S で始まる) のいずれか、またはその両方です。

TRACE 命令とは異なり、TRACE 関数は、対話式デバッグがアクティブであっても、トレース・アクションを更新します。さらに、TRACE 命令とは異なり、*option* を数値にすることはできません。

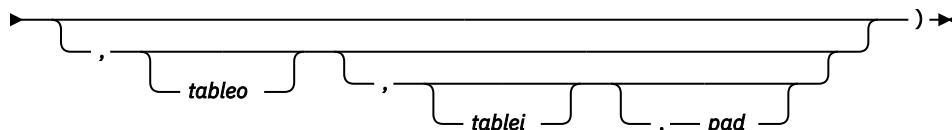
例

```
TRACE()      ->    '?R' /* maybe */
TRACE('O')   ->    '?R' /* also sets tracing off */
TRACE('?I')   ->    'O'  /* now in interactive debug */
```

TRANSLATE

TRANSLATE 関数は、それぞれの文字を別の文字に変更するか、または未変換のまま、*string* を戻します。この関数を使用して、*string* 内の文字を並べ替えることもできます。

➡ TRANSLATE() ➡



出力テーブルは *tableo* であり、入力変換テーブルは *tablei* です。TRANSLATE は、*string* 内の各文字を *tablei* から探します。文字が見つかったら、*tableo* 内の対応する文字が結果のストリングに入ります。*tablei* 内の文字が重複する場合には、最初 (左端) の文字が使用されます。文字が見つからない場合には、*string* 内の元の文字が使用されます。結果のストリングの長さは、必ず *string* と同じ長さになります。

テーブルは、どのような長さでも構いません。変換テーブルを指定せず、さらに *pad* も省略した場合、*string* は単に大文字に変換されます (つまり、小文字の a から z が大文字の A から Z に変換されます)。また、変換テーブルは指定しないが *pad* は指定した場合は、言語処理プログラムはストリング全体を *pad* 文字に変換します。*tablei* のデフォルトは XRange('00'x, 'FF'x)、*tableo* のデフォルトはヌル・ストリン

グとなり、必要に応じて *pad* で埋め込まれるか、切り捨てられます。デフォルトの *pad* は、ブランクです。

例

```
TRANSLATE('abcdef')      -> 'ABCDEF'
TRANSLATE('abbc','&','b') -> 'a&&c'
TRANSLATE('abcdef','12','ec') -> 'ab2d1f'
TRANSLATE('abcdef','12','abcd','.') -> '12..ef'
TRANSLATE('APQRV','PR') -> 'A Q V'
TRANSLATE('APQRV',X'00'X,'Q')) -> 'APQ '
TRANSLATE('4123','abcd','1234') -> 'dabc'
```

注：最後の例は、TRANSLATE 関数を使用して、ストリング内の文字の順序を並べ替える方法を示しています。この例では、2 番目の引数として指定された 4 文字のストリングの最後の文字が、ストリングの先頭に移動されます。

TRUNC (切り捨て)

TRUNC 関数は、*number* の整数部および *n* 桁の小数部を戻します。

➡ TRUNC(— *number* — , — *n* —) ➡

n のデフォルトは 0 で、小数点なしの整数を戻します。*n* を指定する場合は、正の整数またはゼロでなければなりません。*number*+0 演算が実行される場合と同様に、REXX の標準規則に従って、まず最初に *number* が丸められます。数値は、次に、小数部が *n* 桁に切り捨てられます (指定した長さによっては、後ろに 0 が追加されます)。結果が、指数形式になることはありません。

注：*number* は、必要に応じて、この関数によって処理される前に、NUMERIC DIGITS の現行設定値に従って丸められます。

例

```
TRUNC(12.3)      -> 12
TRUNC(127.09782,3) -> 127.097
TRUNC(127.1,3)   -> 127.100
TRUNC(127,2)     -> 127.00
```

USERID

USERID は、ユーザーが CICS にサインオンしている場合は CICS サインオン・ユーザー ID を戻し、そうでない場合は CICS 領域のデフォルトのユーザー ID を戻します (ただし、CICS システム・プログラマーがデフォルト・ユーザー ID を指定しておく必要があります)。

➡ USERID(—) ➡

ユーザー ID は、戻される値の長さが必ず 8 バイトになるように、右側にブランクが埋め込まれます。

例

```
USERID() -> 'ARTHUR' /* Maybe */
```

VALUE

VALUE 関数は、*name* (多くの場合は動的に構成されます) が表すシンボルの値を戻し、オプションでそれに新しい値を割り当てます。

➡ VALUE(— *name* — , — *newvalue* — , — *selector* —) ➡

デフォルトでは、VALUE は現在の REXX 変数環境を参照しますが、*selector* を指定する場合、この値は RLS でなければなりません。選択子 RLS が指定された場合、演算対象となる変数は、REXX 変数ではなく、REXX List System (RLS) 変数になります。この関数を使用して REXX 変数を参照する場合、*name* は有効な REXX シンボルでなければなりません。(SYMBOL 関数を使用してこれを確認できます。) *name* 中の小文字は大文字に変換されます。可能ならば、複合名 (154 ページの『複合記号』参照) の置換が行われます。

newvalue を指定すると、指定した変数にこの新しい値が割り当てられます。これは、戻される結果には影響しません。つまり、関数は、新しい割り当てが行われる前の状態の *name* の値を戻します。

例

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)      -> 'A3' /* looks up A3      */
VALUE('a'k|k)     -> '7'  /* looks up A33     */
VALUE('fred')     -> 'K'  /* looks up FRED    */
VALUE(fred)       -> '3'  /* looks up K       */
VALUE(fred,5)     -> '3'  /* looks up K and   */
                  /* then sets K=5   */
VALUE(fred)       -> '5'  /* looks up K       */
VALUE('LIST.'k)   -> 'Hi' /* looks up LIST.5  */
```

以下の例は、RLS 変数に格納された REXX 変数 FRED の VALUE を戻します。

```
/* REXX EXEC - ASSIGN FIND VALUE OF FRED */
FRED = 7
'RLS VARPUT FRED ¥USERS¥userid¥'
X = VALUE(FRED,,RLS)
SAY X
/* X now = 7                               */
```

注:

1. VALUE 関数が初期設定されていない REXX 変数を参照する場合は、常に、変数のデフォルト値が戻されます。NOVALUE 条件は発生しません。RLS 変数への参照で NOVALUE が発生することはありません。
2. *name* を 1 つのリテラル・ストリングとして指定し、*newvalue* と *selector* を省略した場合、記号は定数となるので、通常は、引用符間のストリングが関数呼び出し全体を置き換えることになります。例えば、`fred=VALUE('k');` は、NOVALUE 条件がトラップされていないかぎり、割り当て `fred=k;` と同じです。249 ページの『第 22 章 条件および条件トラップ』を参照してください。

VERIFY

VERIFY 関数は、デフォルトでは、*string* が *reference* の文字のみで構成されているかどうかを示す数値を戻します。

➡ VERIFY(— *string* — , — *reference* — , — *option* — , — *start* —) ➡

VERIFY は、*string* 内のすべての文字が *reference* にある場合は 0 を戻し、そうでない場合は、*string* 内の文字のうち *reference* にない最初の文字の位置を戻します。

option は、Nomatch (デフォルト) または Match のいずれかにすることができます。(大文字の文字だけが必要です。それに続く文字はすべて無視されます。この部分は、通常どおり、大文字でも小文字でも指定できます。) Match を指定した場合、この関数は *reference* 内にある *string* 内の最初の文字の位置を戻しますが、そのような文字が見つからなければ、0 を戻します。

start のデフォルトは 1 なので、探索は *string* の最初の文字から開始されます。この開始位置は、異なる *start* 位置 (正の整数でなければなりません) を指定してオーバーライドできます。

string がヌルの場合、3 番目の引数の値に関係なく、この関数は 0 を戻します。同様に、*start* が LENGTH(*string*) を超えている場合も、この関数は 0 を戻します。*reference* がヌルの場合、この関数は、Match が指定されていれば 0 を戻し、指定されていなければ *start* の値を戻します。

例

```
VERIFY('123','1234567890')      -> 0
VERIFY('123','1234567890')      -> 2
VERIFY('AB4T','1234567890')     -> 1
VERIFY('AB4T','1234567890','M') -> 3
VERIFY('AB4T','1234567890','N') -> 1
VERIFY('1P3Q4','1234567890',,3) -> 4
VERIFY('123',,,N,2)              -> 2
VERIFY('ABCDE',,,3)              -> 3
VERIFY('AB3CD5','1234567890','M',4) -> 6
```

WORD

WORD 関数は、*string* 内の空白で区切られた *n* 番目のワードを返します。*string* 内のワード数が *n* よりも少ない場合は、ヌル・ストリングを返します。

➡ WORD(— *string* — , — *n* —) ➡

n は、正の整数でなければなりません。この関数は、SUBWORD (*string*, *n*, 1) とまったく同等です。

例

```
WORD('Now is the time',3)      -> 'the'
WORD('Now is the time',5)      -> ''
```

WORDINDEX

WORDINDEX 関数は、*string* 内の空白で区切られた *n* 番目のワードの最初の文字の位置を返すか、あるいは、*string* 内のワード数が *n* より少ない場合には 0 を返します。

➡ WORDINDEX(— *string* — , — *n* —) ➡

n は、正の整数でなければなりません。

例

```
WORDINDEX('Now is the time',3)  -> 8
WORDINDEX('Now is the time',6)  -> 0
```

WORDLENGTH

WORDLENGTH 関数は、*string* 内の空白で区切られた *n* 番目のワードの長さを返すか、あるいは、*string* 内のワード数が *n* より少ない場合には 0 を返します。

➡ WORDLENGTH(— *string* — , — *n* —) ➡

n は、正の整数でなければなりません。

例

```
WORDLENGTH('Now is the time',2) -> 2
WORDLENGTH('Now comes the time',2) -> 5
WORDLENGTH('Now is the time',6) -> 0
```

WORDPOS (ワード位置)

WORDPOS 関数は、*string* 内で見つかった *phrase* の最初のワードのワード番号を戻すか、あるいは *phrase* 内にワードがない場合や *phrase* が見つからない場合には、0 を戻します。

➡ WORDPOS(— *phrase* — , — *string* —) ➡

phrase または *string* 内のワード間にある複数の空白は、比較の際には 1 個の空白として扱われますが、それ以外については、ワード同士が正確に一致しなければなりません。

デフォルトを使用すると、*string* 内の最初のワードから探索が開始されます。 *start* を指定すれば、この開始位置を変更することができます。 *start* は、探索を開始するワードであり、正の数でなければなりません。

例

```
WORDPOS('the','now is the time')      -> 3
WORDPOS('The','now is the time')      -> 0
WORDPOS('is the','now is the time')   -> 2
WORDPOS('is the','now is the time')   -> 2
WORDPOS('is time','now is the time')  -> 0
WORDPOS('be','To be or not to be')     -> 2
WORDPOS('be','To be or not to be',3)  -> 6
```

WORDS

WORDS 関数は、*string* 内の空白で区切られたワードの数を返します。

►► WORDS(— *string* —) ◄◄

例

```
WORDS('Now is the time')    ->    4
WORDS(' ')                  ->    0
```

XRANGE (16 進数範囲)

XRANGE 関数は、*start* 値から *end* 値までの範囲内 (これらの値も含みます) にある すべての有効な 1 バイト・エンコード (昇順) からなるストリングを戻します。

➡ **XRANGE(** start **,** end **)** ➡

start のデフォルト値は '00'x であり、*end* のデフォルト値は 'FF'x です。*start* が *end* より大きい場合、値は、'FF'x から '00'x ヘラップします。指定する場合、*start* と *end* は単一文字でなければなりません。

例

```

X RANGE('a','f')      -> 'abcdef'
X RANGE('03'x,'07'x)   -> '0304050607'x
X RANGE('04'x)         -> '0001020304'x
X RANGE('i','j')       -> '89A8B8C8D8E8F9091'x  /* EBCDIC */
X RANGE('FE'x,'02'x)   -> 'FEFF000102'x

```

X2B (16 進数から 2 進数へ)

この関数は、`hexstring`を2進数に変換したものを表すstringを、文字形式で戻します。

►► X2B(— *hexstring* —) ◄◄

hexstring は、16 進文字のストリングです。この長さは任意です。16 進文字はそれぞれ、4 つの 2 進数のストリングに変換されます。読みやすくするために、オプションで、*hexstring* にブランクを入れることができます (追加できる個所はバイト境界のみで、先頭や末尾には入れられません)。これらのブランクは無視されます。

戻されるストリングは、長さが 4 の倍数で、ブランクは含みません。

hexstring がヌルの場合、この関数はヌル・ストリングを戻します。

例

```
X2B('C3')      -> '11000011'
X2B('7')        -> '0111'
X2B('1 C1')     -> '000111000001'
```

X2B を関数 D2X および C2X と組み合わせて、数値または文字ストリングを 2 進数形式に変換することができます。例えば、次のようになります。

```
X2B(C2X('C3'x)) -> '11000011'
X2B(D2X('129')) -> '10000001'
X2B(D2X('12'))  -> '1100'
```

X2C (16 進数から文字へ)

X2C 関数は、*hexstring* を文字に変換したものを表す、文字フォーマットのストリングを戻します。

➡ X2C(— *hexstring* —) ➡

戻されるストリングは、元の *hexstring* の半分のバイト数です。*hexstring* は任意の長さにすることができます。必要に応じて、偶数の 16 進数字になるように先行 0 が埋め込まれます。

読みやすくするために、オプションで、*hexstring* にブランクを入れることができます (追加できる個所はバイト境界のみで、先頭や末尾には入れられません)。これらのブランクは無視されます。

hexstring がヌルの場合、この関数はヌル・ストリングを戻します。

例

```
X2C('F7F2 A2') -> '72s' /* EBCDIC */
X2C('F7f2a2') -> '72s' /* EBCDIC */
X2C('F')       -> ' ' /* '0F' is unprintable EBCDIC */
```

X2D (16 進数から 10 進数へ)

X2D 関数は、*hexstring* の 10 進表記を戻します。

➡ X2D(— *hexstring* — , — *n* —) ➡

hexstring は、16 進文字のストリングです。結果を整数として表せない場合には、エラーになります。つまり、結果の桁数は、NUMERIC DIGITS の現行設定値を超えてはなりません。

読みやすくするために、オプションで、*hexstring* にブランクを入れることができます (追加できる個所はバイト境界のみで、先頭や末尾には入れられません)。これらのブランクは無視されます。

hexstring がヌルの場合、この関数は 0 を戻します。

n を指定しないと、*hexstring* は、符号なしの 2 進数として処理されます。

実際の最大値: 入力ストリングの 16 進文字は、最終結果を出すうえで有効な 500 文字を超えてはなりません。先行符号文字 (0 および F) は、この合計数にはカウントされません。

例

```
X2D('0E')      -> 14
X2D('81')      -> 129
X2D('F81')     -> 3969
X2D('FF81')    -> 65409
X2D('c6 f0'X)  -> 240      /* EBCDIC */
```

n を指定すると、ストリングは n 桁の 16 進数で表した符号付きの数値と見なされます。左端のビットがオフであれば、正の数値となります。そうでなければ、2 の補数表記にした負の数値になります。いずれにしても、これは整数に変換されるので、負になる場合があります。 n が 0 の場合、この関数は 0 を返します。

必要に応じて、*hexstring* は、0 文字で左側が埋め込まれるか (“符号拡張” なしであることに注意)、または n 文字になるまで左側が切り捨てられます。例えば、次のようになります。

```
X2D('81',2)    -> -127
X2D('81',4)    -> 129
X2D('F081',4)  -> -3967
X2D('F081',3)  -> 129
X2D('F081',2)  -> -127
X2D('F081',1)  -> 1
X2D('0031',0)  -> 0
```

REXX/CICS で提供される外部関数

REXX/CICS 環境には、追加の外部関数が用意されています。

STORAGE

STORAGE 関数は、*address* で始まるユーザーのメモリーから *length* バイトを戻します。

重要: これは、許可関数です。REXX ユーザーは STORAGE 関数を使用して、CICS 領域の 31 ビット仮想記憶域の表示または変更 (あるいはその両方) を行えます。STORAGE 関数は、許可 EXEC から、または許可ユーザーでのみ正常に呼び出すことができます。

➡ STORAGE(— *address* — , — *length* — , — *data* —) ➡

length は、10 進数です。デフォルトは、1 バイトです。*address* は、16 進数です。*address* の最高位ビットは無視されます。必ず 31 ビット・アドレスを指定してください。そうしないと、結果が予測不能になる可能性があります。64 ビット・アドレスは指定できません。

data を指定した場合は、古い値が取得された後に、*address* で始まるストレージが *data* で上書きされます (*length* 引数はこの上書きに影響を及ぼしません)。

注: STORAGE 関数は 31 ビット・アドレスに対して機能します。STORAGE 関数は 64 ビット・アドレスに対して機能しません。

例

```
/* The following results vary from system to system. */
STORAGE(200000,32)
/* This returns 32 bytes of storage at hex address 200000 as a result. */
```

SYSSBA

関数

➡ SYSSBA(— *row* — , — *col* —) ➡

SYSSBA は、screen *row,col* をセット・バッファ・アドレス (SBA) に変換します。

row

画面の一番上から数えた行番号を指定します。

col

列番号 (画面の左から数えます) を指定します。

注: SYSSBA 関数は、各呼び出しで端末モデルを照会し、これを使用して SBA 計算を端末タイプに調整します。

例

```
x = SYSSBA(10,20)
```

この例は、画面行 10、列 20 の 3 バイトのセット・バッファー・アドレスを REXX 変数 x に戻します。

第 20 章 構文解析

解析とは、ソース・ストリング内のデータを分割し、テンプレートで指定された変数にそれぞれ割り当てることです。

構文解析命令は、ARG、PARSE、および PULL です。[164 ページの『ARG』](#)、[179 ページの『PARSE』](#)、および [182 ページの『PULL』](#) を参照してください。

解析されるデータが、ソース・ストリングです。テンプレートは、ソース・ストリングの区切り方を指定するモデルです。最も単純な種類のテンプレートは、変数名のリストだけから成るものです。以下に例を示します。

```
variable1 variable2 variable3
```

この種のテンプレートは、ソース・ストリングを空白で区切られたワードに分解します。より複雑なテンプレートになると、変数名の他にパターンも入っています。

ストリング・パターン

ソース・ストリングをどこで区切るかを指示する、ソース・ストリング内の突き合わせ文字です。[229 ページの『ストリング・パターンを含むテンプレート』](#) を参照してください。

定位置パターン

ソース・ストリングをどこで区切るかを、文字位置によって指示します。[230 ページの『定位置\(数値\)パターンを含むテンプレート』](#) を参照してください。

解析は、基本的には 2 つのステップをとります。

1. パターンを基に、ソース・ストリングを適切なサブストリングに分解します。
2. 各サブストリングをワードに分解します。

いくつかのワードに構文解析するための単純形式のテンプレート

構文解析の命令は以下のとおりです。

```
parse value 'time and tide' with var1 var2 var3
```

この命令でのテンプレートは、var1 var2 var3 です。構文解析するデータは、キーワード PARSE VALUE と キーワード WITH の間にあるもの、すなわち、ソース・ストリング time and tide です。構文解析が行われると、ソース・ストリングは、空白で区切られたいくつかのワードに分割され、以下のようなテンプレートで指名された変数に割り当てられます。

```
var1='time'  
var2='and'  
var3='tide'
```

この例では、解析されるソース・ストリングはリテラル・ストリングの time and tide です。次の例では、ソース・ストリングは変数です。

```
/* PARSE VALUE using a variable as the source string to parse */  
string='time and tide'  
parse value string with var1 var2 var3          /* same results */
```

PARSE VALUE は、ソース・ストリングにある小文字の a から z を大文字の A から Z には変換しません。文字を大文字に変換したいときは、PARSE UPPER VALUE を使用してください。構文解析命令が大文字小文字の区別に与える影響が、[234 ページの『UPPER の使用』](#) に要約されているので参照してください。

構文解析命令はすべて、ソース・ストリングの各部分をテンプレートで指定された変数に割り当てます。ソース・ストリングの性質や起点に相違があるため、さまざまな構文解析命令があります。[234 ページの『解析命令の要約』](#) を参照してください。

PARSE VAR 命令は、構文解析するソース・ストリングが常に変数である点を除き、PARSE VALUE とよく似ています。PARSE VAR では、キーワード PARSE VAR の後にソース・ストリングが入っている変数の名前

を続けます。次の例では、変数 `stars` にソース・ストリングが入っています。テンプレートは、`star1 star2 star3` です。

```
/* PARSE VAR example */
stars='Sirius Polaris Rigil'
parse var stars star1 star2 star3          /* star1='Sirius' */
                                           /* star2='Polaris' */
                                           /* star3='Rigil' */
```

テンプレート内のすべての変数に新規に値が入ります。ソース・ストリング内のワード数よりもテンプレート内の変数の方が多い場合、余った変数にはヌル (空) 値が入ります。これは、すべての構文解析にあてはまります。すなわち、単純テンプレートを使用したワードへの構文解析の場合や、パターンを含むテンプレートを使用した構文解析の場合にもあてはまります。以下に、ワードへの構文解析の使用例を示します。

```
/* More variables in template than (words in) the source string */
satellite='moon'
parse var satellite Earth Mercury          /* Earth='moon' */
                                           /* Mercury='' */
```

テンプレートにある変数よりもソース・ストリングにあるワードの方が多い場合は、テンプレートにある最後の変数が残りのすべてのデータを受け取ります。以下に例を挙げます。

```
/* More (words in the) source string than variables in template */
satellites='moon Io Europa Callisto...'
parse var satellites Earth Jupiter         /* Earth='moon' */
                                           /* Jupiter='Io Europa Callisto...'*/
```

ワードに構文解析すると、先行空白と末尾空白は、ワードが変数に割り当てられる前に各ワードから削除されます。ただし、最後の変数に割り当てられるワードまたはワード・グループの場合は、例外となります。テンプレート内の最後の変数には、超過分の先行空白と後続空白が付いたまま、残りのデータが入ります。以下に例を挙げます。

```
/* Preserving extra blanks */
solar5='Mercury Venus Earth Mars Jupiter '
parse var solar5 var1 var2 var3 var4
/* var1 = 'Mercury' */
/* var2 = 'Venus' */
/* var3 = 'Earth' */
/* var4 = ' Mars Jupiter ' */
```

ソース・ストリング内の `Earth` には、先行空白が 2 個付いています。解析では、この両方 (ワード区切り文字の空白と余分にある空白) を除去してから、`var3='Earth'` の割り当てを行います。`Mars` には、先行空白が 3 個付いています。解析により、ワードを区切る空白が 1 個除去されますが、あと 2 個の先行空白は残されます。`Mars` と `Jupiter` の間の 5 個のすべての空白、および `Jupiter` の後の末尾空白は両方ともそのまま残されます。

テンプレート内に変数が 1 個しかない場合、構文解析で空白は除去されません。以下に例を示します。

```
parse value ' Pluto ' with var1          /* var1=' Pluto '*/
```

ピリオドはプレースホルダー

テンプレート内のピリオドはプレースホルダーです。ピリオドは変数名の代わりに使用されますが、データは受け取りません。プレースホルダーを使用することにより、不要な変数のオーバーヘッドが省かれます。これには、次のような利点があります。

- 変数リスト内の「仮変数」として使用する
- スtringの終わりにある不要な情報を処理する

最初の例のピリオドがプレースホルダーです。隣接するピリオドは、必ず、スペースで区切ります。そうしないと、エラーになります。

```
/* Period as a placeholder */
stars='Arcturus Betelgeuse Sirius Rigil' */
```

```

parse var stars . . brightest .          /* brightest='Sirius' */

/* Alternative to period as placeholder          */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars drop junk brightest rest  /* brightest='Sirius' */

```

ストリング・パターンを含むテンプレート

ストリング・パターンがソース・ストリング内の文字と突き合わされて、ソース・ストリングの区切り位置が示されます。

ストリング・パターンは、リテラルまたは変数にすることができます。

リテラル・ストリング・パターン

引用符で囲んだ1文字または複数の文字。

変数ストリング・パターン

括弧に入った変数で、左括弧の前に、正符号(+)、負符号(-)、または等号(=)が付いていないもの。詳しくは、[233 ページの『変数パターンによる解析』](#)を参照してください。

以下に、2つのテンプレート、すなわち、単純なテンプレートとリテラル・ストリング・パターンを含むテンプレートを示します。

```

var1 var2          /* simple template          */
var1 ' , ' var2    /* template with literal string pattern */

```

リテラル・ストリング・パターンは、', 'です。このテンプレートは、次のことを行います。

- ソース・ストリングの初めから一致した最初の文字(コンマ、ただしこれは入りません)までの文字を、var1に入れます。
- 一致した最後の文字の次の文字(コンマに続くブランクの次の文字)からそのストリングの終わりまでの文字を、var2に入れます。

ストリング・パターンを使用するテンプレートでは、ソース・ストリング内のデータを変数に割り当てるときに、データの一部が含まれないことになります。次の2つの例は、単純形式テンプレートと、リテラル・ストリング・パターンを含むテンプレートとを対比しています。

```

/* Simple template          */
name='Smith, John'
parse var name ln fn        /* Assigns: ln='Smith,' */
/*                          /*      fn='John' */

```

コンマが残っていることに注意してください(ln 変数には 'Smith,' が入ります)。次の例のテンプレートは ln ' , ' fn です。これにより、コンマが削除されます。

```

/* Template with literal string pattern          */
name='Smith, John'
parse var name ln ' , ' fn          /* Assigns: ln='Smith' */
/*                          /*      fn='John' */

```

まず、言語処理プログラムは、ソース・ストリングで','をスキャンします。それが見つかった位置でソース・ストリングが分割されます。ln 変数には、ソース・ストリングの最初の文字から、一致した文字の手前までのデータが入ります。変数 fn には、一致した文字の次の文字から、ストリングの終わりまでのデータが入ります。

ストリング・パターンを持つテンプレートは、パターンに一致する、ソース・ストリング内のデータを除外します。ストリング・パターンを使用するテンプレートでも、ソース・ストリング内の一致するデータが省略されないという特殊な場合があります。[236 ページの『ストリング・パターンと定位置パターンの組み合わせ: 特殊なケース』](#)を参照してください。前の例では、', '(ブランクなし)のパターンではなく、パターン ' , ' (ブランクあり)を使用しました。なぜなら、パターンにブランクがないと、変数 fn には 'John' が(ブランクも含めて)入るからです。

ソース・ストリングにストリング・パターンに一致するものが含まれていない場合、一致しないストリング・パターンの前にあるどの変数にも、該当のデータがすべて入ります。そのパターンの後にある変数はいずれも、ヌル・ストリングを受け取ります。

ヌル・ストリングを検出することはできません。ヌル・ストリングは、常にソース・ストリングの終わりと突き合わせされます。

定位置 (数値) パターンを含むテンプレート

定位置パターンは、ソース・ストリング内のデータを分割する文字位置を示す数値です。この数値は整数でなければなりません。

次のものは絶対定位置パターンです。

- 数値の前に正符号 (+) も負符号 (-) も付いていないもの。数値の前に等号 (=) が付いているもの。
- 括弧に入った変数で左括弧の前に等号を付けたもの。可変定位置パターンについて詳しくは、[233 ページの『変数パターンによる解析』](#)を参照してください。

この数字は、ソース・ストリングを分割する絶対的な文字位置を表します。

次の例は、絶対定位置パターンを使用したテンプレートの例です。

```
variable1 11 variable2 21 variable3
```

数値の 11 と 21 が、絶対定位置パターンです。数値 11 は、入力ストリングの 11 番目の位置を指し、21 は同じく 21 番目の位置を指します。このテンプレートは、次のことを行います。

- ソース・ストリングの 1～10 桁目までの文字を variable1 に入れます。
- 11～20 桁目までの文字を variable2 に入れます。
- 21 桁目から終わりまでの文字を variable3 に入れます。

定位置パターンは、次のようなレコードのファイルを処理する場合に、最も有用と考えられます。

```
character positions:
      1      11      21      40
+-----+-----+-----+-----+end of
FIELDS: |LASTNAME |FIRST  |PSEUDONYM |record
+-----+-----+-----+-----+
```

以下の例では、このレコード構造を使用します。

```
/* Parsing with absolute positional patterns in template */
record.1='Clemens Samuel Mark Twain '
record.2='Evans Mary Ann George Eliot '
record.3='Munro H.H. Saki '
do n=1 to 3
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end
/* Says 'By George!' after record 2 */
```

ソース・ストリングは、まず 11 桁目と 21 桁目の文字位置で分割されます。言語処理プログラムは、1～10 桁目までの文字を lastname に、11～20 桁目までの文字を firstname に、21～40 桁目までの文字を pseudonym に割り当てます。

このテンプレートは次のように指定することもできます。

```
1 lastname 11 firstname 21 pseudonym
```

上記の例では、以下のテンプレートを使用していました。

```
lastname 11 firstname 21 pseudonym
```

1 の指定は任意です。

必要に応じて、テンプレート内の数値の前に等号を置くことができます。等号は、テンプレートの中では数値の前に符号を置かないのと同じです。数値が、ソース・ストリング内の特定の文字位置を表します。以下の 2 つのテンプレートは、同じ働きをします。

```

lastname 11 first 21 pseudonym
lastname =11 first =21 pseudonym

```

相対定位置パターンは、数値の前に正符号 (+) または負符号 (-) が付いているものです。(これは、左括弧の前にプラス (+) 符号またはマイナス (-) 符号を付けた、括弧で囲んだ変数にすることもできます。詳しくは、[233 ページの『変数パターンによる解析』](#)を参照してください。)

この数値は、ソース・ストリングを分割する相対的な文字位置を表します。正または負はそれぞれ、ストリングの先頭(最初のパターンの場合)または最後に一致した位置から右方または左方に移動することを示します。最後に一致した位置とは、最後に一致した個所の最初の文字のことです。以下の例は、絶対定位置パターンの場合と同じ例を相対定位置パターンを用いて行ったものです。

```

/* Parsing with relative positional patterns in template */
record.1='Clemens Samuel Mark Twain'
record.2='Evans Mary Ann George Eliot'
record.3='Munro H.H. Saki'
do n=1 to 3
  parse var record.n lastname +10 firstname + 10 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end
/* same results */

```

符号と数値の間のブランクには意味がありません。したがって、+10 と + 10 は同じ結果になります。+0 は相対定位置パターンとして有効です。

絶対定位置パターンと相対定位置パターンはどちらを使用してもかまいません(ただし、変数名の前にストリング・パターンを置き、変数名の後に定位置パターンを置くような特殊な場合は除きます。[236 ページの『ストリング・パターンと定位置パターンの組み合わせ: 特殊なケース』](#)を参照してください)。絶対定位置パターンと相対定位置パターンの例にあるテンプレートの結果は、同じになります。

	lastname 11	firstname 21	pseudonym
	lastname +10	firstname + 10	pseudonym
(Implied starting point is position 1.)	Put characters 1 through 10 in lastname. (Non-inclusive stopping point is 11 (1+10).)	Put characters 11 through 20 in firstname. (Non-inclusive stopping point is 21 (11+10).)	Put characters 21 through end of string in pseudonym.

定位置パターンを使用する場合に限り、ソース・ストリング内の以前の位置に戻ってマッチング操作を行うことができます。絶対定位置パターンを使用する例を以下に示します。

```

/* Backing up to an earlier position (with absolute positional) */
string='astronomers'
parse var string 2 var1 4 1 var2 2 4 var3 5 11 var4
say string 'study' var1||var2||var3||var4
/* Displays: "astronomers study stars" */

```

絶対定位置パターンの 1 は、ソース・ストリングの最初の文字に戻ります。

相対定位置パターンの場合は、負符号が前に付いている数を使用すれば、以前の位置に戻れます。同じ例を、相対定位置パターンを用いて書くと次のようになります。

```

/* Backing up to an earlier position (with relative positional) */
string='astronomers'
parse var string 2 var1 +2 -3 var2 +1 +2 var3 +1 +6 var4
say string 'study' var1||var2||var3||var4 /* same results */

```

この例では、相対定位置パターンの -3 でソース・ストリングの最初の文字に戻ります。

最後の 2 つの例のテンプレートは同等です。

2	var1 4	1	var2 2	4 var3 5	11 var4
2	var1 +2	-3	var2 +1	+2 var3 +1	+6 var4

Start	Non-	Go to 1.	Non-	Go to 4	Go to 11
at 2.	inclusive	(4-3=1)	inclusive	(2+2=4).	(5+6=11)
	stopping		stopping	Non-inclusive	
	point is 4		point is	stopping point	
	(2+2=4).		2 (1+1=2).	is 5 (4+1=5).	

以下のように、定位置パターンを持つテンプレートを使用して、複数の割り当てを行うことができます。

```
/* Making multiple assignments */
books='Silas Marner, Felix Holt, Daniel Deronda, Middlemarch'
parse var books 1 Eliot 1 Evans
/* Assigns the (entire) value of books to Eliot and to Evans. */
```

パターンの結合とワードへの構文解析

テンプレートのパターンが、複数のワードから成るセクションにソース・ストリングを分割する場合はどうなるでしょうか。ストリング・パターンと定位置パターンで、ソース・ストリングがサブストリングに分割されます。次に、言語処理プログラムは、ワードへの解析規則に従って、各サブストリングにテンプレートの 1 セクションを当てはめます。

```
/* Combining string pattern and parsing into words */
name=' John Q. Public'
parse var name fn init '.' ln
/* Assigns: fn='John' */
/*          init=' Q' */
/*          ln=' Public' */
```

パターンは、テンプレートを次の 2 つのセクションに分割します。

- fn init
- ln

突き合わせパターンは、ソース・ストリングを次の 2 つのサブストリングに分割します。

- ' John Q'
- ' Public'

言語処理プログラムは、テンプレートの該当するセクションに基づいて、これらのサブストリングをワードに解析します。

John には、先行空白が 3 個ありますが、ワードに構文解析すると、最後の変数を除き、先行空白および末尾空白は削除されるので、すべて削除されます。

Q には先行空白が 6 個ありますが、init はこのテンプレートの該当のセクションの最後の変数であるため、構文解析をすると、ワード区切り文字空白が 1 つ削除され、残りは保持されます。

サブストリング 'Public' の場合は、構文解析をすると、空白は何も削除されずに、ストリング全体が ln に割り当てられます。これは、ln が、テンプレートのこのセクション内の唯一の変数であるためです。(空白の取り扱いの詳細については、[227 ページの『いくつかのワードに構文解析するための単純形式のテンプレート』](#) ページを参照してください。)

```
/* Combining positional patterns with parsing into words */
string='R E X X'
parse var string var1 var2 4 var3 6 var4
/* Assigns: var1='R' */
/*          var2='E' */
/*          var3=' X' */
/*          var4=' X' */
```

このパターンは、テンプレートを次の 3 つのセクションに分割します。

- var1 var2
- var3
- var4

突き合わせパターンはソース・ストリングを次の 3 つのサブストリングに分割し、それらは個々にワードに構文解析されます。

- 'R E'
- ' X'
- ' X'

変数 `var1` には 'R' が、`var2` には 'E' が入ります。 `var3` と `var4` はそれぞれ、テンプレートの そのセクション内の唯一の変数なので、この両方に、' X' (1 個の空白と X) が入ります。(空白の取り扱いの詳細については、[227 ページの『いくつかのワードに構文解析するための単純形式のテンプレート』](#) ページを参照してください。)

変数パターンによる解析

固定文字列または数値の代わりに、変数の値を使用してパターンを指定することもできます。この場合は、変数の名前を括弧に入れます。これは、変数参照です。

このタスクについて

空白は括弧の内側でも外側でも必須ではありませんが、必要な場合はそれらを追加することができます。

次の構文解析命令のテンプレートには、以下のリテラル・文字列・パターン ' . ' が入っています。

```
parse var name fn init ' . ' ln
```

以下に、変数文字列・パターンとしてそのパターンを指定する方法を示します。

```
stringptrn=' . '
parse var name fn init (stringptrn) ln
```

等号、正符号、または負符号が変数名の前にある括弧の前に付いていない場合、変数の値は文字列・パターンとして扱われます。変数は、同じテンプレートの中で以前に設定されたものにすることができます。以下に例を示します。

```
/* Using a variable as a string pattern */
/* The variable (delim) is set in the same template */
SAY "Enter a date (mm/dd/yy format). =====> " /* assume 11/15/90 */
pull date
parse var date month 3 delim +1 day +2 (delim) year
/* Sets: month='11'; delim='/'; day='15'; year='90' */
```

左括弧の前に等号、プラス符号、またはマイナス符号がある場合は、変数の値は、絶対または相対の位置パターンとして扱われます。変数の値は、正の整数またはゼロでなければなりません。

変数は、同じテンプレートの中で以前に設定されたものにすることができます。次の例では、最初の 2 つのフィールドで最後の 2 つのフィールドの開始文字位置を指定しています。

```
/* Using a variable as a positional pattern */
dataline = '12 26 .....Samuel ClemensMark Twain'
parse var dataline pos1 pos2 6 =(pos1) realname =(pos2) pseudonym
/* Assigns: realname='Samuel Clemens'; pseudonym='Mark Twain' */
```

どうして位置パターン 6 がテンプレートの中に必要なのでしょうか? ワードの解析は、言語処理プログラムがパターンを使用してソース・文字列をサブ文字列に分割した後で行われることを、思い出してください。したがって、位置パターン =(pos1) を =12 として正しく解釈することは、言語プロセッサが文字列を 6 桁目で分割した後に、空白で区切ったワード 12 と 26 をそれぞれ、pos1 と pos2 に割り当てるまで不可能です。

UPPER の使用

どの PARSE 命令でも UPPER が指定されると、解析の前に文字が大文字に変換 (小文字の a から z が大文字の A から Z に) されます。

このタスクについて

以下の表は、構文解析命令による大/小文字に対する影響を要約したものです。

解析の前に英字は大文字に変換されます	英字は入力されたときの状態のままです
ARG PARSE UPPER ARG	PARSE ARG
PARSE UPPER EXTERNAL	PARSE EXTERNAL
PARSE UPPER NUMERIC	PARSE NUMERIC
PULL PARSE UPPER PULL	PARSE PULL
PARSE UPPER SOURCE	PARSE SOURCE
PARSE UPPER VALUE	PARSE VALUE
PARSE UPPER VAR	PARSE VAR
PARSE UPPER VERSION	PARSE VERSION

ARG 命令は、PARSE UPPER ARG を単に短くしただけの形であり、PULL 命令は、PARSE UPPER PULL を単に短くしただけの形です。大文字変換を行いたくなければ、(ARG または PARSE UPPER ARG を使用せずに) PARSE ARG を使用し、(PULL または PARSE UPPER PULL を使用せずに) PARSE PULL を使用してください。

解析命令の要約

すべての構文解析命令は、ソース・ストリングの一部を、テンプレートで指定された変数に割り当てます。

このタスクについて

以下の表は、ソース・ストリングがある場所を要約したものです。

命令	ソース・ストリングがある場所
ARG PARSE ARG	プログラムを呼び出すときに指定した引数、またはサブルーチンや関数呼び出しの引数。
PARSE EXTERNAL	端末入力バッファの次の行。
PARSE NUMERIC	数値制御情報 (NUMERIC 命令から)。
PULL PARSE PULL	外部データ・キューの先頭にあるストリング。(キューが空の場合は、デフォルトの入力、一般には端末装置を使用します。)
PARSE SOURCE	システム提供のストリングで、実行中のプログラムについての情報です。
PARSE VALUE	命令内の VALUE キーワードと WITH キーワードの間にある式。
PARSE VAR <i>name</i>	<i>name</i> を解析します。
PARSE VERSION	システム提供のストリングで、言語、言語レベル、および (3 ワードの) 日付を表します。

解析命令の例

ソース・ストリングを解析してワードに分解する例を示します。

このタスクについて

ARG

```
/* ARG with source string named in REXX program invocation */
/* Program name is PALETTE. Specify 2 primary colors (yellow, */
/* red, blue) on call. Assume call is: palette red blue */
arg var1 var2 /* Assigns: var1='RED'; var2='BLUE' */
If var1<>'RED' & var1<>'YELLOW' & var1<>'BLUE' then signal err
If var2<>'RED' & var2<>'YELLOW' & var2<>'BLUE' then signal err
total=length(var1)+length(var2)
SELECT;
  When total=7 then new='purple'
  When total=9 then new='orange'
  When total=10 then new='green'
  Otherwise new=var1 /* entered duplicates */
END
Say new; exit /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit
```

ARG は、解析前に英字を大文字に変換します。サブルーチンへの CALL 内に引数が入っている ARG の例が、[236 ページの『複数のストリングの構文解析』](#)に示されています。

PARSE ARG

ARG と同じ働きをします。ただし、PARSE ARG は、解析前に英字を大文字に変換しません。

PARSE EXTERNAL

```
Say "Enter Yes or No =====> "
parse upper external answer 2 .
If answer='Y'
  then say "You said 'Yes'!"
  else say "You said 'No'!"
```

PARSE NUMERIC

```
parse numeric digits fuzz form
say digits fuzz form /* Displays: '9 0 SCIENTIFIC' */
/* (if defaults are in effect) */
```

PARSE PULL

```
PUSH '80 7' /* Puts data on queue */
parse pull fourscore seven /* Assigns: fourscore='80'; seven='7' */
SAY fourscore+seven /* Displays: "87" */
```

PARSE SOURCE

```
parse source sysname .
Say sysname /* Displays: "CICS" */
```

PARSE VALUE

[227 ページの『いくつかのワードに構文解析するための単純形式のテンプレート』](#)の例を参照してください。

PARSE VAR

[227 ページの『いくつかのワードに構文解析するための単純形式のテンプレート』](#)から始めて、構文解析に関する情報全体に含まれている例を参照してください。

PARSE VERSION

```
parse version . level .
say level /* Displays: "3.48" */
```

PULL

PARSE PULL と同じ働きをします。ただし、PULL は、解析前に英字を大文字に変換します。

高度な解析に関する情報

高度な解析には、複数のストリングの解析、DBCS 文字の解析などの特殊なケースがあります。

解析の概念的視点を表すフローチャートを用意しています。

複数のストリングの構文解析

複数のソース・ストリングを指定することができるのは、ARG と PARSE ARG だけです。複数のストリングを解析する場合は、複数のテンプレートをコンマで分けて指定します。

このタスクについて

例を、次に示します。

```
parse arg template1, template2, template3
```

この命令は、PARSE ARG キーワードと、コンマで分けた 3 つのテンプレートで構成されています。(ARG 命令の場合、解析されるソース・ストリングは、プログラム呼び出し時か、サブルーチンまたは関数の CALL 時に指定した引数です。) コンマがそれぞれ、次のストリングへ移動するようにコマンド解析機能に命令します。

例:

```
/* Parsing multiple strings in a subroutine          */
num='3'
musketeers="Porthos Athos Aramis D'Artagnon"
CALL Sub num,musketeers /* Passes num and musketeers to sub */
SAY total; say fourth /* Displays: "4" and " D'Artagnon"    */
EXIT

Sub:
parse arg subtotal, . . . fourth
total=subtotal+1
RETURN
```

REXX プログラムをコマンドとして開始する場合、引数ストリングは 1 つしか認められません。次の場合には、複数の引数ストリングを渡して解析することができます。

- ある REXX プログラムが、CALL 命令または関数呼び出しを使用して別の REXX プログラムを呼び出す場合
- REXX プログラムが、他の言語で書かれているプログラムによって開始される場合

ソース・ストリングよりもテンプレートが多いときは、残ったテンプレート内の各変数に、ヌル・ストリングが入ります。テンプレートよりもソース・ストリングが多いときは、言語処理プログラムは、残っているソース・ストリングを無視します。テンプレートが空 (2 個のコンマを並べただけ) であるか、または変数名が書かれていなければ、その次のテンプレートとソース・ストリングの解析が行われます。

ストリング・パターンと定位置パターンの組み合わせ: 特殊なケース

絶対定位置パターンと相対定位置パターンが同じ働きをしない特殊な場合があります。

このタスクについて

ストリング・パターンを含んでいるテンプレートを用いる構文解析で、パターンと一致するソース・ストリングの中のデータがどのようにスキップされるかを示してきました ([229 ページの『ストリング・パターンを含むテンプレート』](#) ページを参照)。ただし、以下のシーケンスを含むテンプレートでは、一致するデータはスキップされません。

- ストリング・パターン
- 変数名
- 相対定位置パターン

相対位置パターンは、ストリング・パターンに一致した最初の文字との相対関係で移動します。その結果として、ストリング・パターンに一致する、ソース・ストリング内のデータも割り当てられることになります。

```
/* Template containing string pattern, then variable name, then */
/* relative positional pattern does not skip over any data.      */
string='REstructured eXtended eXecutor'
parse var string var1 3 junk 'X' var2 +1 junk 'X' var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

以下に、どのようにこのテンプレートが働くかを示します。

var1 3	junk 'X'	var2 +1	junk 'X'	var3 +1	junk
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+
Put	Starting	Starting	Starting	Starting	Starting
characters	at 3, put	with first	with char-	with	with char-
1 through	characters	'X' put 1	acter after	second 'X'	acter
2 in var1.	up to (not	(+1)	first 'X'	put 1 (+1)	after sec-
(Stopping	including)	character	put up to	character	ond 'X'
point is	first 'X'	in var2.	second 'X'	in var3.	put rest
3.)	in junk.		in junk.		in junk.
var1='RE'	junk='	var2='X'	junk='	var3='X'	junk='
	structured e'		tended e'		ecutor'

DBCS 文字の解析

DBCS 文字の解析は、一般に SBCS 文字の解析と同じ規則に従います。

このタスクについて

リテラル・ストリングおよび記号には DBCS 文字を含められますが、数値は SBCS 文字でなければなりません。DBCS の解析例については、[456 ページの『PARSE』](#)を参照してください。

解析ステップの詳細

解析の概念的視点を図で示します。

以下に示す 3 つの図は、解析の概念を説明したものです。ただし、これらの図にはエラーの場合を含めてありません。

これらの図で使用している用語の意味は次のとおりです。

string start

ソース・ストリング (またはサブストリング) の始め。

string end

ソース・ストリング (またはサブストリング) の終わり。

長さ

ソース・ストリングの長さ。

match start

ソース・ストリング内の、突き合わせで一致した最初の文字。

match end

ソース・ストリング内の文字で、ストリング・パターンの場合は、一致している最後の文字の次の文字。定位置パターンの場合は、match start と同じ。

match position

ソース・ストリング内の文字で、ストリング・パターンの場合は、最初のマッチング文字。定位置パターンの場合は、マッチング文字の位置。

token

テンプレート内の、変数、ピリオド、パターン、またはコンマなどの、構文上の個別の要素。

value

定位置パターンの数値。これは、変数から求めた値または定数のいずれかにすることができます。

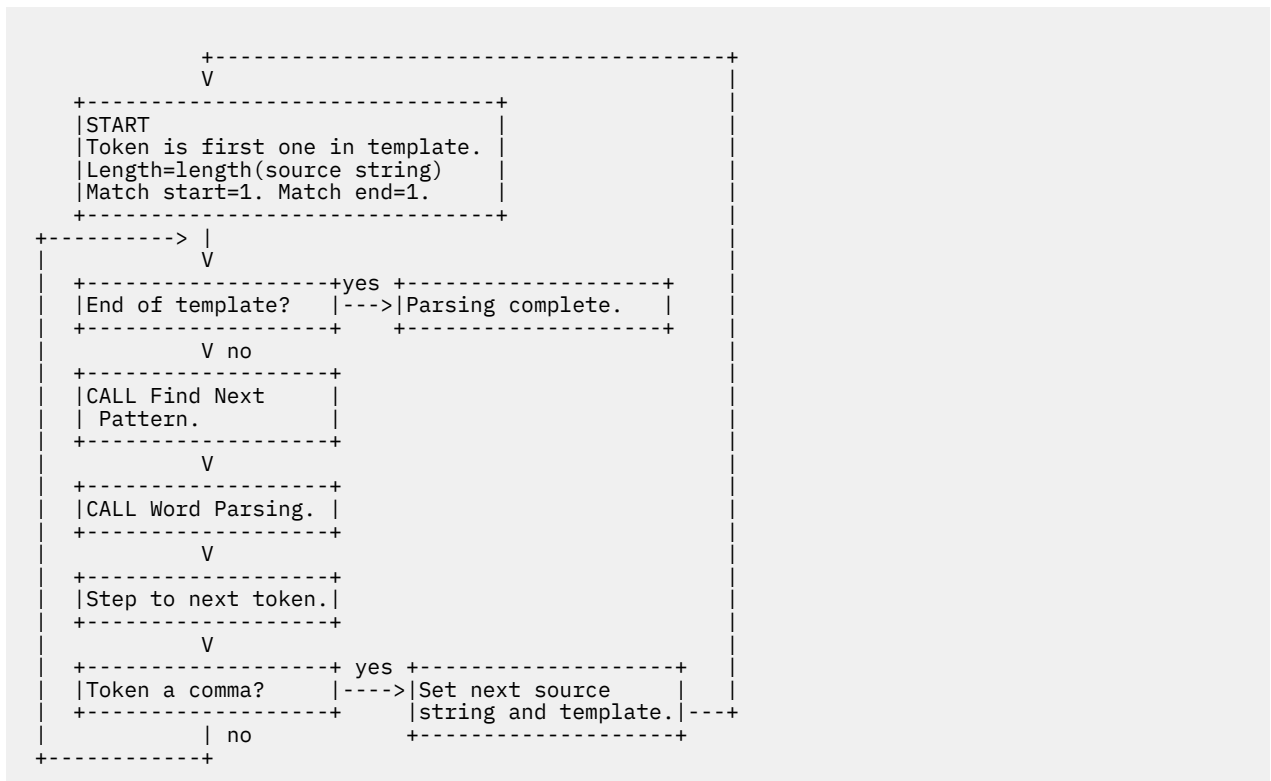


図 48. 構文解析の概説

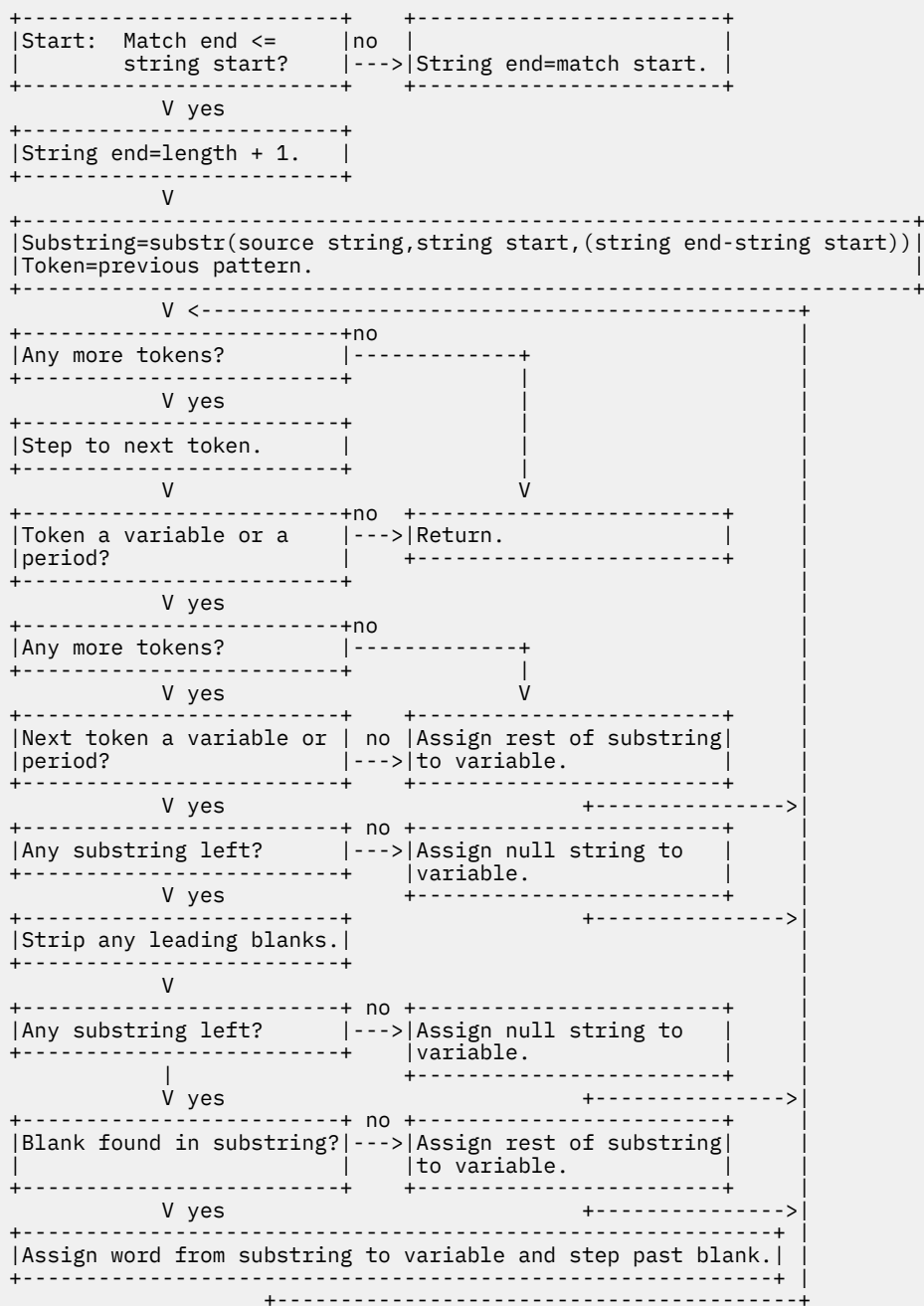


図 50. Word Parsing (ワード解析) の概念

第 21 章 数値と算術演算

REXX は、通常の算術演算 (加算、減算、乗算、および除算) を、可能な限り通常の方法で定義しています。つまり、学校で学習する型どおりの規則に従います。

ただし、残念なことです。使用する人およびアプリケーション・プログラムなどによって使用規則がかなり (実際には、一般に認められているよりもさらに大幅に) 異なっており、必ずしもそれらの規則の違いを予測できないということが、これらの機能の設計時に分かりました。したがって、本書で説明している算術演算は、大部分のアプリケーション・プログラムに受け入れられるようにした妥協案です (ただし、最も単純な方法とはいえません)。

概要: 数値

数値 (つまり、REXX 算術演算および組み込み関数への入力として使用される文字ストリング) の表現方法は、柔軟性に富んでいます。先行ブランクおよび末尾ブランクが許されており、指数表記を使用することもできます。

いくつかの有効な数値を、下記に示します。

```
12          /* a whole number          */
'-76'       /* a signed whole number       */
12.76       /* decimal places             */
' + 0.003 '  /* blanks around the sign and so forth */
17.         /* same as "17"               */
.5          /* same as "0.5"              */
4E9         /* exponential notation       */
0.73e-7     /* exponential notation       */
```

指数表記では、数値に指数が組み込まれています。この指数は、数値が使用される前に乗算される 10 の累乗です。この指数は、小数点がどのように移動するかを示します。したがって、前の例で、4E9 は 4000000000 の短縮形式であり、0.73e-7 は 0.000000073 の短縮形式です。

算術演算子には、加算 (+)、減算 (-)、乗算 (*)、累乗 (**)、除算 (/)、接頭正符号 (+)、および符号反転前置 (-) があります。その他に、除算に関する 2 つの演算子があります。整数の除算 (%) は、除算後に整数部分を戻します。剰余 (/ /) は、除算後に剰余を戻します。

算術演算の結果は、一定の規則に従って、文字ストリングとして形式設定されます。これらの規則の中で最も重要な規則は、以下のとおりです (詳細については、[242 ページの『算術機能の定義』](#)を参照してください)。

- 結果は、有効数字の最大桁数まで計算されます (デフォルトは 9 ですが、NUMERIC DIGITS 命令を使用すればデフォルトを変更できるので、必要な正確度が得られます)。したがって、結果が 9 桁を超える場合は、通常、9 桁に丸められます。例えば、2 を 3 で除算した結果は、0.666666667 になります (完全に正確に計算するには、桁数が無限に必要になります)。
- 除算および累乗の場合を除いて、後続ゼロは保持されます (これは、結果の小数部から後続のゼロがすべて除去される、大部分の計算器と異なります)。したがって、例えば次のようになります。

```
2.40 + 2    ->    4.40
2.40 - 2    ->    0.40
2.40 * 2    ->    4.80
2.40 / 2    ->    1.2
```

このようにすることは、ほとんどの計算 (特に、財務計算) に好都合であるといえます。

必要に応じ、STRIP 関数 ([215 ページの『STRIP』](#) ページを参照) を使用するか、あるいは、1 で割って、後続ゼロを除去できます。

- 結果が 0 であれば、それは必ず 1 桁の 0 で表されます。

- 結果の値および NUMERIC DIGITS の設定値 (デフォルトは 9) に応じて、結果は指数形式で表されます。小数点の前に必要となる桁数が NUMERIC DIGITS の設定値を超える場合、または小数点の後の桁数が NUMERIC DIGITS の設定値の 2 倍を超える場合、数値は下記のように指数表記で表されます。

```
1e6 * 1e6    ->    1E+12      /* not 10000000000000 */
1 / 3E10     ->    3.33333333E-11 /* not 0.000000000033333333 */
```

算術機能の定義

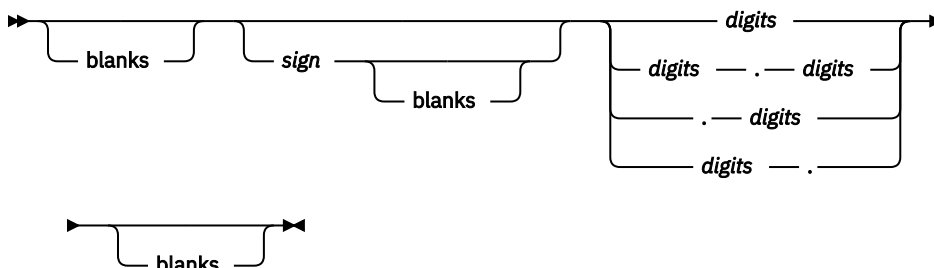
ここでは、REXX 言語の算術機能の正確な定義を説明します。

数字

REXX の数値とは、1 桁またはそれ以上の 10 進数からなる文字ストリングで、オプションで小数点が付けられます。

この定義の拡張については、246 ページの『指数表記』を参照してください。小数点は、数値の中に組み込むことも、数値の前または後に付けることもできます。この方法で構成した数字 (および、オプションの小数点) のグループには、先行ブランクまたは後続のブランクの他に、オプションの符号 (+ または - で、数字または小数点の前に置く必要がある) を付け加えることができます。符号にも、先行ブランクまたは後続ブランクを付け加えることができます。

したがって、数値は次のように定義されます。



ブランク

1 桁またはそれ以上のスペースです。

sign

+ または - です。

digits

1 桁またはそれ以上の 10 進数 (0 から 9) です。

1 個のピリオドだけでは、有効な数値とはいえません。

Precision (精度)

精度とは、演算結果とすることができる有効数字の最大数です。

精度は、次の命令によって制御されます。

```
➡➡ NUMERIC DIGITS expression ;➡➡
```

expression の計算結果は、正の整数にならなければなりません。この命令は、計算結果の精度 (有効数字の桁数) を定義します。結果は、必要に応じてその精度に合わせて丸められます。

この命令に *expression* を指定しない場合、またはプログラムを開始してから NUMERIC DIGITS 命令を一度も実行していない場合は、デフォルトの精度が使用されます。REXX 標準のデフォルト精度は、9 です。

NUMERIC DIGITS には、デフォルトの 9 未満の値をセットすることができます。ただし、小さな値を使用する場合は注意してください。精度が失われたり、小さな値にするために丸めが要求されると、REXX のすべての計算に影響を与えることになります。例えば、DO ループにおける制御変数の新しい値を計算する場合も影響を受けます。

算術演算子

REXX 演算は、2つの項に適用される演算子 +、-、*、/、%、//、および ** (加算、減算、乗算、除算、整数の除算、剰余、および累乗) と、単一の項に適用される正と負の接頭演算子を使用して実行されます。

算術演算の前に必ず、演算の対象となる項 (単数または複数) の先行ゼロが削除されます (小数点の位置に注意し、数値内のすべての数字がゼロのときはゼロを 1 個だけ残すようにされます)。次に、(必要に応じて) DIGITS + 1 桁の有効数字に切り捨てられてから、計算で使用されます。(付け加えた余分な桁は“保護”桁です。これは、数値が所要の精度に丸められる場合、演算の終わりに検査されるため、正確度が上がります。) 次に、以降の個々の演算の説明にあるとおり、最大 2 倍の精度で演算が実行されます。演算が完了すると、必要に応じて、NUMERIC DIGITS 命令によって指定された精度に合わせて結果が丸められます。

丸めは、従来方式で行われます。結果の最下位数字の右側の桁 (「保護桁」) が検査され、5 から 9 までの値は切り上げられ、0 から 4 までの値は切り捨てられます。偶数 / 奇数の丸めには、計算結果を常に任意の精度で出せなければならないので、このメカニズムは REXX には定義されていません。

小数点の前に数字がない場合には、慣習上、0 を小数点の前に付けます。加算、減算、および乗算の場合は、後述する規則に従って、有効な後続ゼロが保持されます。ただし、0 という結果は常に 1 桁の 0 で表されます。除算の場合は、丸められた後に無効な後続ゼロは除去されます。

提供された標準の結果がお客様の要件に合致しない場合は、[FORMAT 組み込み関数 \(208 ページの『FORMAT』を参照\)](#) を使用すると、特定の形式で数値を表すことができます。

算術演算規則: 基本演算子

基本演算子 (加算、減算、乗算、および除算) によって行われる数値の演算は、以下のとおりです。

加算および減算

一方が 0 ならば、もう一方の数値が必要に応じて NUMERIC DIGITS 桁に丸められてから、結果として使用されます (該当する符号が付きます)。上記以外の場合は、2つの数値は、必要に応じて、合計で最大 DIGITS + 1 桁まで右左に拡張され (このため、絶対値が小さい方の数値は、右側でその数字が数桁またはすべて失われることもあります)、そのあとで、適宜、加算または減算が行われます。

例:

```
xxx.xxx + yy.yyyyy
```

この演算は、次のように計算されます。

```
xxx.xxx00
+ 0yy.yyyyy
-----
zzz.zzzzz
```

その後、結果は必要に応じて NUMERIC DIGITS の現行設定値に丸められます (加算の後では、左側での余分の「繰り上がり数字」も計算に入れますが、それ以外の場合は、加算または減算が行われている項の最上位数字に対応する桁からカウントします)。最後に、無意味な先行ゼロは除去されます。

接頭演算子は、同じ規則を用いて評価されます。つまり、+number および -number という演算はそれぞれ、0+number および 0-number として計算されます。

乗算

数値はまとめて乗算 (「長精度乗算」) され、結果は、2つのオペランドの長さを合計したのと同じ長さの数値になります。

例:

```
xxx.xxx * yy.yyyyy
```

この演算は、次のように計算されます。

```
zzzzz.zzzzzzzz
```

結果は、最初の有効数字からカウントされて、NUMERIC DIGITS の現行設定値になるまで丸められます。

除算

次の除算は、下記のステップで実行されます。

```
yyy / xxxxx
```

まず、数値 xxxxx より大きくなるまで、数値 yyy の右側に 0 が追加されます (10 の累乗の指数が変更されることを意味します)。したがって、この例では、yyy が yyy00 になります。次に、通常の長精度除算が行われます。これは、以下のように記述されます。

```
      zzzz
    +-----
xxxxx | yyy00
```

この例では、結果 (zzzz) の長さは、右端の z が、(拡張された) y の数値の右端の桁と同じかそれより右にくるようになります。除算の間に、y の数値は、必要に応じてさらに拡張されます。z の数値は、NUMERIC DIGITS+1 桁まで増やすことができます (その時点で除算は停止し、結果が丸められます)。除算が完了したあとで (必要なら丸めもして)、有効でない後続ゼロは削除されます。

基本演算子の例

以下は、ここまでで説明した規則の主な意味を表す例です。

```
/* With: Numeric digits 5 */
12+7.00    -> 19.00
1.3-1.07   -> 0.23
1.3-2.07   -> -0.77
1.20*3     -> 3.60
7*3        -> 21
0.9*0.8    -> 0.72
1/3        -> 0.33333
2/3        -> 0.66667
5/2        -> 2.5
1/10       -> 0.1
12/12      -> 1
8.0/2      -> 4
```

注: すべての基本演算子で、演算の対象となる項の小数点の位置は任意です。演算は整数演算として実行され、指数は後で計算されて適用されます。したがって、結果の有効数字は、演算に関与するいずれの項の小数点の位置とも関係ありません。

算術演算規則: 加算演算子

累乗 (**)、整数除算 (%)、および剰余 (/) の各演算子の演算規則について説明します。

Power®

** (累乗) 演算子は、数値の累乗を計算します。結果は正、負、または 0 になります。指数は整数でなければなりません。(この演算の第 2 項は整数でなければならないので、必要に応じて、[248 ページの『REXX によって直接使用される数値』](#)で説明するように DIGITS 桁に丸められます。) 負の場合は、累乗の絶対値が使用され、結果は逆数 (1 で除算) になります。累乗の計算では、累乗で表された回数だけ、実際にその数値を乗算し、最後に、(結果を 1 で除算した場合と同様に) 後続のゼロを除去します。

実際には、累乗は左から右への 2 進数減算処理によって計算されます (理由については、[245 ページの『1』](#)を参照してください)。a**n の場合は、n が 2 進数に変換され、一時的なアキュムレーターが 1 に設定されます。n = 0 であれば、初期の計算は完了します (したがって、すべての a (0**0 も含みます) に対して、a**0=1 となります。) それ以外の場合は、各ビット (最初の非ゼロ・ビットから始めて) が、左から右へ検査されます。現行ビットが 1 の場合、アキュムレーターは a 倍されます。すべてのビットが検査されると、初期の計算は完了します。それ以外の場合は、アキュムレーターは二乗され、次のビットが乗算のために検査されます。初期計算が完了した時点で、累乗が負だった場合は、一時結果を 1 で除算します。

乗算および除算は、 $\text{DIGITS} + L + 1$ 桁の精度を使用して、算術演算規則のもとで行われます。Lは、整数nの整数部分の桁数(すなわち、組み込み関数 $\text{TRUNC}(n)$ を使用した場合と同様の、小数部を除いた桁数)です。結果は最終的には、必要であれば **NUMERIC DIGITS** 桁に丸められ、意味をもたない後続ゼロは除去されます。

整数の除算

% (整数の除算) 演算子は、2つの数で除算を行い、結果の整数部分を戻します。戻される結果は、被除数が除数より大きい間は、被除数から除数を繰り返し引いていき、その引いた回数として定義されます。この減算中、被除数と除数は両方とも絶対値が使用されます。最終結果の符号は、通常の除算から得られるものと同じです。

戻される結果には小数部がありません(すなわち、小数点やそれに続くゼロはありません)。結果を整数として表せない場合、演算はエラーとなり、失敗に終わります。すなわち、結果の桁数が **NUMERIC DIGITS** の現行設定値を超えてはなりません。例えば、`100000000000%3` では結果 (`3333333333`) を表すのに 10 桁必要となるので、**NUMERIC DIGITS** 9 が有効であったとすると、失敗することになります。この演算子を使用した場合、通常の除算で切り捨てを行ったときの結果と同じにならないことがあります(丸めによる影響を受けることがあります)。

リマインダー

// (剰余) 演算子は、整数の除算からの剰余を戻すもので、前述した整数結果の除算を行った後の被除数の剰余として定義されています。剰余(非ゼロの場合)の符号は、元の被除数の符号と同じです。

この演算は、整数除算と同じ条件の下で失敗します(つまり、同じ2つの項での整数除算が失敗すると、剰余は計算できません)。

加算演算子の例

累乗、整数除算、および剰余の各演算子を使用する例を以下に示します。

```
/* Again with: Numeric digits 5 */
2**3      ->      8
2**-3     ->     0.125
1.7**8    ->    69.758
2%3       ->      0
2.1//3    ->     2.1
10%3      ->      3
10//3     ->      1
-10//3    ->     -1
10.2//1   ->     0.2
10//0.3   ->     0.1
3.6//1.3  ->     1.0
```

注:

1. 累乗の計算では、特別なアルゴリズムが使用されています。その理由は、このアルゴリズムが効率的(最適ではありませんが)であり、実際の乗算の実行回数をかなり減らすためです。したがって、このアルゴリズムを使用すると、乗算の繰り返しという単純な定義に比べてパフォーマンスが向上します。乗算を繰り返す場合とは結果が異なることがあるため、このアルゴリズムをここで定義しています。
2. 整数の除算および剰余演算子は、標準の除算演算の副産物として計算できるように定義されています。除算処理は、整数結果が出たときにほぼ終了です。被除数の剰余が、剰余だからです。

数値の比較

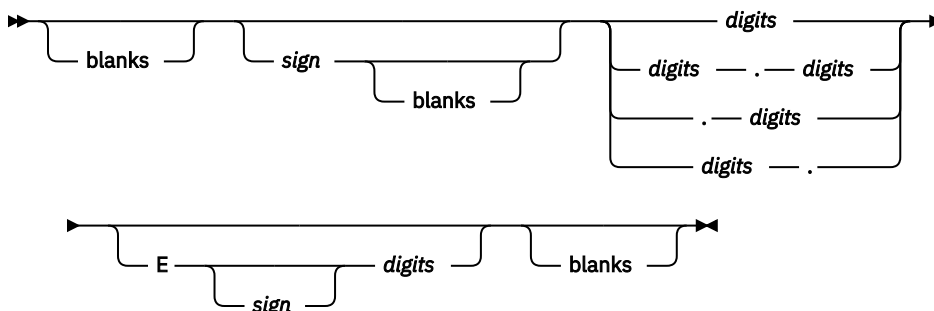
任意の比較演算子を使用して、数値ストリングを比較できます。

比較演算子は、[149 ページ](#)の『比較演算子』のセクションにリストされています。ただし、`==`、`!=`、`==>`、`>>`、`!=>`、`<<`、`!=<<`、および `<<<` は、数値を比較するのに使用しないでください。これらの演算子では、先行および末尾のブランク、および先行ゼロが意味を持つからです。

数値の比較は、2つの数値を減算(差を計算)してから、その結果を0と比較することで実行されます。つまり、次のような演算になります。

この場合、長形式を使用すると 2950800000 と表示されます。このままでは明らかに誤解を招くため、結果は 2.9508E+9 と表されます。

したがって、数値の定義は、下記のように拡張されています。



E の後の整数は、数値に適用される 10 の累乗を表します。E は、大文字でも小文字でもかまいません。

文字ストリングの中には、ユーザーには数値に見えなくても、数値のものがあります。特に、指数表記の数値はその形式のために数値らしくありませんが、0E123 (0 を 123 乗する) や 1E342 (1 を 342 乗する) などのストリングは数値です。さらに、0E123=0E567 などの比較は、1 という真の結果 (0 は 0 に等しい) になります。数値以外のストリングを比較する際に問題が起きないようにするためには、厳密な比較演算子を使用してください。

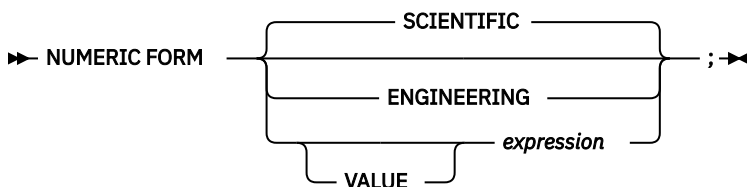
以下に、いくつかの例を示します。

```
12E7  = 120000000 /* Displays "1" */
12E-5 = 0.00012    /* Displays "1" */
-12e4 = -120000     /* Displays "1" */
0e123 = 0e456       /* Displays "1" */
0e123 == 0e456      /* Displays "0" */
```

上記の数値は、入力データとしていつでも有効です。計算結果は、NUMERIC DIGITS の設定値に応じて、通常の形式または指数形式で戻されます。小数点の前に必要な桁数が DIGITS を超える場合、または小数点の後の桁数が DIGITS の 2 倍を超える場合には、指数形式が使用されます。REXX で生成される指数形式には、E の後に必ず符号が付いて、読みやすくなっています。指数が 0 の場合は、指数部分が省略されます。つまり、E+0 という指数部分が生成されることはありません。

FORMAT 組み込み関数を使用すれば、数値を指数形式に明示的に変換したり、あるいは、それらを強制的に長形式で表示させることができます (208 ページの『FORMAT』ページを参照してください)。

科学計算表記法は指数表記の 1 つの形式であり、小数点の左側がゼロ以外の 1 桁の数字になるように 10 の累乗が調整されます。工学表記法は指数表記の 1 つの形式であり、1 桁から 3 桁の数字 (ただし、単なる 0 は除く) が小数点の前にあり、10 の累乗が常に 3 の倍数として表されます。したがって、整数部分は、1 から 999 の範囲になる可能性があります。科学計算表記法と工学表記法のどちらを使用するかを、次の命令で制御できます。



科学計算表記法がデフォルトです。

```
/* after the instruction */
Numeric form scientific

123.45 * 1e11 -> 1.2345E+13

/* after the instruction */
Numeric form engineering

123.45 * 1e11 -> 12.345E+12
```

数値情報

NUMERIC オプションの現行設定値は、組み込み関数の DIGITS、FORM、および FUZZ を使用して知ることができます。

これらの関数は、それぞれ、NUMERIC DIGITS、NUMERIC FORM、および NUMERIC FUZZ の現行設定値を戻します。

整数

REXX が数値として理解している集合のうち、整数として定義されたサブセットを区別しておくとう便利です。

REXX の整数 とは、小数部のすべての桁が 0 である数値 (または小数部のない数値) のことです。さらに、NUMERIC DIGITS 命令によってセットされた精度の範囲内の数字として、整数部分を単純に表すことができなければなりません。この範囲内で表せない大きな数値は、REXX では、丸めてから指数表記で表します。したがって、それらの数値は、整数であると記述することも、整数として使用することもできなくなります。

REXX によって直接使用される数値

すでに説明したように、算術演算の結果は、NUMERIC DIGITS の設定値に応じて (必要であれば) 丸められます。

同様に、REXX が直接数値 (必ずしも算術演算に使用される数値ではありません) を使用する場合にも、同じ丸めが適用されます。これは、数値に 0 を追加する場合に行われることと同じです。

下記の場合、使用する数値は整数でなければならない、また、使用できる最大の数値は 999999999 となります。

- 解析テンプレート内の定位置パターン (可変定位置パターンを含みます)
- 累乗演算子の累乗値 (右側のオペランド)
- DO 命令内の *expr* および *exprf* の値
- NUMERIC 命令内の DIGITS または FUZZ で指定する値
- TRACE 命令内の 数字オプション で使用する数値

エラー

算術演算時には、2 種類のエラーが起こる可能性があります。

- オーバーフローまたはアンダーフロー

このエラーは、NUMERIC DIGITS および NUMERIC FORM の現行設定値に応じて結果を形式設定するときに、言語処理プログラムが処理できる範囲を結果の指数部分が超えた場合に起こります。この言語では、指数部分の最小値が定義されています。これは、デフォルトの精度で正確に整数として表すことができる最大の数値です。デフォルトの精度は 9 なので、REXX/CICS では、-999999999 から 999999999 までの範囲の指数がサポートされます。

これは、(非常に) 大きな指数を許容できるので、オーバーフローもアンダーフロー も構文エラーと見なされます。

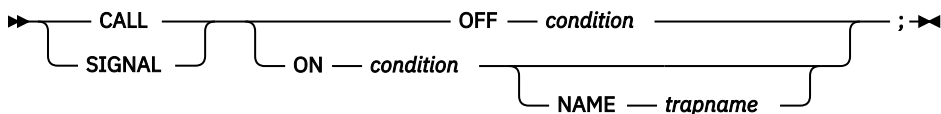
- ストレージ不足

ストレージは、計算及び中間結果に必要であり、ストレージの不足が原因で算術演算が失敗することもあります。このエラーは通常どおり、算術演算エラーではなく終了エラーと見なされます。

第 22 章 条件および条件トラップ

条件とは、CALL ON または SIGNAL ON がトラップできる、指定されたイベントまたは状態です。条件トラップは、REXX プログラム中の実行の流れを変更することができます。

条件トラップは、SIGNAL 命令および CALL 命令の ON または OFF のサブキーワード (165 ページの『CALL』および 186 ページの『SIGNAL』を参照) を使用して、オンまたはオフにされます



condition および *trapname* は、定数と見なされる 1 つの記号です。これらのいずれかの命令があると、条件トラップはオン (使用不可) またはオフ (使用禁止) にセットされます。すべての条件トラップの初期設定値は、オフです。

条件トラップが使用可能にされている場合に、指定された *condition* が発生すると、*trapname* が指定してあれば、*trapname* というルーチンまたはラベルに制御が渡されます。指定していない場合には、*condition* ルーチンまたはラベルに制御が渡されます。条件の最新のトラップが CALL ON と SIGNAL ON のどちらを使用してセットされたのかに応じて、CALL または SIGNAL のどちらかが使用されます。

注: CALL を使用する場合、*trapname* は内部ラベル、組み込み関数、または外部ルーチンにすることができます。SIGNAL を使用する場合、*trapname* は内部ラベルでなければなりません。

条件とそれに対応するトラップ可能なイベントは次のとおりです。

ERROR

コマンドが戻り時にエラー条件を示した場合に発生します。また、いずれかのコマンドが障害を示しているが、CALL ON FAILURE も SIGNAL ON FAILURE もアクティブでない場合にも発生します。この条件は、そのコマンドを呼び出した文節の終わりで発生しますが、エラー条件トラップがすでに遅延状態になっている場合には無視されます。遅延状態とは、条件が発生したが、トラップが「使用可能 (ON)」または「使用不可 (OFF)」にまだリセットされていない場合の条件の状態です。注を参照してください。

CALL ON ERROR および SIGNAL ON ERROR はすべての正の戻りコードをトラップします。さらに、CALL ON FAILURE も SIGNAL ON FAILURE もセットされていない場合には、負の戻りコードもトラップします。

FAILURE

コマンドが戻り時に障害条件を示した場合に発生します。この条件は、そのコマンドを呼び出した文節の終わりで発生しますが、FAILURE 条件トラップがすでに遅延状態にある場合には無視されます。

CALL ON FAILURE および SIGNAL ON FAILURE は、コマンドからのすべての負の戻りコードをトラップします。

HALT

外部から割り込んでプログラムの実行を終了させようとする、発生します。この条件は、通常、外部割り込みが発生したときに処理されていた文節の終わりに発生します。

NOVALUE

初期設定されていない変数が以下のように使用された場合に発生します。

- 式の中の項として
- PARSE 命令の VAR サブキーワードに続く *name* として
- 解析テンプレート、PROCEDURE 命令、または DROP 命令の中の変数参照として

注: SIGNAL ON NOVALUE は、複合変数の語尾以外の、すべての初期設定されていない変数をトラップすることができます。

```
/* The following does not raise NOVALUE. */  
signal on novalue
```

```
a.=0
say a.z
say 'NOVALUE is not raised.'
exit

novalue:
say 'NOVALUE is raised.'
```

この条件を指定できるのは、**SIGNAL ON** の場合だけです。

SYNTAX

プログラムの実行中に言語処理エラーが検出された場合に発生します。このエラーには、本当の構文エラーおよび "実行時" エラー (非数値項で算術演算を試みた場合など) を含む、すべての種類の処理エラーがあてはまります。この条件を指定できるのは、**SIGNAL ON** の場合だけです。

条件トラップを **ON** または **OFF** を用いて参照すると、その条件トラップの前の状態 (**ON**、**OFF**、または **DELAY**、ならびに任意の *trapname*) が置き換えられます。したがって、**CALL ON HALT** によって現在の **SIGNAL ON HALT** を (またその逆に) 置き換える、新しいトラップ名を指定した **CALL ON** または **SIGNAL ON** によって前のトラップ名を置き換える、**OFF** 参照によって **CALL** または **SIGNAL** のためのトラップを使用不可にするといったことが行われます。

注: 条件トラップそれぞれの状態 (**ON**、**OFF**、または **DELAY**、および任意の *trapname*) は、サブルーチンに入るときに保管され、その後 **RETURN** 時に復元されます。つまり、呼び出し元でセットアップされた条件に影響を与えずに、**CALL ON**、**CALL OFF**、**SIGNAL ON** および **SIGNAL OFF** をサブルーチンの中で使用できるようになります。サブルーチン呼び出し時に保管される他の情報の詳細については、**CALL** 命令 (**CALL**) を参照してください。

関連資料

250 ページの『条件がトラップされたときのアクション』

条件トラップが現在使用可能 (**ON**) になっているときに、指定された条件が発生すると、通常の制御の流れの代わりに、**CALL trapname** 命令または **SIGNAL trapname** 命令が自動的に処理されます。

条件がトラップされないときのアクション

条件トラップが、現在、使用禁止になっている (**OFF**) のに、指定された条件が起こった場合のデフォルトのアクションは、条件によって異なります。

- **HALT** および **SYNTAX** の場合、プログラムの実行が終了し、通常は、起こったイベントの性質を記述するメッセージ (411 ページの『第 31 章 エラー番号とメッセージ』を参照してください) に、その条件が示されます。
- その他の条件の場合、条件は無視され、その状態は変わらずに **OFF** のままです。

条件がトラップされたときのアクション

条件トラップが現在使用可能 (**ON**) になっているときに、指定された条件が発生すると、通常の制御の流れの代わりに、**CALL trapname** 命令または **SIGNAL trapname** 命令が自動的に処理されます。

CALL ON または **SIGNAL ON** 命令の **NAME** サブキーワードの後に、*trapname* を指定できます。 *trapname* を指定しない場合は、条件自体の名前 (**ERROR**、**FAILURE**、**HALT**、**NOTREADY**、**NOVALUE**、または **SYNTAX**) が使用されます。

例えば、**call on error** 命令は、**ERROR** 条件の条件トラップを使用可能にします。条件が発生すると、**ERROR** という名前で識別されるルーチンの呼び出しが行われます。**call on error name commanderror** 命令は、トラップを使用可能にし、条件が発生すると **COMMANDERROR** ルーチン呼び出すことになります。

条件がトラップされた後のイベントのシーケンスは、次のように、**SIGNAL** が処理されるか **CALL** が処理されるかによって異なります。

- 取られるアクションが **SIGNAL** の場合は、現在の命令の実行が直ちに停止し、条件が使用不可になり (**OFF** に設定され)、通常の場合とまったく同様に **SIGNAL** が実行されます (**SIGNAL** を参照)。

新しい条件発生をトラップする場合は、ラベルに到達したら、条件を再び使用可能にするために、条件に対する新しい **CALL ON** 命令または **SIGNAL ON** 命令が必要になります。例えば、**SYNTAX** 条件が発生し

たときに SIGNAL ON SYNTAX が使用可能になっている場合、SIGNAL ON SYNTAX ラベル名が見つからなければ、通常の構文エラー終了になります。

- 取られるアクションが CALL の場合 (節の境界でのみ出現できる)、その CALL は、特殊変数 RESULT に影響を与えないことを除いて、通常の方法で行われます (CALL を参照)。ルーチンが何らかのデータを戻さなければならない場合、戻された文字ストリングは無視されます。

これらの条件 (ERROR、FAILURE、および HALT) は INTERPRET 命令の実行中に発生する可能性があるため、CALL ON が使用されていた場合は、INTERPRET の実行が中断され、後で再開されることがあります。

条件が発生すると、CALL が行われる前に、条件トラップが遅延状態になります。この状態は、CALL からの RETURN まで、あるいは条件に対して明示的な CALL (または SIGNAL) ON (または OFF) が行われるまで続きます。この遅延状態は、条件トラップを処理するために呼び出されたルーチンの開始時の早すぎる条件トラップを防止します。条件トラップは遅延状態のときは使用可能のままですが、条件が再び発生した場合は、無視されるか (ERROR、FAILURE、または NOTREADY の場合)、あるいは (その他の条件の場合) 以下のいずれかのイベントが発生するまでどのようなアクションも (条件情報の更新を含む) 遅らせられます。

1. 遅延条件に対する CALL ON または SIGNAL ON が処理される。この場合は、新しい CALL ON 命令または SIGNAL ON 命令が実行された直後に、CALL または SIGNAL が実行されます。
2. 遅延条件に対する CALL OFF または SIGNAL OFF が処理される。この場合は、条件トラップが使用不可になり、CALL OFF 命令または SIGNAL OFF 命令の終わりに、その条件に対するデフォルト・アクションが実行されます。
3. サブルーチンからの RETURN が行われる。この場合、条件トラップは遅延状態ではなくなり、直ちにサブルーチンが再度呼び出されます。

CALL からの RETURN では、実行の元の流れが再開されます (つまり、流れは CALL の影響を受けません)。

注:

1. 構文トラップ・ルーチンを書くときは、特に注意する必要があります。可能であれば、プログラムの先頭付近にルーチンを置いてください。そうする必要があるのは、終了コメントの欠落などの特定のスキャン・エラーがあると、トラップ・ルーチン・ラベルが見つからない可能性があるからです。また、エラーのあるプログラムをさらにスキャンさせる可能性のあるステートメントをトラップ・ルーチンに含めてはなりません。この例として、名前が引用符で囲まれていない組み込み関数の呼び出しがあります。組み込み関数名が大文字になっていて、引用符で囲まれている場合、REXX は、内部ラベルを探索せずに関数に直行します。
2. どのような場合でも、条件は検出されると直ちに発生します。SIGNAL ON が条件をトラップした場合、必要であれば現在の命令は終了させられます。したがって、実行中にイベントが発生した命令は、一部しか処理されない場合があります。例えば、割り当ての中の式の評価中に SYNTAX が発生した場合、割り当ては行われません。ERROR、FAILURE、HALT、および NOTREADY トラップに対する CALL は、節の境界でのみ出現できます。これらの条件が INTERPRET 命令の最中に起こった場合は、INTERPRET の実行が中断されて、後で再開される可能性があります。同様に、その他の命令 (例えば DO や SELECT) が、節の境界で CALL によって一時的に中断されることもあります。
3. 条件トラップそれぞれの状態 (ON、OFF、または DELAY、および任意の *trapname*) は、サブルーチンに入るときに保管され、その後 RETURN 時に復元されます。つまり、呼び出し元でセットアップされた条件に影響を与えずに、CALL ON、CALL OFF、SIGNAL ON および SIGNAL OFF をサブルーチンの中で使用できることになります。サブルーチン呼び出し時に保管される他の情報の詳細については、CALL 命令 (CALL) を参照してください。
4. CALL によって外部ルーチンが呼び出された場合、その外部ルーチンが REXX プログラムであっても、条件トラップの状態は影響を受けません。REXX プログラムに入るときは、どのような条件トラップでも、初期設定は OFF です。
5. 対話式トレース中にユーザー入力処理されている間は、すべての条件トラップが一時的に OFF に設定されます。これにより、例えば、SIGNAL ON NOVALUE がアクティブの間にユーザーが未初期化変数を誤って使用してしまった場合に、予期しない制御の移動を防止できます。同じ理由から、対話式トレース中に構文エラーがあっても、プログラムから出ることはならず、特別にトラップされて、メッセージが出された後は無視されます。

6. プログラム実行開始前またはプログラム終了後のどちらかのタイミングで、特定の実行エラーがシステム・インターフェースによって検出されます。 SIGNAL ON SYNTAX はこのようなエラーをトラップできません。

ラベルは単一シンボルとそれに続くコロンで構成される節であることに注意してください。任意の数の連続した節をラベルにすることができます。したがって、複数のラベルを別のタイプの節の前に置くことができます。

関連情報

[条件および条件トラップ](#)

条件情報

条件がトラップされ、SIGNAL または CALL が生じた場合は、これが現在のトラップされている条件となり、それと関連付けられた特定の条件情報が記録されます。

この情報は、CONDITION 組み込み関数を使用して検査することができます ([200 ページの『CONDITION』](#)を参照してください)。

条件情報には、下記のものがあります。

- 現在トラップされている条件の名前
- 条件トラップの結果として処理される命令 (CALL または SIGNAL)
- トラップされている条件の状況
- その条件に関連した記述ストリング

現在の条件情報は、条件トラップ (CALL ON または SIGNAL ON) の結果として制御がラベルに渡されたときに置き換えられます。条件情報は、保管され、CALL ON トラップによるものを含め、サブルーチン呼び出しまたは関数呼び出し全体で復元されます。したがって、CALL ON によって呼び出されたルーチンは該当する条件情報にアクセスすることができます。そのルーチンから戻った後は、以前の条件情報も使用可能のままになっています。

記述ストリング

記述ストリングは、トラップされている条件によって異なります。

ERROR

処理の結果、エラー条件になったストリング。

FAILURE

処理の結果、障害条件になったストリング。

HALT

停止要求と関連付けられた、あらゆるストリング。ストリングが提供されなかった場合は、ヌル・ストリングになる可能性があります。

NOVALUE

参照した結果、NOVALUE 条件になった変数の派生名。NOVALUE 条件トラップを使用可能にできるのは、SIGNAL ON を使用した場合のみです。

SYNTAX

言語処理プログラムがこのエラーに関連付けた、あらゆるストリング。特定のストリングを指定しなかった場合、これは、ヌル・ストリングになる可能性があります。特殊変数の RC および SIGL には、処理エラーの種類および位置に関する情報が入ります。SYNTAX 条件トラップを使用可能にできるのは、SIGNAL ON を使用した場合のみです。

特殊変数

特殊変数とは、REXX プログラムの処理中に自動的に設定される可能性があるものです。

特殊変数には、RC、RESULT、および SIGL の 3 つがあります。このどれにも初期値はありませんが、プログラムで変更することはできます。(RESULT については、[184 ページの『RETURN』](#)を参照してください。)

RC

ERROR および FAILURE の場合は、制御が条件ラベルに転送される前に、REXX 特殊変数 RC には通常通り、コマンドの戻りコードがセットされます。

SIGNAL ON SYNTAX の場合には、RC に構文エラー番号がセットされます。

SIGL

CALL または SIGNAL による制御が移動すると、それに続いて、制御を移動させる原因となった文節のプログラム行番号が特殊変数 SIGL に保管されます。制御の移動が条件トラップによるものであれば、SIGL には、CALL または SIGNAL が行われる前に (現在のサブルーチン・レベルで) 最後に処理された文節の行番号が割り当てられます。これは、特に SIGNAL ON SYNTAX で役立ちます。例えば、エラーのある行番号を使用してテキスト・エディターを制御する場合です。一般に、SYNTAX ラベルの後のコードが PARSE SOURCE を行ってデータのソースを見つけ、その後、エディターを呼び出して、エラーのあった行に配置されたソース・ファイルを編集します。この場合、エディターで行った変更は、プログラムを再実行しなければ有効にならないことがあります。

この他にも、下記の例のように、SIGL を使用すると、エラー (関数呼び出しの一時的な障害など) の原因の判別に役立ちます。

```
signal on syntax
a = a + 1      /* This is to create a syntax error */
say 'SYNTAX error not raised'
exit

/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  say 'REXX error' rc 'in line' sigl ':' 'ERRORTEXT'(rc)
  say "SOURCELINE"(sigl)
  trace ?r; nop
```

このコードは、最初に、エラー・コード、行番号、およびエラー・メッセージを表示します。このあと、エラーのあった行を表示し、最後にデバッグ・モードになって、エラーの行で使用された変数の値を調べます。

第 23 章 REXX/CICS テキスト・エディター

REXX/CICS は、CICS ベースの汎用テキスト・エディターを提供します。

このエディターは、EXEC およびデータを CICS 環境内から作成、更新、および表示できるようにするために提供されています。

REXX/CICS エディターには、接頭部コマンド (例えば、C、CC、M、MM、B、A、F、P) が複数組み込まれています。このエディターを使用するためには、EDIT と入力します。

エディター・セッションでは、コマンド行に入力されたコマンドは、各コマンドの間に「;」を置くことによってチェーニングすることができます。REXX で作成されたマクロについてもサポートが提供されます。これにより、プロファイル EXEC を使用したエディター設定のカスタマイズ、エディターへの新規コマンドの追加、またはアプリケーションの一部としてのエディター機能の使用が可能になります。

注: REXX/CICS テキスト・エディターでは、バイナリー・ファイルはサポートされません。

REXX/CICS テキスト・エディターの呼び出し

CICS 端末 (クリア画面) からの REXX/CICS 編集セッション、REXX/CICS セッション、別の編集セッション、または REXX EXEC を開始することができます。CICS EDIT トランザクション ID (または REXX/CICS エディター用のユーザーの地域別定義トランザクション ID) を実行すると、エディターに入れます。REXX/CICS のクリア画面から、EDIT を入力し、続いて REXX ファイル・システム (RFS) のファイル ID を入力すると、このファイルに対して編集セッションが発行されます。

注: EXEC 名なしの CICS トランザクション ID として REXX を指定すると、IBM 提供の REXXTRY 対話式ユーティリティ (CICRTRY EXEC) が発行されます。REXXTRY は、REXX ステートメントおよびコマンドを実行するための対話式シェルを提供します。

エディターで作業中に、EDIT *fileid* と入力すると、別の編集セッションを開始することができます。この呼び出し方式の構文は、EDIT コマンドで定義されています (264 ページの『EDIT』)。EXEC を作成する際に、EDIT CICS トランザクション ID をコマンドとして発行することで、REXX/CICS エディター・セッションを開始できます。エディターを呼び出すときにファイル ID が指定されない場合、ファイル ID は、デフォルトで現行 RFS ディレクトリー内のファイル NONAME に設定されます。完全修飾ファイル ID は次のものです。poolid:¥dir1¥dir2¥fn.ft

ただし、ターゲット・ファイルが (CD コマンドで指定されたとおりの) 現行ディレクトリー内の場合 (または現行ディレクトリー内に作成する予定の場合) は、部分ファイル ID を指定できます。

次の例は、編集セッションを開始するための 4 つの異なる方法を示しています。

```
EDIT TEST.EXEC (From terminal using CICS transaction ID EDIT)
EDIT TEST.EXEC (From within an edit session using the EDIT editor command)
'EDIT TEST.EXEC' (From within an exec using the EDIT command)
'EDIT TEST.EXEC' (From terminal using the REXX/CICS REXXTRY utility)
```

最初の例では、REXX/CICS からセッションを開始します。2 つ目の例では、既存のセッションから新規セッションを開始します。3 つ目は、REXX EXEC からセッションを開始し、4 つ目は、REXX/CICS REXXTRY ユーティリティからセッションを開始します。

画面フォーマット

プロファイルなしでエディターを呼び出すと、次の図のように、デフォルトの画面定義が表示されます。

```

EDIT ---- POOL1:¥USERS¥USER1¥TEST ----- COLUMN 1 73
COMMAND ==>
00000 ***** TOP OF DATA *****
00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011
00012
00013 ***** BOTTOM OF DATA *****

```

F1=HELP F2=LADD F3=FIL F4=SPLT F5=F F6=JN F7=BA F8=FWD F10=LFT F11=RGT F12=QUI

最初の行は、タイトル行用に予約されています。この行には、以下のものが含まれます。

- 完全修飾ファイル ID
- 桁番号
- 表示されるメッセージ。

新規編集セッションを開始した時点で表示されるのは、ファイルの始めと終わりの場所を示す情報行のみです。これらの行は、接頭部域とデータ域で構成されています。この例では、「*** TOP OF DATA ***」が表示されている行が現在行です。これは、強調表示によって他の行と区別され、コマンド行コマンドのほとんどが作用する行です。コマンド行を使用すると、高いレベルでエディターと対話することができます。タイトル行を除き、画面の行をすべて、便宜上、移動させることができます。

接頭部コマンド

エディターには、接頭部コマンドがいくつか用意されています。これらのコマンドを接頭部域に入力すると、個別ブロック単位または連続ブロック単位での行のコピー、移動、挿入、削除、および複製が可能になります。

- 行単位のコマンド。

以下のコマンドは、個別行で機能するものであり、以下の 1 文字で構成されます。

/	ファイル内の現在行を指定します
I	行を 1 行挿入します
D	行を 1 行削除します
C	行を 1 行コピーします
M	行を 1 行移動します
R	行を 1 行複製します
"	複製の同義語

行の接頭部域に上記コマンドのいずれかを入力すると、そのコマンドは、それぞれの機能をその行に対して実行します。行 00000 の接頭部域に「I」を入力すると、エディターは、入力できるように、その行の

直ぐ下に新しい行を開きます。接頭部コマンドの末尾に数値を付加することもできます。これは、複製係数として機能します。数値「5」を「I」に付加すると、1行ではなく、5行が入力用に開かれます。

- 連続ブロック・コマンド。

以下のコマンドは、連続する行ブロックで機能するものであり、2文字で構成されます。

DD

行のブロックを削除します

CC

行のブロックをコピーします

MM

行のブロックを移動します

RR

行のブロックを複製します

""

複製の同義語

ブロック・コマンドはペアで処理されます。ブロックの最初の行の接頭部域にコマンドを1つ置き、そのブロックの最後の行の接頭部域に同じコマンドを置きます。例えば、行のブロックを削除するには、行 00001 の接頭部域に「DD」を入れ、さらに行 00005 の接頭部にも「DD」を入れます。これにより、行 00001 から行 00005 までのすべての行が削除されます。複製係数を指定できるブロック接頭部コマンドは、複製コマンドのみです。したがって、複製ブロック・コマンドと一緒に数値を指定する場合には、ブロック (個別行ではありません) を複製する回数を把握しておく必要があります。複製係数は、最初の RR 複製コマンドで指定する必要があります。

- 宛先コマンド。

以下のコマンドは、宛先接頭部コマンドと呼ばれます。

A

変更後

B

変更前

F

後続

P

先行

これらのコマンドは、移動接頭部コマンドおよびコピー接頭部コマンドに、テキストのブロックの宛先を提供します。それらを接頭部域に入力すると、コピーまたは移動からのテキストが、指定される宛先コマンドに応じて、その行の前または後に配置されます。複製接頭部コマンドでは、宛先接頭部コマンドは使用しません。代わりに、その出力を、複製対象となるブロックの直ぐ後に置きます。

REXX/CICS エディターでのマクロ

このエディターは、REXX マクロをサポートしており、マクロが、エディター設定を変更してエディター画面を表示できるようにします。マクロは、すべてのエディター・コマンド行コマンドを処理することができます。

以下の例は、エディター・コマンド環境をアドレッシングし、エディター設定を変更します。

```
/* Macro to alter the setting of the REXX/CICS editor */
ADDRESS EDITSVR
'SET NUMBERS OFF'
'SET CURLINE 10'
'SET MSGLINE 2'
'SET CMDLINE TOP'
'SET CASE MIXED IGNORE'
```

次の例は、3つの別々の行にテキスト **Some**、**More**、**DATA** を入力した後、現在の編集画面を表示します。

```
/* Macro to use the REXX/CICS editor as an I/O interface */
ADDRESS EDITSVR
'INPUT Some'
'INPUT More'
'INPUT DATA'
'DISPLAY'
```

コマンド行コマンド

コマンド行コマンドのそれぞれの構文と説明を記述します。

ARBCHAR

ARBCHAR は、任意文字を設定します。

➡ ARBCHAR — *arbchar* ➡

オペランド

arbchar

印刷可能 (キーボードで打てる) 文字を指定します。

戻りコード

0

正常な戻り

202

無効なオペランド

例

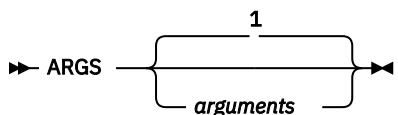
```
'ARBCHAR .'
```

この例は、文字「**.**」を任意文字として定義します。

注: 任意文字は、ストリング内でテキストの代わりをします。任意文字のデフォルト値は「**.**」です。任意文字を使用して検索を実行する方法については、**FIND** コマンド [267 ページ](#)の『**FIND**』を参照してください。

ARGS

ARGS は、テキスト・エディター EXEC コマンドで呼び出されたときに編集集中のプログラムに渡されるデフォルトのパラメーターを格納します。



注:

¹ *arguments* が指定されない場合、以前に定義された引数はすべて削除されます。

オペランド

arguments

渡されるパラメーター・ストリングを指定します。*arguments* を指定しない場合、以前に定義された引数はすべて削除されます。

戻りコード

0

Normal return

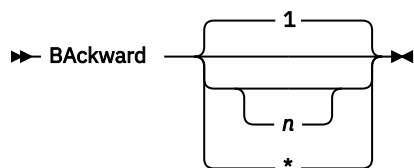
例

```
'ARGS A B C'  
'EXEC'  
'ARGS'
```

この例の最初の行は、A、B、およびCとして渡される引数を定義します。2番目の行は、現在編集集中のファイルの、最後に保管されたコピーを実行し、行1で定義された引数をそれに渡します。最後の行は引数を削除します。

BACKWARD

BACKWARD は、指定された数の画面表示の分だけ、ファイルの始め方向へ上方スクロールします。



オペランド

n

上方スクロールする画面表示数を指定します。アスタリスク (*) を指定すると、画面はファイルのトップまでスクロールし、現在の行はファイルのトップに設定されます。*n* が指定されない場合、画面は、1 表示分上方にスクロールします。

戻りコード

0

Normal return

202

無効なオペランド

例

```
'BACKWARD'
```

この例は、ファイルのトップ方向へ1画面スクロールします。

注: エディターは、デフォルトで、PF7 を BACKWARD に、PF8 を FORWARD に設定します。

BOTTOM

BOTTOM は、ファイルの終わりまでスクロールします。

➡ BOTTOM ➡

戻りコード

0

Normal return

例

```
'BOTTOM'
```

この例は、ファイルの終わりまでスクロールします。

CANCEL

CANCEL は、変更を保管せずに、現行編集セッションを終了します。

➡ CANCEL ⬅

戻りコード

0

Normal return

210

要求は失敗しました

例

```
'CANCEL'
```

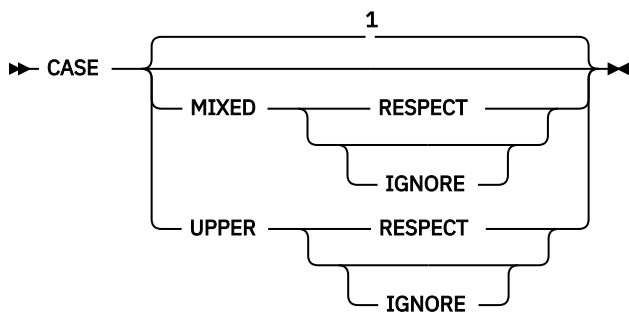
この例は、ファイルの変更を保管せずに現行エディター・セッションを無条件に終了します。

注：

1. CANCEL を指定すると、変更を保管せずにエディターを終了することができ、変更が行われたことを示す警告メッセージは出されません。
2. CANCEL は QQUIT の同義語です。

CASE

CASE は、大/小文字の変換と解釈の方法を設定します。



注:

¹ デフォルトは、ユーザー・プロファイルに設定されます。

オペランド

UPPER

入力された時点で小文字を大文字に変換します。

MIXED

各文字を、その元の形式で処理します。

RESPECT

検索時に、各文字の大/小文字を尊重します。

IGNORE

検索時に、各文字の大/小文字を無視します。

戻りコード

0

通常要求

202

無効なオペランド

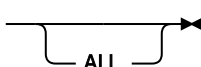
例

```
'CASE MIXED RESPECT'
```

この例は、大/小文字を「MIXED」に設定し、大/小文字の区別を「RESPECT」に設定します。大/小文字の区別について詳しくは、[FIND コマンド \(267 ページの『FIND』\)](#) を参照してください。

CHANGE

CHANGE は、ファイル内のストリングを変更します。

➡ CHANGE — /string1/string2/ 

オペランド

string1

置換するストリングを指定します。

string2

string1 に取って代わるストリングを指定します。

ALL

現在行からファイルの終わりまでのすべての行におけるすべてのオカレンスを変更されることを示すキーワードです。

/

区切り文字です。

戻りコード

0

Normal return

202

無効なオペランド

210

要求は失敗しました

223

検索指数が見つかりません

例

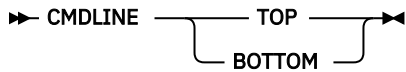
```
'CHANGE /noeditor/editor/'
```

この例は、noeditor の最初に現れる位置を editor に置き換えます。

注: CHANGE コマンドは、string1 の探索時および string2 の変更時に、エディターの感度の設定値を、CASE コマンドまたはシステム・デフォルトによって定義されたとおりに尊重します。

CMDLINE

CMDLINE は、コマンド行の表示設定を設定します。



オペランド

TOP

コマンド行を、画面の 2 行目に表示します。

BOTTOM

コマンド行を、画面の最終行に表示します。

戻りコード

0

Normal return

202

無効なオペランド

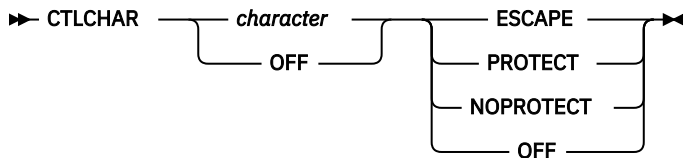
例

```
'CMDLINE TOP'
```

この例は、コマンド行を画面の 2 行目に表示します。

CTLCHAR

CTLCHAR は、制御文字の機能を設定します。



オペランド

文字

使用する制御文字を指定します。

OFF

文字を指定せずに OFF が使用された場合、制御文字のすべての定義をドロップし、指定された場合は、特定の文字をドロップします。

ESCAPE

RESERVED コマンドに渡される、ストリング内の後続文字が制御文字であることを指定します。

PROTECT

RESERVED コマンドに渡されるストリングがユーザー入力から保護されることを指定します。

NOPROTECT

RESERVED コマンドに渡されるユーザー入力がこのストリングで許可されることを指定します。

戻りコード

0

Normal return

202

無効なオペランド

例

```
'CTLCHAR ! ESCAPE'  
'CTLCHAR % PROTECT'  
'RESERVED 20 HIGH !% Important Info'
```

この例は、！をエスケープ文字として、また、%をフィールド保護文字として定義します。これらのコマンドを入力した後、画面の行 20 は保護され、制御文字 !% に続くテキストが含まれています。

CURLINE

CURLINE は、現在行表示設定を設定します。

➡ CURLINE — *number* ➡

オペランド

number

画面の行番号を指定します。

戻りコード

0

Normal return

202

無効なオペランド

例

```
'CURLINE 3'
```

この例は、現在の表示行を画面行 3 に設定します。

注：現在行は、このコマンドに指定された画面行番号に表示されます。ただし、現在行は行 1 には表示できません。それは、行 1 がタイトル行用に予約されているためです。

DISPLAY

DISPLAY は、現行編集画面を表示します。

➡ DISPLAY ➡

戻りコード

0

Normal return

210

要求は失敗しました

例

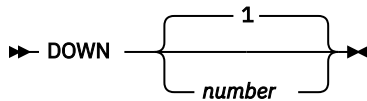
```
'DISPLAY'
```

この例は、現行編集画面を表示します。

注：DISPLAY コマンドは、マクロから実行された場合にのみ、有効です。現行編集セッションの画面が表示されます。通常の端末編集セッションから実行された場合、顕著な効果はありません。

DOWN

DOWN は、ファイル内で下方にスクロールします。



オペランド

number

スクロールする行数を指定します。*number* が指定されない場合、画面は 1 行だけ下へ移動します。

戻りコード

0

Normal return

202

無効なオペランド

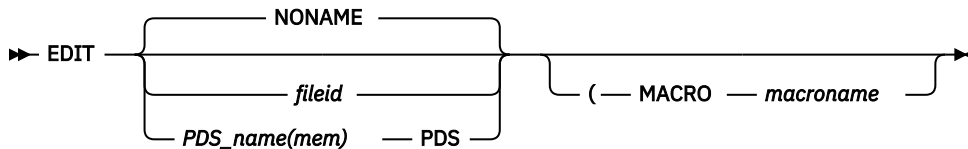
例

```
'DOWN 5'
```

この例は、ファイルを 5 行下方スクロールします。

EDIT

EDIT は、新規編集セッションを開きます。



オペランド

fileid

作成または編集するファイルのファイル ID を指定します。

PDS_name(mem)

編集する完全修飾 MVS PDS 名およびメンバー名を指定します。

PDS

PDS の編集時に PDS メンバー名の後続くキーワードです。

MACRO

編集中のファイルに適用される命令のグループを指定するキーワードです。

macroname

プロファイル・マクロ・ファイル ID (REXX EXEC 名) のファイル名部分を指定します。

戻りコード

0

Normal return

203

ファイルが見つかりません

204

許可されません

211

ファイル ID が無効です

226

ファイルは現在、編集集中です

299

内部エラー

1748

Task Input/Output Table (TIOT) に DD 名のエントリーがありません

1749

複合データ・セットにはエクスポートできません

1750

DD 名として複数のデータ・セットが連結されています

例

```
'EDIT TEST.EXEC'
```

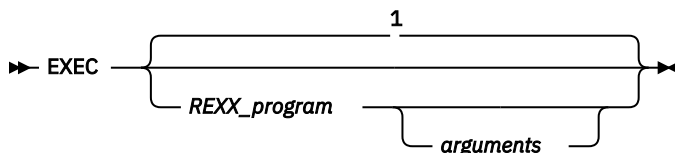
この例は、ファイル TEST.EXEC の編集セッションを開きます。

注:

1. ファイルの編集時に使用されるディレクトリーは、次のように決定されます。
 - 完全修飾ディレクトリー ID が明示的に指定された場合には、それが常に使用されます。
 - 部分修飾ファイル ID が明示的に指定された場合、ファイル ID がそのディレクトリー名を使用して完全に解決されていれば、それに一致する既存のファイルを見つけるのに現行ディレクトリーおよびパスが検索されます。
 - そのような一致が見つかった場合には、編集セッションは既存のファイルが対象となり、ファイル ID は、そのファイルが入っていたディレクトリーを使用して完全に解決されます。
 - ファイルの配置が検索順になっていない場合、(CD コマンドで指定されたとおりの) 現行作業ディレクトリーを指定する完全解決ファイル ID で、新しいファイルのための編集セッションが作成されます。
2. MVS PDS メンバーを区別するには、メンバー名の後にキーワード PDS を使用します。メンバーは、括弧で囲む必要があり、引用符は許されません。
3. エディターが呼び出そうとするデフォルトのユーザー・プロファイル・マクロは CICEPROF です。CICEPROF マクロは、ISPF/PDF に似た環境を作成します。CICXPROF という名前の 2 つ目のプロファイル・マクロが提供されます。CICXPROF は、VM/CMS XEDIT に似た環境を作成します。
4. ファイル ID または PDS 名が指定されない場合、特殊名 NONAME を持つ RFS ファイルが作成されます。

EXEC

EXEC は、エディター・セッション内で REXX プログラムを実行します。



注:

¹ REXX_program が指定されない場合、現在編集集中のファイルの最後に保管されたコピーが実行されます。

オペランド

REXX_program

実行する REXX プログラム名を指定します。名前を指定しない場合、現在編集集中のファイルの最後に保管されたコピーが実行されます。ARGS コマンドを使用して引数が定義されている場合には、それが渡されます。

arguments

REXX_program オペランドに指定されたプログラムに渡される引数を指定します。

戻りコード

n

呼び出された EXEC の出口によって設定された戻りコード。[172 ページの『EXIT』](#)を参照してください。

0

Normal return

-3

EXEC が見つかりません

-10

EXEC 名が指定されていません

-11

EXEC 名が無効です

-12

GETMAIN error

-99

内部エラー

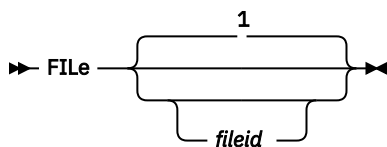
例

```
'EXEC TEST1.EXEC X Y Z'
```

この例は、プログラム TEST1.EXEC を実行し、X、Y、および Z を引数として渡します。

FILE

FILE は、編集集中の現行ファイルを保管します。



注:

¹ *fileid* が指定されなかった場合、ファイルはデフォルトのファイル ID として保管されます。

オペランド

fileid

ファイルのファイル ID を指定します。*fileid* を指定しなかった場合、ファイルはデフォルトのファイル ID として保管されます。

戻りコード

0

Normal return

202

無効なオペランド

204

許可されません

207

ファイル・プールのスペースが不十分です

210

要求は失敗しました

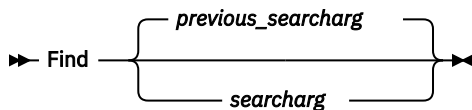
例

```
'FILE'
```

この例は、編集セッションの現行ファイル ID 指定を使用して、編集中の現行ファイルを保管します。編集セッションの作成時に、最初に、現行ファイル ID が、編集コマンドで指定されたファイル ID から取得されます。

FIND

FIND は、ファイル内でテキストのストリングを検出します。



オペランド

searcharg

検索を行うテキスト・ストリングを指定します。*searcharg* を指定しない場合、検索は、前の検索ストリング (*previous_searcharg*) で実行されます。

戻りコード

0

Normal return

202

無効なオペランド

223

検索引数が見つかりません

例

```
'FIND REDT'
```

この例は、REDT が最初に現れる位置を検索します。

```
'FIND Redt'
```

CASE が RESPECT に設定されている場合、この例では REDT が最初に現れる位置は検出されません。Redt が最初に現れる位置が検出されます。詳しくは、CASE コマンド [260 ページの『CASE』](#)を参照してください。

searcharg には任意の文字を入れることができます。この場合、任意の文字とは、その文字位置に埋め込まれる可能性のあるテキスト・ストリングを表します。

```
'FIND ONE.THREE'
```

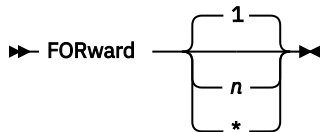
この例は、別のストリングによって ONE と THREE が結合された、任意のストリングが最初に現れる位置を検索します。

注:

1. CASE コマンドを使用して RESPECT フラグを設定すると、*searcharg* の大/小文字は区別されます。
2. 検索は現在行から開始され、BOTTOM OF DATA に達するか、または一致があるまで下方向に続けられます。一致なしで BOTTOM OF DATA に達すると、BOTTOM OF DATA が現在行になるのではなく、現在行は、FIND が処理される前の位置のままになります。

FORWARD

FORWARD は、指定された数の画面表示の分だけ、ファイルの終わり方向へ下方スクロールします。



オペランド

n

下方スクロールする画面表示数を指定します。アスタリスク (*) を指定すると、画面はファイルの終わりまでスクロールし、現在の行はデータの最後の行に設定されます。*n* が指定されない場合、画面は、1 表示分下方にスクロールします。

戻りコード

0

Normal return

202

無効なオペランド

例

```
' FORWARD '
```

この例は、ファイルの終わり方向へ 1 画面スクロールします。

注: エディターは、デフォルトで、PF7 を BACKWARD に、PF8 を FORWARD に設定します。

GET

GET は、RFS ファイルを現行編集セッションにインポートします。



オペランド

fileid

ファイルのファイル ID を指定します。

戻りコード

0

Normal return

203

ファイルが見つかりません

204

許可されません

210

要求は失敗しました

例

```
'GET POOL1:¥USERS¥USER1¥TEST.EXEC'
```

この例は、現在の行の後に、REXX ファイル・システム・ファイル TEST.EXEC を入れます。

GETPDS

GETPDS は、MVS PDS から現行編集セッションにメンバーをインポートします。ファイルは、現在行の後に挿入されます。

➡ GETPDS — *PDS_name(mem)* ➡

オペランド

PDS_name(mem)

完全修飾 MVS PDS とメンバーの名前を指定します。

戻りコード

0

Normal return

203

ファイルが見つかりません

204

許可されません

210

要求は失敗しました

例

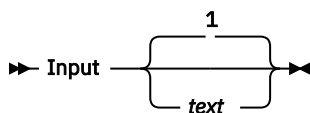
```
'GETLIB MYFILE.PROJ1(MEM1)'
```

この例は、編集セッションで、PDS メンバー MEM1 を取得し、それを現在行の後に置きます。

注: GETPDS を使用してデータ・セットのメンバーを取得できるのは、CICS 領域に外部セキュリティー・マネージャーがデータ・セットへのアクセスを許可していて、CICS サインオンがデータ・セット高位接頭部と一致している場合のみです。

INPUT

INPUT は、現在行の後に新しい行を挿入します。



注:

¹ *text* が指定されない場合、新しい行はブランクになります。

オペランド

text

新しい行に挿入されるテキストを指定します。*text* を指定しないと、新しい行はブランクになります。

戻りコード

0

Normal return

例

```
'INPUT Test Input Data'
```

この例は、現在行の後の、新しく挿入された行にテキスト Test Input Data を入れます。

JOIN

JOIN は、2 行を 1 行に結合します。

➡ JOIN ⇐

戻りコード

0

Normal return

210

要求は失敗しました

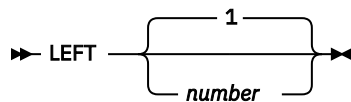
例

```
' JOIN '
```

この例は、カーソルが置かれている行を、その直後の行と結合します。

LEFT

LEFT はファイル内で左へスクロールします。



オペランド

number

スクロールする文字数を指定します。*number* が指定されない場合、画面は、ファイル内で左へ 1 文字スクロールします。*number* に 0 を指定すると、ファイルは左端までスクロールします。

戻りコード

0

Normal return

202

無効なオペランド

210

要求は失敗しました

例

```
' LEFT 20 '
```

この例は、左へ 20 文字スクロールします。

LINEADD

LINEADD は、カーソル行の後にブランク行を追加します。

▶▶ LINEADD ◀◀

戻りコード

0

Normal return

230

カーソルがファイル域内にありません

例

```
'PFKEY 2 LINEADD'
```

この例では、PF2 が押されるたびに、カーソルが置かれている行 (それがファイル行である場合) の後にブランク行が追加されます。

注: LINEADD は、プログラム・ファンクション (PF) キーに割り当てるのが最良です。デフォルトで PF2 に割り当てられます。

LPREFIX

LPREFIX は、接頭部コマンドを現在行の接頭部域に入れます。

▶▶ LPREFIX — *prefix* ◀◀

オペランド

接頭部

編集セッション中に入力されたあらゆる標準接頭部 (C、CC、M、MM、B、A など) を指定します。

戻りコード

0

Normal return

202

無効なオペランド

例

```
'LPREFIX D'
```

この例では、現在のファイル行が削除されます。

注: LPREFIX は、編集マクロ内から接頭部コマンドを使用できるようにするために提供されています。

MACRO

MACRO はマクロを呼び出します。

▶▶ MACRO — *fileid* ◀◀

オペランド

fileid

実行するマクロのファイル ID を指定します。このファイル ID にファイル・タイプ接尾部が含まれている場合は、その接尾部を持つ EXEC を呼び出す試みが行われます。そうでない場合は、接尾部が EXEC である EXEC を呼び出す試みが行われます。

戻りコード

n

呼び出された EXEC の出口によって設定された戻りコード。[172 ページの『EXIT』](#)を参照してください。

0

Normal return

-3

EXEC が見つかりません

-10

EXEC 名が指定されていません

-11

EXEC 名が無効です

-12

GETMAIN error

-99

内部エラー

例

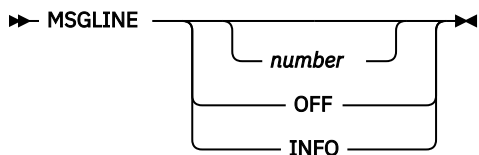
```
'MACRO POOL1:¥USERS¥USER1¥TEST'
```

この例は、マクロ POOL1:¥USERS¥USER1¥TEST.EXEC を呼び出します。

注: マクロは、REXX/CICS エディター・サーバーへの呼び出しを行うことができます。エディターのコマンド行から入力できるコマンドはいずれも、マクロから実行できます。

MSGLINE

MSGLINE は、メッセージ行の表示設定を設定します。



オペランド

number

メッセージ行を、該当の画面行に表示します。

OFF

メッセージ行を表示しません。

INFO

メッセージを、ヘッダー行に表示します。

戻りコード

0

Normal return

202

無効なオペランド

例

```
'MSGLINE 2'
```

この例は、メッセージ行を画面行 2 に置きます。

NULLS

NULLS は、画面上のフィールドを、末尾ブランクで書き込むのか、末尾ヌルで書き込むのかを制御します。



オペランド

ON

画面上のフィールドを末尾ヌルで書き込むことを指定します。

OFF

画面上のフィールドを末尾ブランクで書き込むことを指定します。

戻りコード

0

Normal return

202

無効なオペランド

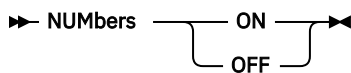
例

```
'NULLS ON'
```

この例では、画面のフィールドに末尾ヌルが設定されます。

NUMBERS

NUMBERS は、接頭部域の表示設定を設定します。



オペランド

ON

順序番号を接頭部域に表示します。

OFF

等号を接頭部域に表示します。

戻りコード

0

Normal return

202

無効なオペランド

例

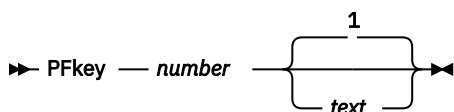
```
'NUMBERS ON'
```

この例は、接頭領域に順序番号を表示します。

注：行番号の順序付けは、編集セッション内のデータに対しては行われず、疑似行番号が編集セッション中のファイル行にのみ関連付けられます。

PFKEY

PFKEY は、プログラム・ファンクション (PF) キーを設定または処理します。



注:

¹ *text* を指定しない場合、PF キーが処理されます。

オペランド

number

設定または処理される PF キーを指定します。

text

PF キーが設定されるテキストを指定します。 *text* を指定しない場合、PF キーが処理されます。

戻りコード

0

Normal return

202

無効なオペランド

229

範囲外の数値

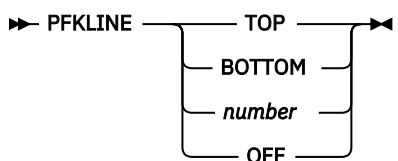
例

```
'PFKEY 3 quit'  
'PFKEY 3'
```

この例では、最初に PFKEY 3 を quit に設定してから、PF キーを処理します。

PFKLINE

PFKLINE は、プログラム・ファンクション (PF) キーの行表示設定を設定します。



オペランド

TOP

画面の 2 行目に PF キー行を表示します。

BOTTOM

画面の最終行に PF キー行を表示します。

number

画面の行番号を指定します。

OFF

表示画面から PF キーを除去します。

戻りコード**0**

Normal return

202

無効なオペランド

例

```
'PFKLINE BOTTOM'
```

この例は、画面の最終行に PF キー行を表示します。

QQUIT

QQUIT は、変更を保管せずに、現行編集セッションを終了します。

➡ QQuit ⬅

戻りコード**0**

Normal return

例

```
'QQUIT'
```

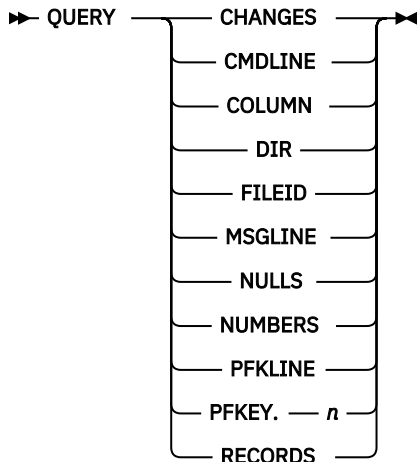
この例は、ファイルの変更を保管せずに現行エディター・セッションを無条件に終了します。

注：

1. QQUIT を指定すると、変更を保管せずにエディターを終了することができ、変更が行われたことを示す警告メッセージは出されません。
2. QQUIT は CANCEL の同義語です。

QUERY

QUERY は、エディターの現在の設定を表示します。



オペランド

CHANGES

ファイルが最後に保管された後にそのファイルに対して行われた変更の数を表示します。

CMDLINE

コマンド行の現在の設定を表示します。詳細については、テキスト・エディター・コマンドの [262 ページ](#) の『[CMDLINE](#)』を参照してください。

COLUMN

画面上に表示される、ファイル内の開始列を表示します。

DIR

ファイルと関連付けられているディレクトリーを表示します。

FILEID

編集中的ファイルの名前を表示します。

MSGLINE

メッセージ行の現在の設定を表示します。詳細については、テキスト・エディター・コマンドの [272 ページ](#) の『[MSGLINE](#)』を参照してください。

NULLS

NULLS の現在の設定を表示します。詳細については、テキスト・エディター・コマンドの [273 ページ](#) の『[NULLS](#)』を参照してください。

NUMBERS

NUMBERS の現在の設定を表示します。詳細については、テキスト・エディター・コマンドの [273 ページ](#) の『[NUMBERS](#)』を参照してください。

PFKEYLINE

PFKEYLINE の現在の設定を表示します。詳細については、テキスト・エディター・コマンドの [274 ページ](#) の『[PFKEYLINE](#)』を参照してください。

PFKEY.n

PFKEY *n* が押された場合に処理されるコマンドを表示します。*n* は、1 から 24 までの任意の数値です。

RECORDS

ファイル内の行数を表示します。

戻りコード

0

Normal return

202

無効なオペランド

236

Not defined

例

```
'QUERY PFKEY.1'
```

この例は、PFKEY 1 が押された場合に処理されるコマンドを表示します。

QUIT

QUIT は、現行編集セッションを終了します。

➡ QUIT ➡

戻りコード

0

Normal return

210

要求は失敗しました

例

```
'QUIT'
```

この例は、エディターを終了します。

注：現行ファイルが変更されている場合、エディターは、保管が行われるか、または QQUIT コマンドが入力されるまで、ユーザーによる終了を許可しません。

RESERVED

RESERVED は、出力用の画面の 1 行を予約します。

➡ REServed — *line* —

HIGH
NOHIGH
OFF

 — *text* ➡

オペランド

line

予約済みで、テキストが表示される行を指定します。

HIGH

テキストが強調表示されることを指定するキーワードです。

NOHIGH

テキストが通常の輝度であることを指定するキーワードです。

OFF

その行が予約済み状態から解放されることを指定するキーワードです。

text

予約された行に表示される、オプションの制御文字付きのテキストのストリングを指定します。

戻りコード

0

Normal return

202

無効なオペランド

210

要求は失敗しました

229

範囲外の数値

例

```
'CTLCHAR ! ESCAPE'  
'CTLCHAR % PROTECT'  
'RESERVED 20 HIGH !% Important Info'
```

この例は、画面の行 20 の保護フィールド Important Info を高輝度で表示します。

RESET

RESET は、保留中の接頭部コマンドを終了します。

➡ RESET ⇐

戻りコード

0

Normal return

例

```
'RESET'
```

この例は、保留中の接頭部コマンドをすべて取り消します。

RIGHT

RIGHT はファイル内で右へスクロールします。

➡ R~~I~~ght { 1
 number }

オペランド

number

スクロールする文字数を指定します。*number* が指定されない場合、画面は、ファイル内で右へ 1 文字スクロールします。*number* に 0 を指定すると、ファイルは右端までスクロールします。

戻りコード

0

Normal return

202

無効なオペランド

例

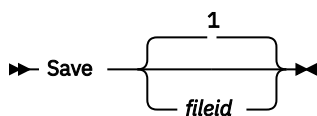
```
'RIGHT 20'
```

この例は、ファイル内で右へ 20 文字スクロールします。

注：ユーザーが指定した値によりターゲットがレコードの外側になってしまう場合には、スクロールはレコードの右側で停止します。

SAVE

SAVE は、ファイルを RFS ファイルまたは PDS メンバーに保管します。



注:

¹ *fileid* が指定されなかった場合、ファイルはデフォルトのファイル ID として保管されます。

オペランド

fileid

ファイルのファイル ID を指定します。*fileid* を指定しなかった場合、ファイルはデフォルトのファイル ID として保管されます。

戻りコード

0

Normal return

202

無効なオペランド

207

ファイル・プールのスペースが不十分です

210

要求は失敗しました

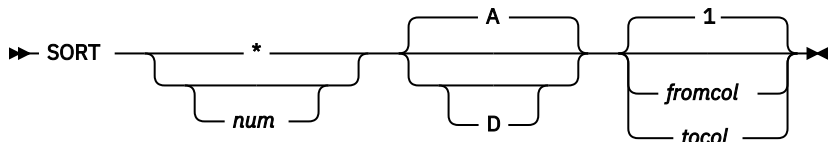
例

```
'SAVE SYSTEM:¥USERS¥USER1¥TEST.EXEC'
```

この例は、現行ファイルを RFS に保管し、それに SYSTEM:¥USERS¥USER1¥TEST.EXEC という名前を付けます。

SORT

SORT は、現在行から下の行をソートします。



オペランド

*

現在行からファイルの終わりまでのすべての行がソートされることを指定します。

num

現在行から値 *num* の行がソートされることを指定します。

A

行が昇順でソートされることを指定します。(これはデフォルトです。)

D

行が降順でソートされることを指定します。

fromcol

行が、この桁で始まるデータに格納されることを指定します。*fromcol tocol* を指定しない場合は、最初の桁からソートが開始されます。

tocol

行が、この桁で終わるデータに格納されることを指定します。

戻りコード

0

Normal return

202

無効なオペランド

229

範囲外の数値

例

```
'SORT * A 5 10'
```

この例は、ファイル内のすべての行を、現在行から下へソートし、桁 5 から桁 10 までソートされます。

注: 多くの行数をソートする場合、ソートの動作速度は非常に遅くなります。

SPLIT

SPLIT は 1 行を 2 行に分割します。

➡ SPLIT ⇐

戻りコード

0

Normal return

210

要求は失敗しました

例

```
'SPLIT'
```

この例では、カーソルが置かれている行が 2 つの行に分割されます。1 つの行に、その行のカーソルより左にあるすべてのテキストが含まれ、後に続く行には、残りのテキスト (カーソルの下および右にあるもの) が含まれます。

STRIP

STRIP は、すべてのファイル行から末尾ブランクを取り除きます。

➡ STRIP ⇐

戻りコード

0

Normal return

例

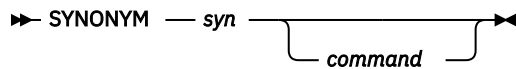
```
'STRIP'
```

この例は、各ファイル行のすべての末尾ブランクを除去します。

注: 区分データ・セットではブランクが埋め込まれるので、このコマンドはデータ・セットを区分データ・セットからインポートしてから RFS 内に保存する場合に特に便利です。

SYNONYM

SYNONYM は、他の任意の有効なコマンドにコマンド・アクションを割り当てます。



オペランド

syn

同義語であるコマンド・アクションを実行する任意の有効なコマンドを指定します。

コマンド

任意の有効なコマンドを指定します。

戻りコード

0

Normal return

例

```
'SYNONYM GL GETLIB'
```

この例は、GL をコマンド GETLIB と同等のものにします。

TOP

TOP は、ファイルのトップまでスクロールします。

```
➡ TOP ➡
```

戻りコード

0

Normal return

例

```
'TOP'
```

この例は、ファイルのトップまでスクロールします。

TRUNC

TRUNC は、ファイル内の各行を、指定された長さまで切り捨てます。

➡ TRUNC — *column* ➡

オペランド

column

保持する最後の桁を指定します。

戻りコード

0

Normal return

202

無効なオペランド

例

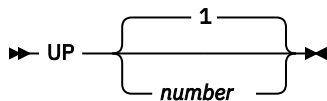
```
'TRUNC 72'
```

この例は、ファイル内のすべての行を 72 文字の長さに切り捨てます。

注: このコマンドは、除去を必要とするシーケンス番号を持つデータ・セットを処理する場合に有用です。エディターでは現在、ファイルでのシーケンス番号の配置や維持はサポートされません。

UP

UP は、ファイル内を上方に (ファイルのトップ方向へ) スクロールします。



オペランド

number

スクロールする行数を指定します。*number* を指定しなかった場合、デフォルトのスクロール移動量は 1 に設定されます。

戻りコード

0

Normal return

202

無効なオペランド

例

```
'UP 20'
```

この例は、ファイル内で 20 行上方にスクロールします。

第 24 章 REXX/CICS ファイル・システム

REXX ファイル・システム (RFS) は、REXX/CICS エディターで作成されるテキスト・ファイルおよび EXEC のストレージ用に提供されるもので、REXX ファイル・システムの外側からインポートされた RFS コマンドおよびデータを使用する EXEC によって提供されます。

RFS は、拡張対話式エグゼクティブ (AIX) ファイル・システムおよび OS/2 ファイル・システムをモデルとしています。これらの環境から得られた主な着想として、ディレクトリー概念があります。各ディレクトリーをいくつかのサブディレクトリーに分割することにより、階層型の編成が得られます。

ユーザーは、RFS コマンドを使用して、RFS 機能にアクセスすることができます。RFS コマンドは、ファイル入出力、RFS コマンドを実行する機能を提供するだけでなく、最も重要なファイル・システムの保守をユーザーが行えるようにします。

ファイル・リスト・ユーティリティ (FLST) は、REXX ファイル・システムへのフルスクリーン・インターフェースとして提供されます。このユーティリティは、他の CICS 作業を実行できるプラットフォームとしても使用できます。

ファイル・プール、ディレクトリー、ファイル

ファイル・プールは、一連の VSAM ファイルです。各ファイル・プールに、ルート・ディレクトリーがあります。ファイルは、データ・ファイルまたは REXX EXEC です。

プール内の最初の VSAM ファイルには、プールに関する情報が含まれており、フォーマット設定してからでないと、データをプールに書き込むことができません。プール内のその他の VSAM ファイルはすべて、この最初のファイルの拡張であるので、アクセス方式サービス・プログラムおよび CICS に対して割り振られ、定義された後でプールの VSAM ファイルのリストで定義するだけで済みます。VSAM ファイルを、随意に、ファイル・プールに追加して、そのプール内に追加のストレージ・スペースを用意することができます。ただし、ファイル・プールへのスペースの定義と追加は、REXX/CICS 管理者または CICS システム・プログラマーが行う作業です。ファイル・プールには、1 文字から 8 文字の固有の名前が与えられます。

ファイル・プールのルート・ディレクトリーの名前は、*file_pool_name:¥* です。このディレクトリーには、ファイルだけでなく、サブディレクトリーも含めることができます。サブディレクトリーとは、別のディレクトリー内のディレクトリーです。サブディレクトリーは、他のいずれのディレクトリー内にも作成できます。新規ディレクトリーは、RFS MKDIR コマンドを使用して作成できます。ルート以外のディレクトリーはすべて、1 文字から 8 文字のディレクトリー・ファイル名で識別されます。ディレクトリー・ファイル名は、ピリオドを使用して、オプションの 1 文字から 8 文字のディレクトリー・ファイル・タイプを結合することができます。これをディレクトリー ID といいます。

ファイル・プールは、ユーザー・ファイル・プールまたは非ユーザー・ファイル・プールとして定義することができます。すべてのユーザー・ファイル・プールに存在する必要がある 1 つのディレクトリーが USERS ディレクトリーです。このディレクトリーには、システム上のユーザーに対応する複数のサブディレクトリーが含まれています。ファイルを初めて RFS に保管すると、USERS ディレクトリー内に新しいサブディレクトリーが作成されます。ユーザーが CICS にサインオンしている場合、この新規ディレクトリーには、ご使用のユーザー ID を使用して名前が付けられます。そうでない場合、ディレクトリー名はデフォルトの CICS DFLTUSER 値が取られます。このディレクトリーが作成されると、その個人用ディレクトリー内に任意の数のサブディレクトリーを作成できます。作成したディレクトリーのどれにでも、ファイルを入れることができます。

ファイルは、データ・ファイルまたは REXX EXEC です。これらには、ディレクトリーと同じ命名規則 (ファイル名と、オプションのファイル・タイプをピリオドで結合する) が使用されます。REXX EXEC のデフォルトのファイル・タイプは EXEC です。ファイルは、REXX/CICS エディターまたは RFS を DISKW コマンドを使用して作成されます。

完全修飾ファイル ID は、ファイル・プール名と、その後に :¥ が続きます。パス内の各ディレクトリーは後に ¥ と、ファイル名およびオプションのファイル・タイプから成るファイル ID が続きます。

例

以下の例は、完全修飾ファイル ID を示しています。POOL1 はファイル・プール名であり、USERS および USER1 はディレクトリー ID、また、TEST.EXEC はファイル ID です。

```
POOL1:¥USERS¥USER1¥TEST.EXEC
```

以下の例は、ファイル・プール、ディレクトリー、およびファイルを示しています。

POOL1:	File Pool
¥	Root Directory
TEST1.EXEC	File
USERS¥	Subdirectory
USER1¥	Subdirectory
TEST2.EXEC	File
DOCS¥	Subdirectory
TEST3.DOCUMENT	File
USER2¥	Subdirectory
LETTER.DOCUMENT	File
PROJECT1¥	Subdirectory
PROD1.EXEC	File
DATA¥	Subdirectory
PROD1.DATA	File
WORK:¥	File Pool and Root Directory
TEST1.DATA	File
CHARTS¥	Subdirectory
CHART1.DATA	File
CHART2.DATA	File

この例は、2 つのファイル・プール (POOL1 および WORK) を示しています。ファイル・プール POOL1 には、ファイル (TEST1.EXEC) と 2 つのサブディレクトリー (USERS および PROJECT1) が含まれています。USERS サブディレクトリーの中には、ユーザー ID (USER1 および USER2) に対応する 2 つのサブディレクトリー (USER1 および USER2) が入っています。ユーザー USER1 には、そのディレクトリーの中にファイル (TEST2.EXEC) およびサブディレクトリー (DOCS) があります。DOCS サブディレクトリー内に別のファイル (TEST3.DOCUMENT) が入っています。ユーザー USER2 には、そのディレクトリーの中にファイル (LETTER.DOCUMENT) があります。ファイル・プール WORK には、ファイル (TEST1.DATA) およびサブディレクトリー (CHARTS) が含まれています。サブディレクトリー CHARTS の中には、2 つのファイル (CHART1.DATA および CHART2.DATA) が入っています。

現行ディレクトリーとパス

現行ディレクトリーは、現行作業ディレクトリーであり、REXX ファイル・システム (RFS) での作業時には最初に検索されます。

現行ディレクトリーは、CD コマンド [351 ページの『CD』](#) を使用して設定できます。CD コマンドの形式は、DOS などのオペレーティング・システムの CD コマンドとほぼ同じです。構文は、CD の後に、部分修飾または完全修飾のディレクトリー名が続きます。サブディレクトリーから親ディレクトリーに戻すには、CD .. と入力します。別のサブディレクトリーに変更する場合は、CD の後に、該当のサブディレクトリー名を続けます。

例

以下の例では、最初のコマンドが現行ディレクトリーを POOL1:¥USERS¥USER1 に設定し、2 番目のコマンドが現行ディレクトリーを POOL1:¥USERS¥USER1¥DOCS に設定します。3 番目のコマンドは、現行ディレクトリーを POOL1:¥USERS¥USER1 に戻します。

```
'CD POOL1:¥USERS¥USER1'  
'CD DOCS'  
'CD ..'
```

PATH コマンドは、現行ディレクトリーが検索された後で、REXX EXEC の検索順序を定義するために使用されます。詳細については、[394 ページの『PATH』](#) を参照してください。構文は、PATH の後にリストがスペースで区切られて続いている、完全修飾ディレクトリー名です。

以下の例は、最初に、現行ディレクトリーを設定し、次に、検索順序を定義します。

```
'CD POOL1:¥USERS¥USER1¥EXEC'  
'PATH POOL1:¥ POOL1:¥USERS¥USER1'  
'EXEC TEST2.EXEC'
```

EXEC 名は完全修飾で、個々のディレクトリーの検索が実行される前に各ディレクトリーのディレクトリー ID を検索で使用します。完全修飾名は以下のとおりです。

```
'POOL1:¥USERS¥USER1¥EXECSTEST2.EXEC'  
'POOL1:¥TEST2.EXEC'  
'POOL1:¥USERS¥USER1¥TEST2.EXEC'
```

REXX/CICS コマンド EXEC が呼び出されると、上記の 3 つのディレクトリーすべてが検索され、REXX/CICS により POOL1:¥USERS¥USER1 内の EXEC が検出されます。TEST2.EXEC が POOL1:¥ ディレクトリー内に存在していた場合、RFS は、それが検出された時点で検索を停止します。検索順で最初に検出されたコピーがアクセスされます。

注：ファイル名が完全修飾になっていない場合、RFS は、現行ディレクトリーから始まる、EXEC を探索する探索順序に従います。最初に検出されたコピーが実行されます。何も見つからなかった場合には、ターゲット・ファイルも EXEC も見つからなかったことを示すエラーが戻されます。

セキュリティ

REXX/CICS ファイル・システム・セキュリティには、ファイル・アクセス・セキュリティとコマンド実行セキュリティという 2 種類の一般的タイプがあります。

- ファイル・アクセス・セキュリティは、EXEC およびデータへのアクセスを制御します。RFS ファイル・セキュリティは、2 つのレベル、つまり、CICS レベルと RFS ディレクトリー・レベルで制御できます。

1. CICS レベルでは、ファイル・プール VSAM ファイルにアクセスする権限を特定のユーザーに付与することができます。これは、ハイレベルのセキュリティを付与します。
2. RFS ディレクトリー・レベルでは、ユーザー・ディレクトリーは専用ディレクトリーであり、アクセスできるのは所有ユーザーのみです (デフォルト)。

ただし、ディレクトリーの所有者は、RFS AUTH コマンドを使用して、ディレクトリーを「publicr」、
「publicw」、または「secured」として定義できます。publicr は、他のすべての REXX/CICS ユーザーがこのディレクトリーに対して読み取り専用アクセス権限を持つことを意味します。publicw は、他のすべての REXX/CICS ユーザーがこのディレクトリーに対して読み取り/書き込みアクセス権限を持つことを意味します。secured は、アクセスを許可するかどうかを判別するために RFS セキュリティ出口ルーチンが呼び出されることを意味します。詳しくは、RFS AUTH コマンドの [286 ページの『AUTH』](#) を参照してください。非ユーザー・ディレクトリーは作成可能であり、そのアクセス・レベルは許可ユーザーによって定義されます。

- コマンド実行セキュリティは、特定の REXX/CICS コマンドや、コマンド・キーワードの使用を制御します。詳しくは、[セキュリティ](#)を参照してください。

RFS コマンド

RFS コマンド環境では、RFS とのインターフェースを取るためにコマンドを発行します。コマンド環境を RFS に設定する場合、RFS コマンドの前に RFS を指定してはなりません。

例

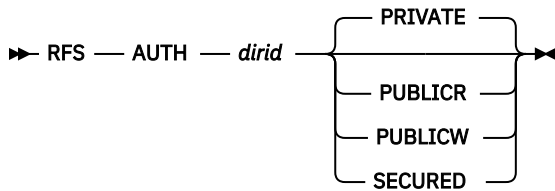
```
'RFS DISKR POOL1:¥USERS¥USER1¥TEST.EXEC DATA.'
```

この例は、RFS ファイル TEST.EXEC の内容を、REXX 複合変数 DATA に読み取ります。TEST.EXEC は、完全修飾ディレクトリー POOL1:¥USERS¥USER1¥ 内にあります。

RFS コマンドの構文は、以下のとおりです。

AUTH

AUTH は、以下のように、RFS ディレクトリーへのアクセスを許可します。



オペランド

dirid

REXX ファイル・システムのディレクトリー ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページ](#)の『CD』を参照してください。

PRIVATE

ディレクトリーの所有者のみが、ファイルへの読み取り/書き込みアクセスを持つことを指定します。これはデフォルトです。

PUBLICR

あらゆるユーザーがディレクトリー内のファイルに対して読み取り専用アクセス権限を持つことを指定します。

PUBLICW

あらゆるユーザーがディレクトリー内のファイルに対して読み取り/書き込みアクセス権限を持つことを指定します。

SECURED

外部セキュリティー・マネージャーがディレクトリー内のファイルに対するアクセス権限を付与することを指定します。

戻りコード

RFS コマンド [397 ページ](#)の『RFS』を参照してください。

例

```
'RFS AUTH POOL1:¥USERS¥USER1¥DOCS PUBLICR'
```

この例は、ディレクトリー DOCS を共通ディレクトリーにします。すべてのユーザーが、ディレクトリー DOCS 内のファイルに対して読み取り専用アクセス権限を持ちます。

CKDIR

CKDIR は、既存の RFS ディレクトリー・レベルがあるか調べます。

```
➡ RFS — CKDIR — dirid ➡
```

オペランド

dirid

REXX ファイル・システムのディレクトリー ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページ](#)の『CD』を参照してください。

戻りコード

RFS コマンドの [397 ページ](#)の『RFS』を参照してください。

例

```
'RFS CKDIR POOL1:¥USERS¥USER1¥DOCS'
```

この例は、既存のディレクトリー POOL1:¥USERS¥USER1 内の DOCS というディレクトリーがあるかどうかを調べます。

CKFILE

CKFILE は、存在するファイル ID が、指定されたものなのか、部分修飾なのか、それとも完全修飾なのかを確認するために検査します。

➡ RFS — CKFILE — *fileid* ➡

オペランド

fileid

ファイル ID を指定します。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

```
'CKFILE POOL1:¥USERS¥USER1¥TEST.EXEC'
```

この例は、既存のディレクトリー POOL1:¥USERS¥USER1 内に TEST.EXEC というファイルがあるかどうか調べます。

COPY

COPY はファイルをコピーします。

➡ RFS — COPY — *fileid1* — *fileid2* ➡

オペランド

fileid1

ソース・ファイル ID を指定します。完全修飾または部分修飾のディレクトリー ID およびファイル ID を使用できます。

fileid2

ターゲット・ファイル ID を指定します。完全修飾または部分修飾のディレクトリー ID およびファイル ID を使用できます。

注：ターゲット・ファイル (*fileid2*) が既に存在している場合、*fileid1* の内容がそれに取って代わります。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

```
'RFS COPY POOL1:¥USERS¥USER1¥TEST1.EXEC POOL1:¥USERS¥USER1¥TEST2.EXEC'
```

この例は、ディレクトリー POOL1:¥USERS¥USER1 内の TEST1.EXEC を TEST2.EXEC にコピーします。

DELETE

DELETE は RFS ファイルを削除します。

➡ RFS — DELETE — *fileid* →

オペランド

fileid

削除されるファイルの名前を指定します。部分修飾にすることも完全修飾にすることもできます。詳しくは、CD コマンドの [351 ページの『CD』](#) を参照してください。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

```
'RFS DELETE POOL1:¥USERS¥USER1¥TEST1.EXEC'
```

この例は、ディレクトリー POOL1:¥USERS¥USER1 内のファイル TEST1.EXEC を削除します。

DISKR

DISKR は、RFS ファイルからレコードを読み取ります。

➡ RFS — DISKR — *fileid* — { *DATA.*
stem. } →

オペランド

fileid

ファイル ID を指定します。

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。 [155 ページの『語幹』](#) を参照してください。デフォルトの語幹は DATA. です。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

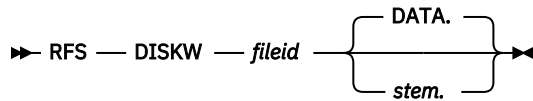
```
'RFS DISKR POOL1:¥USERS¥USER1¥TEST.DATA DATA.'
```

この例では、RFS ファイル POOL1:¥USERS¥USER1¥TEST.DATA の内容全体を DATA. に格納します。REXX 複合変数です。

注: DATA.0 を、ファイルから読み取られるレコードの数に設定します。DATA.*n* に、ファイルから読み取られた *n* 番目のレコードが含まれます。

DISKW

DISKW は、レコードを語幹から RFS ファイルに書き込みます。ファイルは、語幹内のデータでオーバーレイされます。



オペランド

fileid

ファイル ID を指定します。

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#)を参照してください。デフォルトの語幹は DATA. です。

戻りコード

[397 ページの『RFS』](#)を参照してください。

例

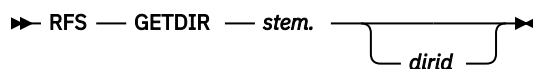
```
'RFS DISKW POOL1:¥USERS¥USER1¥TEST.EXEC DATA.'
```

この例は、REXX 複合変数 DATA. の内容を RFS ファイル POOL1:¥USERS¥USER1¥TEST.EXEC に格納します。

注: DATA.0 を、ファイルに書き込まれるレコードの数に設定します。

GETDIR

GETDIR は、現行ディレクトリーまたは指定されたディレクトリーの内容のリストを、指定された REXX 配列に戻します。



オペランド

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#)を参照してください。

dirid

REXX ファイル・システム・ディレクトリーのレベル ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページの『CD』](#)を参照してください。

戻りコード

RFS コマンドの [397 ページの『RFS』](#)を参照してください。

例

```
'RFS GETDIR DIRDOC. POOL1:¥USERS¥USER1¥DOCS'
```

この例は、ディレクトリー DOCS の内容を DIRDOC に配置します。REXX 複合変数です。

MKDIR

MKDIR は、新しい RFS ディレクトリー・レベルを作成します。

➡ RFS — MKDIR — *dirid* ➡

オペランド

dirid

REXX ファイル・システムのディレクトリー ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページの『CD』](#) を参照してください。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

```
'RFS MKDIR POOL1:¥USERS¥USER1¥DOCS'
```

この例は、既存のディレクトリー POOL1:¥USERS¥USER1 内に DOCS という新しいディレクトリーを作成します。

注: 許可ユーザーのみが、それぞれの ¥USERS¥userid ディレクトリー構造の外側にディレクトリーを作成できます。

RDIR

RDIR は、指定された RFS ディレクトリーを除去します。

➡ RFS — RDIR — *dirid* ➡

オペランド

dirid

REXX ファイル・システムのディレクトリー ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページの『CD』](#) を参照してください。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

```
'RFS RDIR POOL1:¥USERS¥USER1¥DOCS'
```

この例は、既存のディレクトリー POOL1:¥USERS¥USER1 内の DOCS というディレクトリーを削除します。

RENAME

RENAME は、RFS ファイルの名前を新しい名前に変更します。

➡ RFS — RENAME — *fileid1* — *fileid2* ➡

オペランド

fileid1

ソース・ファイル ID を指定します。完全修飾または部分修飾のディレクトリー ID およびファイル ID を使用できます。

fileid2

ソース・ターゲット・ファイル ID を指定します。完全修飾または部分修飾のディレクトリー ID およびファイル ID を使用できます。

注: ターゲット・ファイル (*fileid2*) が既に存在している場合、*fileid1* の内容がそれにとって代わります。

戻りコード

RFS コマンドの [397 ページの『RFS』](#) を参照してください。

例

```
'RFS RENAME POOL1:¥USERS¥USER1¥TEST1.EXEC POOL1:¥USERS¥USER1¥TEST2.EXEC'
```

この例は、ファイル POOL1:¥USERS¥USER1¥TEST1.EXEC の名前を POOL1:¥USERS¥USER1¥TEST2.EXEC に変更します。

REXX/CICS ファイル・リスト・ユーティリティー

ファイル・リスト・ユーティリティー (FLST) は、REXX ファイル・システムへのフルスクリーン・インターフェースを提供します。

実行中、FLST は RFS の管理、EXEC の呼び出し、またはトランザクションの開始を行います。REXX、RFS、および CICS への高水準インターフェースになるようになっています。

呼び出し

FLST を実行する場合は、クリア済みの CICS 画面またはクリア済みの REXX/CICS 画面に進み、FLST と入力すると、FLST が実行し始めます。

FLST の画面フォーマットは、以下のとおりです。

```
USER=USER1 - DIRECTORY=¥USERS¥USER1
CMD  FILENAME FILETYPE ATTRIBUTES RECORDS SIZE  DATE      TIME
     TEST1    EXEC    FILE        11      1   1994/03/27 10:30:29
     TEST2    EXEC    FILE         5      1   1994/03/27 10:31:04
```

```
COMMAND ==>
F1=HELP F2=REFRESH F3=END F7=UP 18 F8=DOWN 18 F11=EDIT F12=CANCEL
```

ご使用のユーザー ID が左上隅に表示されます。現行ディレクトリーは、ユーザー ID の横に表示されます。画面のその他の部分は、REXX/CICS エディター・セッションに極めて似ています。FLST は、すべての入出力に対してエディターを使用します。ただし、エディターで RESERVED コマンドを使用して表示される最初の 2 行は除きます。入出力シェルに REXX/CICS エディターを使用するメリットは、大きいディレクトリー内でファイル名またはファイル・タイプを検索できることと、そのディレクトリーをディスク上のファイルに保管できることです。

REXX/CICS ファイル・リスト・ユーティリティーでのマクロ

REXX ファイル・リスト・ユーティリティーは、REXX マクロをサポートしており、マクロが FLST 設定の変更と FLST 画面の表示を行えるようにします。マクロは、すべての FLST コマンドを処理できます。

例

以下の例では、FLST 環境に対処し、FLST 設定を変更します。

```
/* Macro to set some FLST settings */  
ADDRESS FLSTSVR  
'SET PFKEY 11 EDIT'  
'SET PFKEY 12 CANCEL'  
'SYNONYM DISCARD RFS DELETE'
```

FLST コマンド

このセクションでは、FLST コマンドについて説明します。これらのコマンドは、ソース FLST コマンド列上のどこにでも、あるいはコマンド行から入力することができます。

注：データが複数の場所に入力される場合、プログラム・ファンクション・キーが優先され、次に、コマンド行に入力されたデータ、最後が、コマンド列に入力されたデータの順です。

CANCEL

CANCEL は、コマンド列内のいかなるコマンドも実行せずに終了します。

CANCEL をコマンド行から入力する場合は、以下の構文を使用します。

➡ CANCEL ➡

COPY

COPY はファイルをコピーします。

COPY を FLST コマンド列に入力する場合は、以下の構文を使用します。

➡ COPY — / — *fileid* ➡

オペランド

fileid

結果が配置されるファイルのファイル ID を指定します。

注：*fileid* が既に存在している場合は、置き換えられます。

例

```
'COPY / TEST3.EXEC'
```

この例は、TEST1.EXEC の横のコマンド列から実行され、TEST1.EXEC と同じ新規ファイル TEST3.EXEC を作成します。

COPY をコマンド行から入力する場合は、以下の構文を使用します。

➡ COPY — *fileid1* — *fileid2* ➡

オペランド

fileid1

コマンドが処理するファイルのファイル ID を指定します。

fileid2

結果が配置されるファイルのファイル ID を指定します。

注：*fileid2* が既に存在する場合は、*fileid1* の内容がそれに取って代わります。

例

```
'COPY TEST1.EXEC TEST3.EXEC'
```

この例は、コマンド行から実行されると、TEST1.EXEC と同じ新規ファイル (TEST3.EXEC) を作成します。

DELETE

DELETE はファイルを削除します。

DELETE を FLST コマンド列に入力する場合は、以下の構文を使用します。

➡ DELETE ➡

DELETE をコマンド行から入力する場合は、以下の構文を使用します。

➡ DELETE — *fileid* ➡

オペランド

fileid

コマンドが処理するファイルのファイル ID を指定します。

例

```
'DELETE TEST1.EXEC'
```

この例は、コマンド行から実行され、ファイル TEST1.EXEC を削除します。

DOWN

DOWN は、1 行以上、下にスクロールします。

DOWN をコマンド行から入力する場合は、以下の構文を使用します。

➡ DOWN — *n* ➡

オペランド

n

下にスクロールする行数を指定します。

例

```
'DOWN 5'
```

この例は、リストを 5 行下方にスクロールします。

END

END は、入力されたすべてのコマンドを実行し、END がコマンド行に入力されるか、PF キーとして使用されると、終了します。

END をコマンド行から入力する場合は、以下の構文を使用します。

➡ END ➡

EXEC

EXEC は EXEC を実行してから、終了します。

EXEC を FLST コマンド列に入力する場合は、以下の構文を使用します。

➡ EXEC — */ parameter* ➡

オペランド

parameter

EXEC に引数として渡されるパラメーターを指定します。

例

```
'EXEC / PARMS'
```

この例は、TEST3.EXEC の横のコマンド列で実行され、EXEC の TEST3.EXEC を実行し、PARMS を引数として渡します。

EXEC をコマンド行から入力する場合は、以下の構文を使用します。

➡ EXEC — *fileid1* ————— ➡
 parameter

オペランド

fileid

コマンドが処理するファイルのファイル ID を指定します。

parameter

EXEC に引数として渡されるパラメーターを指定します。

戻りコード

n

呼び出された EXEC の出口によって設定された戻りコード。[172 ページの『EXIT』](#)を参照してください。

0

Normal return

-3

EXEC が見つかりません

-10

EXEC 名が指定されていません

-11

EXEC 名が無効です

-12

GETMAIN error

-99

内部エラー

例

```
'EXEC TEST3 PARMS'
```

この例は、EXEC の TEST3.EXEC を実行し、PARMS を引数として渡します。

FLST

FLST は、ファイル・リスト・ユーティリティを呼び出します。

FLST をコマンド行から入力する場合は、以下の構文を使用します。

➡ FLST ————— ➡
 dirid

オペランド

dirid

完全ディレクトリーまたは部分ディレクトリーの ID を指定します。*dirid* を指定しない場合、FLST は、デフォルトで現行作業ディレクトリーに設定されます。

例

```
'FLST'
```

この例は、現行作業ディレクトリーのメンバーのファイル・リストを表示します。

注: REXX ファイル・システムについて詳しくは、[283 ページの『第 24 章 REXX/CICS ファイル・システム』](#)を参照してください。

MACRO

MACRO はマクロを呼び出します。

MACRO をコマンド行から入力する場合は、以下の構文を使用します。

➡ MACRO — *fileid* ➡

オペランド

fileid

実行するマクロのファイル ID を指定します。このファイル ID にファイル・タイプ接尾部が含まれている場合は、その接尾部を持つ EXEC を呼び出す試みが行われます。そうでない場合は、接尾部が EXEC である EXEC を呼び出す試みが行われます。

戻りコード

n

呼び出された EXEC の出口によって設定された戻りコード。[172 ページの『EXIT』](#)を参照してください。

0

Normal return

-3

EXEC が見つかりません

-10

EXEC 名が指定されていません

-11

EXEC 名が無効です

-12

GETMAIN error

-99

内部エラー

例

```
'MACRO POOL1:¥USERS¥USER1¥TEST'
```

この例は、マクロ POOL1:¥USERS¥USER1¥TEST.EXEC を呼び出します。

注: マクロは、REXX/CICS FLST サーバーへの呼び出しを行うことができます。FLST のコマンド行から入力できるコマンドはいずれも、マクロから実行できます。

PFKEY

PFKEY は、プログラム・ファンクション (PF) キーを設定または処理します。

PFKEY をコマンド行から入力する場合は、以下の構文を使用します。

➡ PFkey — *number* — *text* ➡

オペランド

number

設定または処理される PF キーを指定します。

text

PF キーが設定されるテキストを指定します。

例

```
'PFKEY 3 quit'  
'PFKEY 3'
```

この例は、最初に PFKEY 3 を quit に設定してから、PF キーを処理します。

注: *text* を指定した場合、PF キーはテキストを使用して設定されます。*text* を指定しない場合、PF キーが処理されます。

REFRESH

REFRESH は、ファイル・リストを最新表示します。

REFRESH を FLST コマンド列に入力する場合は、以下の構文を使用します。

➡ REFRESH — *dirid* ➡

オペランド

dirid

REXX ファイル・システムのディレクトリー ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページの『CD』](#) を参照してください。

例

```
'REFRESH'
```

この例は、現行作業ディレクトリーのメンバーのファイル・リストを最新表示します。

RENAME

RENAME はファイルの名前を変更します

RENAME を FLST コマンド列に入力する場合は、以下の構文を使用します。

➡ RENAME — / — *fileid* ➡

オペランド

fileid

新しいファイル ID を指定します。

注: *fileid* が既に存在している場合は、置き換えられます。

例

```
'RENAME / TEST4.EXEC'
```

この例は、TEST3.EXEC の横のコマンド列から実行され、TEST3.EXEC の名前を TEST4.EXEC に変更します。

RENAME をコマンド行から入力する場合は、以下の構文を使用します。

➡ RENAME — *fileid1* — *fileid2* →

オペランド

fileid1

コマンドが処理するファイルのファイル ID を指定します。

fileid2

新しいファイル ID を指定します。

注：*fileid2* が既に存在する場合は、*fileid1* の内容がそれにとって代わります。

例

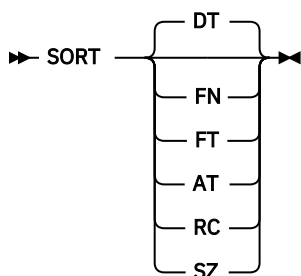
```
'RENAME TEST3.EXEC TEST4.EXEC'
```

この例は、コマンド行から実行され、ファイル TEST3.EXEC の名前を TEST4.EXEC に変更します。

SORT

SORT は、ファイル・リストをソートします。

SORT をコマンド行から入力する場合は、以下の構文を使用します。



オペランド

DT

ファイルを日付/時刻でソートすることを指定します。(これはデフォルトです。)

FN

ファイルをファイル名でソートすることを指定します。

FT

ファイルをファイル・タイプでソートすることを指定します。

AT

ファイルを属性でソートすることを指定します。

RC

ファイルをレコード数でソートすることを指定します。

SZ

ファイルをサイズでソートすることを指定します。

例

```
'SORT FN'
```

この例は、ファイル・リストをファイル名でソートします。

SYNONYM

SYNONYM は、他の任意の有効なコマンドにコマンド・アクションを割り当てます。

SYNONYM をコマンド行から入力する場合は、以下の構文を使用します。

➡ SYNONYM — *syn* ————— ➡
 command

オペランド

syn

同義語であるコマンド・アクションを実行する任意の有効なコマンドを指定します。

コマンド

任意の有効なコマンドを指定します。

例

```
'SYNONYM DISCARD RFS DELETE'
```

この例は、DISCARD を RFS コマンド DELETE と同等のものにします。

UP

UP は、1 行以上、上にスクロールします。

UP をコマンド行から入力する場合は、以下の構文を使用します。

➡ UP ————— ➡
 n

オペランド

n

上方にスクロールする行数を指定します。

例

```
'UP 5'
```

この例は、リストを 5 行上方にスクロールします。

FLST 戻りコード

FLST は、別の指定がない限り、REXX/CICS ファイル・システムの標準的な戻りコードを使用します。

詳しくは、RFS コマンドの [397 ページ](#)の『RFS』を参照してください。

FLST からの EXEC およびトランザクションの実行

ほとんどの REXX/CICS EXEC は、EXEC 名を入力するだけで、FLST から実行することができます。引数を持つ EXEC 名をコマンド行から入力するか、または EXEC 名を FLST コマンド列に入力します。後者の場合は、ファイル ID がトランザクションの引数として使用されます。REXX EXEC は、コマンド行で(この場合、ファイル ID が必要です)、または FLST コマンド列で EXECUTE コマンドを発行することによって実行されます。

第 25 章 REXX/CICS List System

REXX/CICS は、仮想記憶域内のデータのテーブルまたはリストを保守するための機能を提供します。この機能は、REXX List System (RLS) と呼ばれます。

このシステムでは、一時システムおよびユーザー情報のリストを管理することができます。RLS にアクセスするための外部コマンドは、REXX ファイル・システムで使用される RFS コマンドおよび CD コマンドではなく、RLS コマンドおよび CLD コマンドです。また、RLS は、EXEC 用ではなく、データ専用です。

ディレクトリーおよびリスト

RLS にはルート・ディレクトリーが 1 つあります。RLS システムにアクセスするには、*CICREX* という名前の CICS 一時記憶域キューからそのアンカー・アドレスを読み取ります。

この *CICREX* キューでは、RLS の最初のアクセスの後に項目が 1 つ含まれています。この項目は、6 つのフルワードから成る領域のアドレスが含まれています。これには、RLS 制御情報と、ルート・ディレクトリーへのポインターが含まれています。ルート・ディレクトリー RLS の名前は ¥ です。このディレクトリーには、リスト、保管済み変数、またはキューと呼ばれる特別なリストのほか、サブディレクトリーを含めることができます。サブディレクトリーとは、別のディレクトリー内のディレクトリーです。サブディレクトリーは、他のいずれのディレクトリー内にも作成できます。新しいディレクトリーは、RLS MKDIR コマンドを使用して作成できます。ルート以外のすべてのディレクトリーは、1 文字から 250 文字のディレクトリー・ファイル名によって識別されます。これをディレクトリー ID といいます。

常に存在する 1 つの RLS ディレクトリーが、USERS ディレクトリーです。このディレクトリーには、システム上のユーザーに対応するいくつかのサブディレクトリーが含まれています。この RLS にリストを初めて保管すると、USERS ディレクトリー内に新しいサブディレクトリーが作成されることがあります。ユーザーが CICS にサインオンしている場合、この新規ディレクトリーには、ご使用のユーザー ID を使用して名前が付けられます。そうでない場合、ディレクトリー名はデフォルトで CICS DFLTUSER 内の値に設定されます。このディレクトリーが作成されると、その個人用ディレクトリー内に任意の数のサブディレクトリーを作成することができます。作成したディレクトリーのいずれかにリストを入れることができます。

リストは常に、データ・ファイルです。これらでは、ディレクトリーと同じ命名規則が使用されます。

完全修飾リスト ID は、¥、後に ¥ が続く、パス内の各ディレクトリーの ID、およびリスト ID で構成されています。

例

以下の例では、完全修飾リスト ID が表示されます。USERS および USER1 はディレクトリー ID であり、TEST.DATA はリスト ID です。

```
¥USERS¥USER1¥TEST.DATA
```

以下の例では、RLS ディレクトリーおよびリストが示されます。

¥	Root Directory
TEST1.DATA	File
USERS¥	Subdirectory
USER1¥	Subdirectory
TEST2.DATA	File
DOCS¥	Subdirectory
TEST3.DOCUMENT	File
USER2¥	Subdirectory
LETTER.DOCUMENT	File
PROJECT1¥	Subdirectory
PROD1.INFO	File
DATA¥	Subdirectory
PROD1.DATA	File
¥	Root Directory
TEST1.DATA	File
CHARTS¥	Subdirectory
CHART1.DATA	File
CHART2.DATA	File

この例は、リスト・ディレクトリー構造を示します。ルート・ディレクトリーには、ファイル (TEST1.DATA) およびサブディレクトリー (USERS および PROJECT1) が含まれています。USERS サブディレクトリーの中には、ユーザー ID (USER1 および USER2) に対応する 2 つのサブディレクトリー (USER1 および USER2) が入っています。ユーザー USER1 には、そのディレクトリーの中にリスト (TEST2.DATA) とサブディレクトリー (DOCS) があります。DOCS サブディレクトリーの中には、別のリスト (TEST3.DOCUMENT) があります。ユーザー USER2 には、そのディレクトリーの中にファイル (LETTER.DOCUMENT) があります。ルート・ディレクトリーには、ファイル (TEST1.DATA) およびサブディレクトリー (CHARTS) が含まれています。サブディレクトリー CHARTS の中には、2 つのファイル (CHART1.DATA および CHART2.DATA) が入っています。

現行ディレクトリーとパス

現行リスト・ディレクトリーは、現行作業ディレクトリーであり、REXX List System (RLS) で作業する際には検索順序で最初になります。

現行リスト・ディレクトリーは、CLD コマンド [367 ページの『CLD』](#) を使用して設定できます。構文は、CLD の後に、完全修飾または部分修飾のディレクトリー名が続きます。サブディレクトリーから親ディレクトリーに戻るには、CLD .. と入力します。別のサブディレクトリーに変更するためには、CLD の後にサブディレクトリー名を続けます。

例

以下の例では、最初のコマンドは現行ディレクトリーを ¥USERS¥USER1 に、また、2 番目のコマンドは現行ディレクトリーを ¥USERS¥USER1¥DOCS にそれぞれ設定します。3 番目のコマンドは、現行ディレクトリーを ¥USERS¥USER1 に戻します。

```
CLD ¥USERS¥USER1
CLD DOCS
CLD ..
```

注: CLD が指定されない場合、デフォルト・ディレクトリーは、¥USERS¥userid¥ になります。

セキュリティ

RLS コマンドは、許可 REXX/CICS コマンドです。

つまり、このコマンドを実行できるのは、許可ユーザーか、領域始動 JCL の CICAUTH または CICEXEC DD 名からロードされる EXEC 内から実行する場合だけです。

RLS コマンド

RLS コマンド環境では、RLS とのインターフェースを取るためにコマンドを発行します。

コマンド環境を RLS に設定する場合、RLS コマンドの前に RLS を指定してはなりません。

例

```
'RLS READ ¥USERS¥USER1¥TEST.DATA DATA.'
```

この例では、RLS リスト ¥USERS¥USER1¥TEST.DATA の内容を DATA. に読み取ります。REXX 複合変数です。

RLS コマンドの構文は、以下のとおりです。

CKDIR

CKDIR は、既存の RLS ディレクトリー・レベルについて調べます。

➡ RLS — CKDIR — *dirid* ➡

オペランド

dirid

REXX List System のディレクトリー・レベル ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CLD コマンドの [367 ページの『CLD』](#) を参照してください。

戻りコード

RLS コマンドの [400 ページの『RLS』](#) を参照してください。

例

```
'RLS CKDIR ¥USERS¥USER1¥DOCS'
```

この例は、既存のディレクトリー ¥USERS¥USER1 内に DOCS というディレクトリーがあるか調べます。

DELETE

DELETE は RLS リストを削除します。

➡ RLS — DELETE — *listname* ➡

オペランド

listname

REXX List System のリスト ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CLD コマンドの [367 ページの『CLD』](#) を参照してください。

戻りコード

RLS コマンドの [400 ページの『RLS』](#) を参照してください。

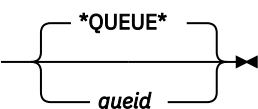
例

```
'RLS DELETE ¥USERS¥USER1¥TEST.DATA'
```

この例は、RLS リスト TEST.DATA を削除します。

LPULL

LPULL は、RLS キューの先頭からレコードをプルします。

➡ RLS — LPULL — *varname* —  ➡

オペランド

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

QUEUE

特殊デフォルト名を指定するキーワードです。

queid

LPULL、LPUSH、または LQUEUE によってアクセスされる特殊タイプの RLS リストの ID を指定します。

戻りコード

RLS コマンドの [400 ページ](#)の『RLS』を参照してください。

例

```
'RLS LPULL VARA QUEUE1'
```

この例は、RLS キューの先頭からレコードをプルします。

LPUSH

LPUSH は、RLS キューの先頭にレコードをプッシュします (LIFO)。



オペランド

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

QUEUE

特殊デフォルト名を指定するキーワードです。

queid

LPULL、LPUSH、または LQUEUE によってアクセスされる特殊タイプの RLS リストの ID を指定します。

戻りコード

RLS コマンドの [400 ページ](#)の『RLS』を参照してください。

例

```
'RLS LPUSH VARA QUEUE1'
```

この例は、レコード (VARA の内容) を RLS キュー QUEUE1 の先頭にプッシュします。

LQUEUE

LQUEUE は、RLS キューの終わりにレコードを追加します (FIFO)。



オペランド

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

QUEUE

特殊デフォルト名を指定するキーワードです。

queid

LPULL、LPUSH、または LQUEUE によってアクセスされる特殊タイプの RLS リストの ID を指定します。

戻りコード

RLS コマンドの [400 ページ](#)の『RLS』を参照してください。

例

```
'RLS LQUEUE VARA QUEUE1'
```

この例は、レコード (VARA の内容) を RLS キュー QUEUE1 の末尾に追加します。

MKDIR

MKDIR は、新しい RLS ディレクトリー・レベルを作成します。

▶ RLS — MKDIR — *dirid* ▶

オペランド

dirid

REXX List System のディレクトリー・レベル ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CLD コマンドの [367 ページ](#)の『CLD』を参照してください。

戻りコード

RLS コマンドの [400 ページ](#)の『RLS』を参照してください。

例

```
'RLS MKDIR ¥USERS¥USER1¥DOCS'
```

この例は、既存のディレクトリー ¥USERS¥USER1 内に DOCS という新しいディレクトリーを作成します。

READ

READ は、RLS リストからレコードを読み取ります。

▶ RLS — READ — *listname* — { *DATA.* / *stem.* } (— UPD) ▶

オペランド

listname

リスト ID を指定します。

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。 [155 ページ](#)の『語幹』を参照してください。デフォルトの語幹は DATA. です。

UPD

更新のためにファイルにエンキューするキーワードです。

戻りコード

RLS コマンドの [400 ページ](#)の『RLS』を参照してください。

例

```
'RLS READ ¥USERS¥USER1¥TEST.DATA DATA.'
```

この例では、RLS リスト ¥USERS¥USER1¥TEST.DATA の内容全体を DATA. に格納します。REXX 複合変数です。

注: DATA.0 は、リストから読み取られたレコードの数に設定されます。DATA.*n* には、リストから読み取られた *n* 番目のレコードが含まれます。

VARDROP

VARDROP は、RLS 保管済み変数を削除します。

➡ RLS — VARDROP — *varname* — *dirid* ➡

オペランド

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

dirid

REXX List System のディレクトリー・レベル ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CLD コマンドの [367 ページの『CLD』](#) を参照してください。

戻りコード

RLS コマンドの [400 ページの『RLS』](#) を参照してください。

例

```
'RLS VARDROP VAR1'
```

この例は、現行ディレクトリーから変数 VAR1 を削除します。

VARGET

VARGET は、RLS 保管済み変数を使用し、それを、同じ名前の REXX 変数にコピーします。

➡ RLS — VARGET — *varname* — *dirid* ➡

オペランド

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

dirid

REXX List System のディレクトリー・レベル ID を指定します。これは、部分修飾または完全修飾です。CLD コマンドの [367 ページの『CLD』](#) を参照してください。

戻りコード

RLS コマンドの [400 ページの『RLS』](#) を参照してください。

例

```
'RLS VARGET VAR1'
```

この例は、現行ディレクトリーからの変数 VAR1 の値を VAR1 という名前の REXX 変数にコピーします。

VARPUT

VARPUT は、REXX 変数を使用し、それを、同じ名前の RLS 保管済み変数にコピーします。

➡ RLS — VARPUT — *varname* — *dirid* ➡

オペランド

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

dirid

REXX List System のディレクトリー・レベル ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CLD コマンドの [367 ページの『CLD』](#) を参照してください。

戻りコード

RLS コマンドの [400 ページの『RLS』](#) を参照してください。

例

```
'RLS VARPUT VAR1'
```

この例は、REXX 変数 VAR1 の値を使用し、それを現行ディレクトリー内の VAR1 にコピーします。

WRITE

WRITE は、レコードを RLS リストに書き込みます。

➡ RLS — WRITE — *listname* —  ➡

オペランド

listname

リスト ID を指定します。

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#) を参照してください。デフォルトの語幹は DATA. です。

戻りコード

RLS コマンドの [400 ページの『RLS』](#) を参照してください。

例

```
'RLS WRITE ¥USERS¥USER1¥TEST.DATA DATA.'
```

この例は、REXX 複合変数 DATA. の内容全体を、RLS リスト ¥USERS¥USER1¥TEST.DATA に格納します。

注: DATA.0 を、リストに書き込まれるレコードの数に設定します。

第 26 章 REXX/CICS コマンド定義

REXX/CICS コマンド定義機能は、REXX コマンドおよび環境を定義または再定義するためのものです。

コマンド定義機能により、以下のことが可能になります。

- REXX EXEC から新規コマンドおよび環境を定義する。
- 複数の独立した開発者と共通コマンド環境を共用する。
- 新規 REXX コマンドを REXX 言語で作成する。
- コマンドの実装言語を透過的に変更する。
- 許可コマンドを選択的に定義する。
- 既存のコマンド名を (コード変更なしで) 再定義する。

背景

REXX の強みの 1 つは、その拡張性にあります。ユーザーは固有の外部関数、サブルーチン、またはコマンドを作成して、REXX 言語の能力を拡張することができます。このため、REXX の使用法の 1 つは、アプリケーション統合プラットフォームとしての使用です。

REXX/CICS は、ユーザーの基幹業務 (LOB) アプリケーション機能、REXX 言語機能、CICS システム機能、および各種ソフトウェア製品を統合ソフトウェア・プラットフォームにシームレスに統合する機能を提供します。

機能を追加したり外部機能や製品へのインターフェースを装備したりするために REXX 言語を拡張する方式としては、主に以下の方式が使用されます。

- REXX 外部関数 (またはサブルーチン)
- REXX コマンド (サブコマンドと呼ばれることがあります)

この 2 つを区別するための例として、REXX 配列をソートする機能を REXX に追加する場合を考えてみましょう。次のように、ARRYSORT というソート関数を追加する方法があります。

```
x = ARRYSORT(STEM1.)
```

また、ARRYSORT という REXX コマンドを追加する方法もあります。

```
'ARRYSORT STEM1.'
```

注: コマンドの例では、コマンドを引用符で囲む必要はありません。しかし、REXX 変数置換が必要でないコマンド・ストリングの部分は単一引用符または二重引用符で囲むことが、コーディングについてのグッド・プラクティスです。

REXX 外部関数は、従来、REXX 言語の機能を拡張するために使用されますが、REXX コマンドは、従来、外部のアプリケーション、製品、またはシステムの機能へのインターフェースを提供するために使用されます。しかし、どちらの方式も同じように使用できます。

REXX コマンドと外部関数との主な相違点は、次のとおりです。

- 関数は、必ず、結果を戻すが、コマンドはその限りではない。
- 関数ではエラー条件が発生するが、コマンドでは、必ず、戻りコードが設定される。

コマンドの定義

DEFCMD コマンドを使用して、自分のユーザー ID にのみ影響する基本コマンド定義を実行できます。システム管理者またはシステム・プログラマーは、DEFSCMD コマンドを使用して、システム全体の REXX/CICS コマンド定義を実行できます。

REXX コマンド定義の定義または変更を行うためには、DEFCMD コマンドおよび DEFSCMD コマンドを REXX EXEC 内から使用します。DEFCMD コマンドを使用すると、特別な許可なしで、ユーザー固有のコマ

ンド定義を追加または変更できます。DEFSCMD を使用して他の REXX/CICS ユーザーに影響するコマンド定義を変更するためには、REXX/CICS 許可ユーザーでなければなりません。詳しくは、[374 ページの『DEFSCMD』](#) および [376 ページの『DEFSCMD』](#) を参照してください。

REXX プログラムに渡されるコマンド引数

REXX/CICS コマンドが REXX で作成され、そのコマンドが使用されると、REXX プログラム (DEFSCMD または DEFSCMD で定義されたもの) が呼び出されるか、または (WAITREQ に誘発された「スリープ」から) 目覚めます。

このプログラムが呼び出されると、そのコマンド・ストリングは EXEC に引数として渡されます。また、呼び出された場合、最初に発行された WAITREQ コマンド (存在する場合) は、コマンド・ストリングが REXX 変数 REQUEST 内に置かれた状態で、即座に失敗に終わります。REXX EXEC が、前もって既に開始され、(前の WAITREQ コマンドのために) 要求を待機していると、コマンド・ストリングのみが REXX 変数 REQUEST に置かれます。

注: REXX で作成されたコマンド・プログラムは、C2S コマンドおよび S2C コマンドを使用して、それらが呼び出される原因となった REXX 変数の内容を取得し、REXX EXEC に設定することができます。詳しくは、[373 ページの『C2S』](#) および [407 ページの『S2C』](#) を参照してください。

アセンブラー・プログラムに渡されるコマンド引数

REXX/CICS コマンド・プログラムはアセンブラー言語で作成できます。

アセンブラー言語ルーチンが、(例えば、CEDA DEFINE PROGRAM コマンドを使用して) 適切に定義された CICS プログラム内に存在している必要があります。これらのプログラムは、DEFSCMD コマンドまたは DEFSCMD コマンドで CICSLINK オプションが指定されていた場合に、EXEC CICS LINK によって呼び出されます。

一方、DEFSCMD または DEFSCMD コマンドで CICSLOAD オプションが指定されていた場合は、現行 CICS タスクでプログラムを呼び出した最初のコマンドによって、プログラムの EXEC CICS LOAD が実行され、そのロード・アドレスが記憶されます。このプログラムを使用する同じ CICS タスク内の後続のコマンドはすべて、(アセンブラー BASSM 命令により) 直接分岐エントリーを実行して、プログラムに入れます。正しいモード切り替え (該当する場合) が行われるように、アセンブラー BSM 命令を使用して、これらのアセンブラー・プログラムから制御を戻すことをお勧めします。

以下の情報で、アセンブラー言語コマンド・プログラムが制御を取得する場合のレジスターの内容と、これらのプログラムに入るときのパラメーターについて説明します。

DEFSCMD CICSLOAD が指定された場合のエントリー仕様:

コマンド・プログラムのコードが直接分岐によって制御を取得した場合、レジスターの内容は、以下のとおりです。

レジスター 0

予測不能

レジスター 1

CICPARMS 制御ブロックのアドレス

レジスター 2 から 12

予測不能

レジスター 13

18 フルワード・レジスター保管域のアドレス

レジスター 14

リターン・アドレス

レジスター 15

入り口点のアドレス

プログラムは、呼び出し元に戻る前に、反映させたい戻りコードを CICPARMS RETCODE フィールドに入れる必要があります。

DEF CMD CICS LINK が指定された場合のエントリー仕様:

コマンド・プログラムのコードが EXEC CICS LINK によって制御を取得した場合、CICS 通信領域に CICPARMS 制御ブロックが含まれています。

プログラムは、呼び出し元に戻る前に、反映させたい戻りコードを CICPARMS RETCODE フィールドに入れる必要があります。

CICPARMS 制御ブロック

このトピックには、プロダクト・センシティブ・プログラミング・インターフェース 情報が含まれています。

以下の表は、渡されたパラメーターをアセンブラー・ルーチンにマップするための CICPARMS 制御ブロックを示しています。

表 4. CICPARMS 制御ブロック			
オフセット (10 進数)	バイト数	フィールド名	説明
0	12		IBM が使用するために予約済みです。
12	4	RXWBADDR	(REXX 変数アクセスのための) CICGETV スタブ・ルーチンへの呼び出し前にレジスター 10 に入っている必要がある REXX 作業ブロック・アドレス
16	8	ENVNAME	DEF CMD コマンド定義または DEF SCMD コマンド定義から取得される内部環境名
24	16	CICCMD	コマンド定義から取られた内部コマンド名、またはアスタリスクが指定されている場合は、コマンド・ストリングからの実際のコマンド名
40	4	ARGSTR	コマンド・ストリング内のコマンド名の後の最初の非空白文字で始まる、コマンド引数ストリングのアドレス
44	4	ARGLEN	上記引数ストリングの長さ (文字数)
48	4	PLIST	8 文字のトークンに解析されたコマンド行の標準構文解析パラメーター・リストのアドレス。後ろに 16 進高位値のリストの終わり境界 (X'FFFFFFFFFFFFFFFF') が続きます。
52	4	EPLIST	前述の標準 PLIST に調和するが、形式が異なる拡張パラメーター・リストのアドレス。拡張 PLIST にはトークンごとに 8 バイトのエントリーがあります。最初の 4 バイトは、トークンを構成するストリングの始まりのフルワード・アドレスです。2 番目のワードには、バイト単位でのトークンの長さが入っています。
56	4	RETCODE	コマンドの実行直後に EXEC で反映させる戻りコード。この戻りコードは、特殊 REXX 変数 RC に自動的に入れられます。
60	4		IBM で使用するために予約済み
64	4	USERWORD	ユーザーが使用できるように、複数のコマンド・ルーチン呼び出しにわたって情報を渡すことができます。
68	4		IBM で使用するために予約済み

表 4. CICPARMS 制御ブロック (続き)			
オフセット (10 進数)	バイト数	フィールド名	説明
72	4		IBM で使用するために予約済み
76	1	TYPEFLAG	DEFECMD 定義または DEFSCMD 定義の呼び出しタイプを識別する 1 文字のコード。REXX のコードは R、CICSLINK のコードは C、CICSLOAD のコードは L です。
77	1	ITRACE	内部トレース・フラグ。これは、内部トレースがアクティブであるかどうか、およびどのレベルのトレースがアクティブになっているかを示す、0 から 9 までの値を持つ 1 文字コードです。値ゼロは、トレースなしという通常の状態を示します。1 から 9 の値は、値に応じて、要求されるトレースの詳細さが次第に増大することを示します。

REXX 以外の言語インターフェース

REXX/CICS では、REXX プロセスを非 REXX プロセスに変換することができます。

これを行うには、非 REXX コマンド・ルーチンが、処理されるコマンドを発行した REXX EXEC 内の REXX 変数にアクセスできることが必要です。これを遂行するのに使用されるルーチンは、CICGETV といい、ユーザーのコマンド・ルーチンでリンク・エディットし、以下の情報に示すように呼び出す必要があります。

CICGETV: REXX 変数を取得、設定、またはドロップするための呼び出し

CICGETV は、この linkedit スタブ・サブルーチンをアセンブラー REXX/CICS コマンド・ルーチンから呼び出します。これは、REXX 変数を、コマンドを発行した REXX プログラムから取り出したり、設定したり、ドロップしたりするための呼び出しです。

このトピックには、プロダクト・センシティブ・プログラミング・インターフェース 情報が含まれています。

➡ CALL — CICGETV — , — (— **オペランド**) — ➡

オペランド

➡ *varname_addr* — , *varname_len* — , *data_addr* — , *data_len* — , *function_name* — ➡

オペランド

varname_addr

REXX 変数の名前を含む文字ストリングのアドレスを指定します。変数名 (このアドレスが指す先) は、大文字でなければなりません。

varname_len

変数名の長さを指定します。

data_addr

変数の内容のアドレスが存在するフルワード・アドレスを指定します。

data_len

data_addr (フルワード) が指す領域の長さを指定します。

function_name

実行する特定の CICGETV 関数を指定します。選択項目は、以下の 3 つです。

GET

REXX 変数のアドレスと長さを取得します。

PUT

REXX 変数を作成または置換します。

DEL

REXX 変数を削除します。

注：

1. CICGETV linkedited ルーチン・スタブを呼び出す直前に、渡される parms (CICPARMS) 内の RXWBADDR フィールドの値を使用してレジスター 10 をロードする必要があります。
2. 存在しない変数に対して get 要求が発行された場合、返される値は変数名と同じになります。
3. CICGETV では、18 フルワード長域を指す R13 を持つ標準保管域規則を使用します。R1 は、標準パラメーター・リストを指します。R14 には戻りアドレスが入っています。R15 は、CICGETV に入るときに、CICGETV のエントリー・ポイントを戻します。戻り際には、R15 に戻りコードが入っています。
4. CICGETV モジュールは、REXX/CICS 配布ライブラリー内にあります。
5. その命令の成功を示すためにゼロの戻りコードが反映されるか、または内部エラーを示す 10 進数の 99 (ストレージ制限を超える状況など) です。

第 27 章 REXX/CICS Db2 インターフェース

REXX/CICS Db2 インターフェースには、REXX EXEC から SQL ステートメントおよび Db2 コマンドを実行するための手段が用意されています。

SQL は、動的に準備されて実行されます。Db2 コマンドを発行するには、Db2 計測機能インターフェース (IFI) を使用します。REXX/CICS Db2 インターフェースは、SQL の結果を REXX 事前定義変数で提供します。REXX/CICS Db2 インターフェースは、DB2[®] バージョン 2.3 以降をサポートしています。この情報では、REXX/CICS からの Db2 へのインターフェースの使用方法を説明します。

SQL ステートメントまたは Db2 コマンドについて詳しくは、[Db2 for z/OS 製品資料内の『SQL: Db2 の言語』](#) および [Db2 コマンドを参照してください](#)。IFI の詳細については、[z/OS 製品資料内の『計測機能インターフェース \(IFI\) のプログラミング』](#)を参照してください。

注: Db2 リソース管理テーブル (RCT) に REXX/CICS トランザクション ID を入力する必要があります。サンプル JCL CICRCT を参照してください。

プログラミング上の考慮事項

SQL を REXX EXEC 内に組み込むには、ホスト・コマンド環境を変更する必要があります。ADDRESS 命令は、後ろに環境の名前が続いており、ホスト・コマンド環境を変更するのに使用されます。

ADDRESS 命令には 2 つの形式があり、一方は、その命令の後に出力されたすべてのコマンドに作用し、もう一方は単一のコマンドにのみ作用します。ホスト・コマンド環境について詳しくは、[93 ページの『ホスト・コマンド環境の変更』](#)を参照し、ADDRESS 命令について詳しくは、[163 ページの『ADDRESS』](#)を参照してください。

REXX/CICS Db2 インターフェースをサポートする REXX/CICS コマンド環境は以下のとおりです。

EXECDB2

Db2 コマンドをサポートするコマンド環境。

EXECSQL

SQL ステートメントをサポートするコマンド環境。

注: EXECSQL と EXECDB2 は許可コマンドです。EXECSQL および EXECDB2 コマンド環境を使用できるのは、REXX/CICS 許可ユーザーのみです。

REXX/CICS には、REXX ステートメントとコマンドを対話式に処理するために使用できる CICRXTRY という EXEC が用意されています。CICRXTRY は疑似会話型にすることができます。疑似会話型モードのオン/オフを切り替えるには、PSEUDO コマンドおよび SETSYS PSEUDO コマンドを使用します。環境が疑似会話型に設定されている場合、CICRXTRY から発行された SQL ステートメントはコミットされます。環境が会話型に設定されている場合、EXEC から発行された SQL ステートメントはいずれもコミットされず、ロックされたリソースはすべて、ユーザーが CICRXTRY EXEC を終了するか、CICS SYNCPOINT コマンドを発行するまで、ロックされたままになります。長い REXX EXEC に SQL ステートメントを組み込む場合にも、同様の考慮が必要です。

SQL ステートメントの組み込み

SQL を処理するためには、EXECSQL コマンド環境を使用します。各 SQL ステートメントは、CICS Db2 接続機能を使用して動的に準備されて、実行されます。

要求ごとに、EXECSQL 環境に向けられた REXX コマンドとして有効な SQL ステートメントを作成する必要があります。SQL ステートメントは、以下のエレメントで構成されます。

- SQL キーワード
- 事前宣言 ID
- リテラル値。

以下の構文を使用します。

```
"EXECSQL statement"
```

または

```
ADDRESS EXECSQL  
"statement"  
"statement"  
.  
.  
.
```

SQL は、複数行にわたってかまいません。ステートメントの各部分を、以下のように、引用符で囲み、追加のステートメント・テキストを 1 つのコンマで区切ります。

```
ADDRESS EXECSQL  
"SQL text",  
"additional text",  
.  
.  
.  
"final text"
```

組み込み SQL には、以下の規則が適用されます。

- 以下の SQL は、EXECSQL コマンド環境に直接渡すことができます。

```
ALTER  
CREATE  
COMMENT ON  
DELETE  
DROP  
EXPLAIN  
GRANT  
INSERT  
LABEL ON  
LOCK  
REVOKE  
SELECT  
SET CURRENT SQLID  
UPDATE
```

- 以下の SQL ステートメントは使用できません。

```
BEGIN DECLARE SECTION  
CLOSE  
COMMIT  
CONNECT  
DECLARE CURSOR  
DECLARE STATEMENT  
DECLARE TABLE  
DESCRIBE  
END DECLARE SECTION  
EXECUTE  
EXECUTE IMMEDIATE  
FETCH  
INCLUDE  
OPEN  
PREPARE  
ROLLBACK  
SET CURRENT PACKAGESET
```


SET HOST VARIABLE
WHENEVER

- ホスト変数は、SQL 内では許可されません。代わりに、REXX 変数を使用して、入力データを EXECSQL 環境に渡すことができます。REXX 変数は引用符で囲みません。EXECSQL 環境からの出力は、REXX 事前定義変数で提供されます (315 ページの『結果の受信』を参照)。
- SQL SELECT ステートメントをコーディングする場合、INTO 節を使用することはできません。代わりに、REXX/CICS Db2 インターフェースは、要求された項目を、Db2 列名と同じ語幹名を持つ複合変数に返します。
- SELECT ステートメントについて戻される行数のデフォルト数は 250 です。必要な行がもっと多い、あるいはもっと少ない場合は、SELECT ステートメントを発行する前に REXX 変数 SQL_SELECT_MAX を設定することができます。

結果の受信

EXECSQL コマンド環境は、事前定義された REXX 変数に結果を返します。

この変数は、以下のものです。

RC

操作ごとに、この戻りコードを設定します。可能な値は次のとおりです。

n

SQL ステートメントの結果がエラーまたは警告である場合は SQLCODE。

0

SQL ステートメントは、EXECSQL 環境によって処理されました。SQLCA の REXX 変数には、SQL ステートメントの完了状況が含まれています。

30

SQLDSECT 変数を作成するための十分なメモリーがありませんでした。

31

SQL ステートメント域を作成するための十分なメモリーがありませんでした。

32

SQLDA 変数を作成するための十分なメモリーがありませんでした。

33

SELECT ステートメント用の結果領域を作成するための十分なメモリーがありませんでした。

SQLCA

SQL ステートメントが処理された後で、一連の SQLCA 変数が更新されます。SQLCA のエントリーについては、316 ページの『SQL 連絡域の使用』で説明しています。

SQL_COLUMN.n

データが SELECT ステートメントによって戻された各 Db2 列の名前が入ります。SQL_COLUMNS を *n* の最大値として使用する必要があります。

SQL_COLTYPE.n

データが SELECT ステートメントによって戻された各 Db2 列のタイプが入ります。SQL_COLUMNS を *n* の最大値として使用する必要があります。

注：すべてのデータ・タイプがサポートされますが、すべてが表示できるわけではありません。REXX 関数を使用すると、希望の形式にデータを変換することができます。

SQL_COLTYPE にある特定の SQLTYPE コードの意味については、Db2 for z/OS 製品資料内の『SQL: Db2 の言語』を参照してください。

SQL_COLLEN.n

データが SELECT ステートメントによって戻された各 Db2 列の長さが入ります。データ・タイプが DECIMAL の場合、位取りは列の長さの後 (1 つのブランク・スペースの後) に置かれます。

SQL_COLUMNS を *n* の最大値として使用する必要があります。

SQL_COLUMNS

戻される列の数のカウントが入ります。

column.n

SQL SELECT ステートメントの結果は、これらの REXX 複合変数に格納されます。*column* は Db2 列の名前です。各項目には、Db2 からの 1 行のデータが含まれます。戻される SQL 行の数のカウントは *column.0* に入ります。このカウントを *n* の最大値として使用する必要があります。

SQLCOLn.1

CURRENT SQLID、MAX、および AVG など、一部の SELECT 関数は、特定の Db2 列に関連付けられていません。結果を確認するには、列名 **SQLCOLn.1** を参照する必要があります。

n は、1 から始まり、SELECT ステートメントに含まれる関数ごとに 1 ずつ増えます。SQLCOL*n* によって表されるすべての列が SQL_COLNAME 複合変数に示されます。

SQL 連絡域の使用

SQL 連絡域 (SQLCA) を構成するフィールドは、SQL ステートメントが発行されたときに、REXX/CICS Db2 インターフェースによって自動的に組み込まれます。

SQLCA の SQLCODE フィールドおよび SQLSTATE フィールドには、SQL 戻りコードが入っています。これらの値は、各 SQL ステートメントが実行された後で、REXX/CICS Db2 インターフェースによって設定されます。

SQLCA フィールドは、連続データ域ではなく、個別の変数に維持されます。維持される変数は、以下のよう定義されます。

SQLCODE

1 次 SQL 戻りコード。

SQLERRM

エラー・メッセージおよび警告メッセージのトークン。隣接するトークンは、X'FF' が含まれているバイトで区切られます。

SQLERRP

製品コード。エラーがあった場合は、そのエラーを返したモジュールの名前。

SQLERRD.n

診断情報が含まれている 6 個の変数。(変数 *n* は、1 から 6 までの数値です。)

注: コマンド DELETE、INSERT、および UPDATE が作用する SQL 行の数のカウントは、SQLERRD.3 に入ります。

SQLWARN.n

警告フラグが含まれている 11 個の変数。(変数 *n* は、0 から 10 までの数値です。)

SQLSTATE

代替 SQL 戻りコード。

SQL ステートメントの使用例

この例では、プログラムは、部門の名前の入力を求めるプロンプトを出し、EMPLOYEE テーブルからその部門のすべてのメンバーの名前と電話番号を取得すると、その情報を画面に示します。

```
/******  
/* Exec to list names and phone numbers by department */  
/******  
  
/*-----*/  
/* Get the department number to be used in the select statement */  
/*-----*/  
  Say 'Enter a department number'  
  Pull dept  
  
/*-----*/  
/* Retrieve all rows from the EMPLOYEE table for the department */  
/*-----*/  
  "EXECSQL SELECT LASTNAME, PHONENO FROM EMPLOYEE ",  
    "WHERE WORKDEPT = '"dept"'"  
  If rc <> 0 then  
    do  
      Say ' '  
      Say 'Error accessing EMPLOYEE table'  
      Say 'RC      =' rc
```

```

        Say 'SQLCODE =' SQLCODE
        Exit rc
    end

/*-----*/
/* Display the members of the department */
/*-----*/
    Say 'Here are the members of Department' dept
    Do n = 1 to lastname.0
        Say lastname.n phoneno.n
    End

Exit

```

Db2 コマンドの埋め込み

Db2 コマンドを処理するためには、EXECDB2 コマンド環境を使用します。各 Db2 コマンドは、Db2 計測機能インターフェース (IFI) に渡されます。

TSO で DSN を使用して発行される Db2 コマンドのほとんどは、REXX/CICS Db2 インターフェースを使用して CICS から発行できます。Db2 メッセージは、事前定義された REXX 複合変数 DB2_OUTPUT.n に戻されます。

有効な Db2 コマンドを、EXECDB2 環境に送られる REXX コマンドとして指定することで、各要求を実行できます。以下の構文を使用します。

```
ADDRESS EXECDB2 "DB2 command"
```

または

```

ADDRESS EXECDB2
"DB2 command"
"DB2 command"
.
.
.

```

Db2 コマンドは、複数行にわたってかまいません。次のように、コマンドの各部分を単一引用符で囲み、コンマで区切ってコマンド・テキストを追加できます。

```

ADDRESS EXECDB2
"DB2 command",
"additional text",
.
.
.
"final text"

```

組み込み Db2 コマンドには、以下の規則が適用されます。

- REXX EXEC はさまざまな Db2 コマンドを発行できますが、使用するコマンドには注意してください。例えば、"-STOP DB2" は使用しないでください。
- 以下の Db2 コマンドは、EXECDB2 コマンド環境に直接渡されます。

```

-ALTER BUFFERPOOL
-ARCHIVE LOG
-CANCEL DDF THREAD
-DISPLAY ARCHIVE
-DISPLAY BUFFERPOOL
-DISPLAY DATABASE
-DISPLAY LOCATION
-DISPLAY RLIMIT
-DISPLAY THREAD
-DISPLAY TRACE
-DISPLAY UTILITY
-MODIFY TRACE

```

- RECOVER BSDS
 - RECOVER INDOUBT
 - RESET INDOUBT
 - SET ARCHIVE
 - START DATABASE
 - START DDF
 - START RLIMIT
 - START TRACE
 - STOP DATABASE
 - STOP DB2
 - STOP DDF
 - STOP RLIMIT
 - STOP TRACE
 - TERM UTILITY
- 以下の Db2 コマンドは使用できません。
 - START DB2。このコマンドは、MVS コンソールからのみ発行できます。
 - EXECDB2 環境に入力データを渡すときには REXX 変数を使用できます。REXX 変数は引用符で囲みません。EXECDB2 環境からの出力は、REXX 事前定義変数で提供されます ([318 ページの『結果の受信』](#)を参照)。
 - Db2 コマンド・ストリングの長さは、6 文字から 4092 文字までの範囲になければなりません。
 - Db2 IFI から戻されるすべてのメッセージの長さの合計は、24K に制限されています。メッセージが多くて 24K バイトのメモリーに収まらない場合は、出力メッセージの数が減るように、より限定的なパラメーター値を指定して、Db2 コマンドを再発行できます。

結果の受信

EXECDB2 コマンド環境は、事前定義された REXX 変数に結果を返します。

この変数は、以下のものです。

RC

操作ごとに、この戻りコードを設定します。可能な値は次のとおりです。

n

Db2 計測機能インターフェース (IFI) の呼び出しの結果を表す正の値。Db2 IFI からの RC がゼロでない場合、REXX 変数 DB2_RC2 には、Db2 IFI 理由コードが含まれます。DB2_RC2 値をこの RC と一緒に使用して、Db2 コードでエラー判別を行えます。[z/OS 製品資料内の『Db2 の Db2 コード』](#)を参照してください。

0

Db2 コマンドは Db2 IFI によって処理されました。

50

指定された Db2 コマンドは、短すぎるか長すぎるために Db2 IFI で処理できません。Db2 コマンドの長さは、6 文字以上かつ 4092 文字以下にする必要があります。

51

Db2 IFI 用の出力域を作成するための十分なメモリーがありませんでした。

52

Db2 IFI 用の通信域を作成するための十分なメモリーがありませんでした。

53

Db2 IFI 用の戻り域を作成するための十分なメモリーがありませんでした。

54

DB2_RC2 REXX 変数を作成できません。

55

DB2_BNM REXX 変数を作成できません。

56

DB2_OUTPUT.*n* REXX 変数を作成できません。

57

DB2_OUTPUT.0 REXX 変数を作成できません。

DB2_OUTPUT.*n*

Db2 IFI から戻された Db2 メッセージは、この REXX 複合変数に格納されます。DB2_OUTPUT.*n* 項目ごとに Db2 メッセージが 1 つ含まれます。DB2_OUTPUT.0 には、戻された Db2 メッセージの数のカウントが含まれます。このカウントは、*n* の最大値として使用されます。

DB2_RC2

Db2 IFI からの戻りコードがゼロでない場合、この変数には、戻りコードに関連する理由コードが入ります。DB2_RC2 変数には 16 進値が入ります。印刷可能な値に変換するには、組み込み REXX 関数の C2X を使用してください。202 ページの『C2X (文字から 16 進数へ)』を参照してください。DB2_RC2 値をこの RC と一緒に使用して、Db2 コードでエラー判別を行えます。z/OS 製品資料内の『Db2 の Db2 コード』を参照してください。

DB2_BNM

Db2 IFI がすべての Db2 メッセージを戻すことができない場合に、移動されていないバイト数が DB2_BNM に入ります。戻されるすべてのメッセージの合計長が 24 K を超える場合に、この状態になることがあります。出力を減らすためにさらにパラメーター値を指定して、Db2 コマンドを再発行してください。

Db2 コマンドの使用例

データベース内のすべての表スペースが RW 状況にあるかどうかを調べる REXX コードの例。

```

/*****
/* Exec to verify that tablespaces are in RW status */
/*****

db_name = 'DSN8D23A'
cmd = "-DISPLAY DATABASE("db_name") LIMIT(500)"
"EXECDB2" cmd
If rc <> 0 then
do
  Say ' '
  Say 'Error in DB2 -DISPLAY DATABASE command'
  Say 'RC      =' rc
  Say 'DB2_RC2 =' C2X(DB2_RC2)
  Exit rc
end

/*-----*/
/* Scan the DB2 messages, skipping over the "header" */
/* and "trailer" messages.                               */
/*-----*/
first = 10
last  = DB2_OUTPUT.0 - 2
Do n = first to last
  If substr(DB2_OUTPUT.n, 20, 2) <> "RW" then
    Say "Tablespace" substr(DB2_OUTPUT.n, 1, 8) "is not in RW status"
End

Exit

```


第 28 章 REXX/CICS ハイレベル・クライアント/サーバー・サポート

REXX/CICS では、REXX 言語クライアント/サーバー・サポートが導入されました。REXX/CICS は、ハイレベルのクライアント/サーバー機能を提供します。

この機能には、次のものが含まれます。

- ・ハイレベルで、最適な透過的 REXX クライアント・インターフェース
- ・REXX ベース・アプリケーションのクライアントおよびサーバーのサポート

クライアント/サーバー・コンピューティングの利点をいくつか以下に示します。

- ・メインフレーム、ミニコンピューター、およびワークステーションの強みを、透過的な形で効果的に統合できる。
- ・コンピューターのシステム・サイズを、徐々にスケールアップしたりスケールダウンしたり (ライトサイジング) できる。
- ・複雑なアプリケーション・システムを管理しやすいクライアント/サーバーのセットに分割することで、単純化できる。
- ・アプリケーションとデータをネットワーク全体に分散することで、より優れたパフォーマンス、整合性、セキュリティを実現できる。
- ・多種多様の異なるコンピューター・システムやワークステーションから、ネットワーク内での配置場所に関係なく、エンタープライズ全体がデータやアプリケーションにアクセスできる。

ハイレベルで、最適な透過的 REXX クライアント・インターフェース

REXX/CICS は、REXX の ADDRESS キーワード命令を介したクライアント REXX EXEC からアプリケーション・サーバーへのハイレベルな使いやすいインターフェースをサポートします。

ADDRESS 命令には、後続の REXX コマンド・ストリングを処理するために呼び出される外部プロシージャの名前の判別に使用される外部環境名が含まれています。

REXX/CICS は、オプションで、環境名 (ADDRESS 命令で指定) をアプリケーション・サーバーの名前にすることができるようになります。この機能は、REXX/CICS DEFCMD コマンドおよび DEFSCMD コマンドによって提供されます。

DEFCMD コマンドでは、REXX のコマンドおよび環境を定義 (または再定義) することができ、環境とコマンドの組み合わせを従来の CALL されるルーチンまたは REXX アプリケーション・サーバーのどちらで処理するか指定することができます。

REXX ベース・アプリケーションのクライアントおよびサーバーのサポート

REXX クライアント・インターフェースに加えて、いくつかの機能により、REXX で作成されたアプリケーション・サーバーの使用のサポートが提供されます。

1 つの機能は WAITREQ コマンドであり、これは、サーバーがクライアントからの要求を待つために使用します。それとは別に、C2S コマンドおよび S2C コマンドは、サーバーが、クライアント変数の内容を取り出したり、設定したりできるようにします。また、自動サーバー始動 (ASI) により、サーバーは、要求がクライアントから到着すると自動的に始動されます。

クライアント/サーバー・コンピューティングでの REXX の値

クライアント/サーバー・ソリューションを実装するために REXX を使用するメリットの一部をリストします。

- ・REXX/CICS で、その高速な開発サイクルとソース・ベースの対話式デバッグによる REXX インタープリター・サポートが使用可能であると、複雑なシステムの短時間でのプロトタイプングおよび開発が可能になる。

- REXX/CICS でのハイレベルのクライアント/サーバー・インターフェースにより、開発の生産性が向上し、維持コストが低減できる。
- REXX/CICS では REXX クライアントおよびサーバーを REXX 以外の言語で再コーディングすることができるため、アプリケーション・システムのパフォーマンス 主体の部分を、必要に応じて、選択的には再作成できる。

REXX/CICS が提供する FLST コマンドおよび EDIT コマンドは、クライアント/サーバー環境の例です。

REXX/CICS クライアント EXEC の例

```
/* EXAMPLE REXX/CICS EXEC */

TRACE '0' /* turn off source tracing */

ARG parm1 parm2 parm3

"CICS READQ TS QUEUE(MYQ) INTO(DATA) ITEM(5) NUMITEMS(1)"
if rc ~= 0 then EXIT 100

SAY 'TSQ Data=' data
"CICS SEND TEXT FROM(DATA) ERASE"

/* Define the SERVER EXEC as a REXX/CICS command */
'DEFCMD REXXCICS SERVER = SERVER1 (REXX'

/* example of directing a subcommand to a server */
/* named SERVER1, which is written in REXX also */
DATA = 1
'SERVER COMMAND1 DATA'
say data /* ==> 2 */
if rc ~= 0 then SAY 'Request to SERVER1 failed, RC=' rc
EXIT
```

REXX/CICS サーバー EXEC の例

```
/* EXAMPLE REXX/CICS SERVER1 EXEC */

TRACE '0' /* turn off source tracing */

/*-----*/
/* Loop waiting on requests from clients */
/*-----*/
Do Forever
  'WAITREQ'
  parse var request cmd varname
  Select
    When request = 'COMMAND1' then CALL command1
    When request = 'COMMAND2' then CALL command2
    When request = 'STOP'      then CALL stop_server
    Otherwise
  End /* Select */
End /* Do Forever */
exit

/* subroutine to process command1 */
Command1:
'C2S' varname 'WORK'
WORK = WORK + 1
'S2C WORK' varname
return

/* subroutine to process command2 */
Command2:
return

/* routine to shut down this server */
stop_server:
say 'The Server is stopping'
出口
```


第 29 章 REXX/CICS パネル機能

REXX パネル機能は、パネル定義や、3270 タイプの端末へのパネル入出力用の単純なツールとコマンドを、REXX プログラマーに提供します。

パネル機能は、任意のエディターを使用してパネルを簡単に定義できます。パネル・ソース定義ファイルが REXX ファイル・システム (RFS) 内に入っていないと、その処理をさらに進めることはできません。パネル入出力コマンドは、パネル定義機能によって静的に定義されたフィールド属性の多くを REXX プログラム内で動的に変更できるようにします。以下は、パネル機能で何ができるかを示す例です。これは、ここで説明する一般的な概念を理解して視覚化するのに役立ちます。この例の概要は以下のとおりです。

- パネル・フィールドの特性を設定するフィールド制御文字を定義します。
- パネル・レイアウトを定義するフィールド制御文字を使用します。制御文字定義およびパネル定義 (この 2 つの部分をもとめて、パネル・ソースといいます) は REXX ファイル・システムに保管されます。
- パネル・ソースを使用して、REXX プログラムでパネルの送受信に使用するパネル・オブジェクト を生成します。

パネル定義の例

```
** SAMPLE PANEL DEFINITION.

define the field control characters to be used in the panel layout.
.DEFINE < blue protect
.DEFINE @ blue skip
.DEFINE ! red protect
.DEFINE > green unprotect underline
.DEFINE # green unprotect numeric right underline

define a panel named applican, which
queries an applicant's name and address.
(this line and the above lines are treated as comments).

.PANEL applican

< Please type the requested information below @

                                !Applicant's name
@Last name ...:>&lname                @
@First name ...:>&fname                @
@MI.....:>1&mi

                                !Applicant's mailing address

@Street.....:>&mail_street                @
@City.....:>&mail_city                    @
@State.....:>2&mail_state
@Zip...:#5&mail_zip

.PANEL

** END OF SAMPLE PANEL DEFINITION.

** START OF REXX PROGRAM USING THE PREVIOUS PANEL.

/* program to query applicant's name and address */
lname = ''; /* null out all name parts */
fname = '';
mi = '';
mail_street = '';
mail_city = 'DALLAS'; /* prefill the most likely response for city/state */
mail_state = 'TX';
mail_zip = '';
do forever;
  'panel send applican cursor(lname)';
  if rc > 0 then
    call error_routine;

  'panel receive applican'; /* pseudo-conversational this would be separate */
  if pan.aid = 'PF3' | pan.aid = 'PF12' then
```

```

        leave;
    if pan.aid = 'ENTER' & pan.rea = 124 then
        iterate;
    if pan.aid = 'CLEAR' | substr(pan.aid,1,2) = 'PA' then
        iterate; /* go to beginning of loop */
    if rc > 0 then
        call error_routine;

    /* process the name and address */

end;
'panel end';
exit

error_routine:
    Say 'An error has occurred'
return;

** END OF REXX PROGRAM and end of sample.

```

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料で使用されている表記規則および用語](#)も参照してください。

パネルの定義

パネルを定義するには、2つの手順が必要です。

1. エディターを使用して、REXX ファイル・システム内にパネル・ソース・ファイルを作成します。パネル・ソースには、フィールド制御文字とパネル・レイアウトを含める必要があります。パネル・レイアウトには、フィールド制御文字、通常の表示可能テキスト文字、および (場合によっては) 埋め込み変数名を含めることができます。
2. パネル・ソースを、パネル入出力コマンドが使用できる中間形式 (パネル・オブジェクト) に変換します。パネル機能は、そのパネルを参照する入出力コマンドが初めて呼び出されたとき、あるいは中間ファイルを生成するために REXX 環境 (またはそのコマンドが含まれている REXX EXEC) 内の明示的コマンドを呼び出すことができるときに、自動的に中間ファイルを生成します。

注: この自動生成は、パネル・オブジェクトが見つからない場合や、パネル・ソースにパネル・オブジェクトより後の日付または時刻の変更がある場合に実行されます。パネル・ソースに対する変更があると、新しいパネル・オブジェクトが作成されます。したがって、パネルを使用するプログラムのテスト・フェーズが終了した後に、そのパネルのソースを変更する場合は注意が必要です。プロジェクトの実動を開始したら、そのパネル・ソースを RFS から出すか、またはプログラムからアクセスできない RFS ディレクトリーに入れることをお勧めします。

.DEFINE verb を使用したフィールド制御文字の定義

このフィールド制御文字は、パネル上のフィールドの属性を定義します。

これらの制御文字は、.DEFINE verb を使用して定義可能です。パネル・レイアウトの前に置く必要があります。.DEFINE verb は、継続文字 (コンマ) がその行の最後の文字でない限り、'End of line' で終了します。継続文字を .DEFINE verb の直後に続けることはできません。そうすると、コンマが、定義される制御文字であるのか、それとも継続文字であるのかがあいまいになるためです。

スペースが各キーワードを区切るため、定義される制御文字をこの verb の直後に置くこと以外、順序は重要ではありません。

テキストはいずれも、列 1 の .DEFINE で始まらない場合、その行が継続でない限り、無視され、コメントとして扱われます。

合計 32 個の制御文字 (デフォルト制御文字を含む) を一度にアクティブに定義できます。

DROP キーワードを使用してすべての文字を削除すると、5 つのデフォルト制御文字が再活動化されます。

矛盾するために許可されないキーワードの組み合わせがいくつかあります。意味がないように思えるのに許可される組み合わせもあります。例えば、INVISIBLE と color の組み合わせなどです。この組み合わせは、フィールド属性が REXX プログラム内で動的に変更される場合に便利です (不可視のフィールドを可視にして、色に意味を与えることができます)。

.DEFINE verb は、次の特性を持っています。

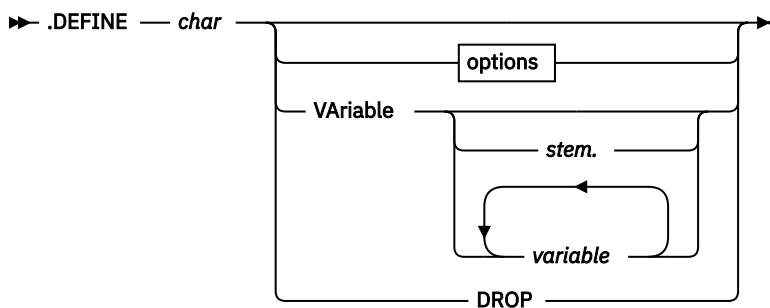
- 最初の列で始まり、後ろに、スペースを 1 つ続け、大文字でなければなりません。
- 継続文字 (コンマ) が使用されている場合を除き、'End of line' で終わります。
- すべてのキーワードは、最小 2 文字の省略形を持ちます。
- 一度に定義できる制御文字の最大数は 32 です。(デフォルトの制御文字は、このカウントに含まれます。)
- .PANEL verb の前に入れる必要があります。

REXX 変数をパネル機能変数に関連付けられるようにする変数 ID 制御文字も定義できます。そのため、REXX 変数名の代わりに変数 ID 制御文字をパネル定義に埋め込むことができます。同じ REXX 変数を、異なる変数 ID 制御文字に割り当てることができます。

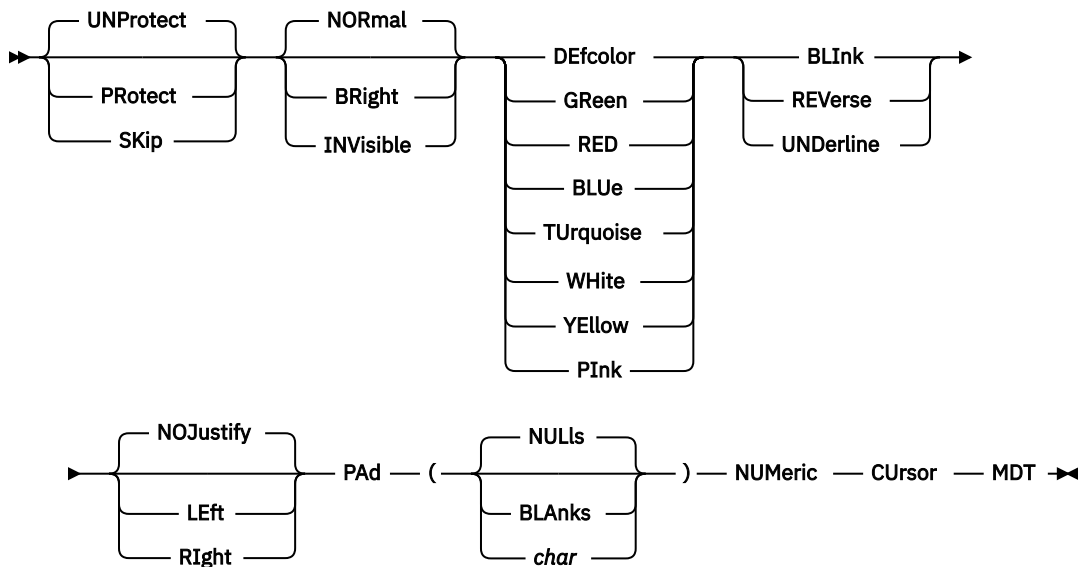
- 語幹名はピリオドで終わる必要があります。
- 変数リストには、使用される変数よりも多くの変数を含めることができます。
- 変数リストに含まれている変数が、使用される変数よりも少なくはいけません。その場合は、エラーが発生します。
- 変数 ID 制御文字は複数定義でき、各文字は互いに独立しています。

.DEFINE

.DEFINE verb の形式を示します。独自の制御文字を定義しない場合に指定されるデフォルトの制御文字も示します。



オプション



デフォルトのフィールド制御文字

Defcolor スキップは正常です
+ Defcolor 保護は高輝度です
% Defcolor 無保護は正常です
! Defcolor 無保護は高輝度です
& 変数 ID

オペランド

char

定義する制御文字を指定します。

Variable

REXX 変数 ID 制御文字を定義します。変数 ID 制御文字は、パネル機能制御文字を REXX 変数名に関連付けるために使用されます。複数の変数制御文字を一度に定義することができます。VARIABLE キーワードの後ろに、変数名 (*variable*) のリストまたは単一語幹名 (*stem.*) が続く場合があります。変数リストには、1 個から 32,767 個の変数名を含めることができます。指定できる語幹名は 1 つのみで、語幹名はピリオドで終わる必要があります。このピリオドは、語幹として変数を識別するため、ピリオドを省くと、名前は単純変数として解釈されます。

変数リストと語幹名を混用することはできません。パネル生成プログラムが変数制御文字を検出すると、置換が行われます。単純変数のリストは、リストされているとおりの順序で置き換えられます。例えば、3 番目の変数制御文字は、その制御文字用にリストされた 3 番目の変数に置き換えられます。語幹変数は、語幹名に 3 文字の番号 (末尾) を付加して置き換えられます。この番号は 1 から始まり、語幹制御文字が検出されるたびに増やされます。したがって、特定の語幹用の 10 番目の語幹制御文字には、末尾として 10 が付きます (STEM.10)。これらの変数は REXX 変数であるため、REXX 変数命名規則に従わなければなりません。

DROP

フィールド制御文字として *char* をドロップします。

オプション

UNProtect

フィールドをオペレーター入力から保護しないことを指定します。(これはデフォルトです。)

Protect

フィールドをオペレーター入力から保護することを指定します。

SKip

自動スキップ機能をもつ保護フィールドを指定します。直前の無保護フィールドの最後の位置にオペレーターが文字を 1 文字入力すると、カーソルはこのフィールドをスキップします。

NORmal

フィールドを強調表示しないことを指定します。(これはデフォルトです。)

BRight

フィールドを強調表示することを指定します。

INVisible

フィールドを非表示にすることを指定します。

Green
RED
BLUe
TUrquoise
WHite
YELLOW
PInk
DEfcolor

カラーの選択肢です。

注:

1. デフォルトのカラーを指定しない場合、カラーはフィールド・タイプと輝度の値に基づきます。
protect/normal は青、protect/bright は白色、unprotect/normal は緑、および unprotect/bright は赤で表示されます。
2. パネル上の任意のフィールドにカラー (DEFCOLOR など) が明示的に指定されている場合には、
DEFCOLOR が指定されるか、またはカラーが指定されていない高輝度フィールドはすべて白で表示され、DEFCOLOR が指定されるか、またはカラーが指定されていない通常のフィールドは緑で表示されます。これは、3270 ハードウェア制限であり、パネルの機能ではありません。

BLInk

フィールドが明滅することを指定します。

REVerse

フィールドが反転表示であることを指定します。

UNDerline

フィールドに下線を付けることを指定します。

NOJustify

位置調整が行われない (左寄せは行われるが、ブランクは取り除かれない) ことを指定します。

LEft

フィールドを左寄せする (先行ブランクは取り除かれます) ことを指定します。

RIght

フィールドを右寄せする (末尾ブランクは取り除かれます) ことを指定します。

PAd()

変数を持つフィールドのコンテキストでのみ指定します。無保護フィールドでは、埋め込み文字は、変数値が埋め込まれていない文字位置に埋められます。保護フィールドでは、埋め込み文字は無保護フィールドと似ていますが、埋め込み域の範囲は無保護フィールドのようにフィールド全体ではありません。これは、フィールドの終わり、あるいは次の変数またはテキストの開始までの保護フィールド内で、変数の開始位置によってバインドされます。

NULLs

フィールドにヌル文字が埋め込まれることを指定します。

BLAnks

フィールドにブランクが埋め込まれることを指定します。

char

フィールドの埋め込みに使用する単一文字を指定します。

NUMeric

フィールドが数値であることを指定します (無保護フィールドのみ)。

Cursor

カーソルをこのフィールドの先頭に配置することを指定します。複数のカーソル・フィールドが定義されている場合は、最後に定義されたフィールドにカーソルが含まれます。カーソル・フィールドが定義されていない場合、カーソルは左上隅に置かれます。

MDT

このフィールドについて変更ビット・タグをオンに設定します。フィールドがオペレーターによって変更されなかった場合でも、常に、読み取りでこのフィールドを戻します。

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料](#)で使用されている表記規則および用語も参照してください。

.PANEL verb を使用した実際の PANEL レイアウトの定義

.PANEL verb は、パネル・レイアウトの開始を通知し、.DEFINE verb の後に続きます。

この verb が列 1 で開始し、大文字であることも必要です。パネル定義内の行は、その行を表示したい位置と同じ位置に指定してください。ブランク行を含め、panel verb の後に入力されるテキストはすべて、パネル生成プログラムにとって意味のあるものであるため、コメント行は許可されません。最初の文字が、保護、スキップ、または無保護の制御文字でなければなりません。パネル定義は、ファイルの終わりか、または列 1 から始まる次の .PANEL verb の終わりで終了します。サポートされるパネル定義は、1 ファイルにつき 1 つのみです。

パネル・レイアウトは、表示されるものに近いものですが、パネルが表示されるときに制御文字と埋め込み変数は表示されない点は例外です。列 10 から始まる .PANEL の後の 3 つ目の行で入力されるフィールドは、端末画面の 3 行目、列 10 に配置されます。

.PANEL verb は、次の特性を持っています。

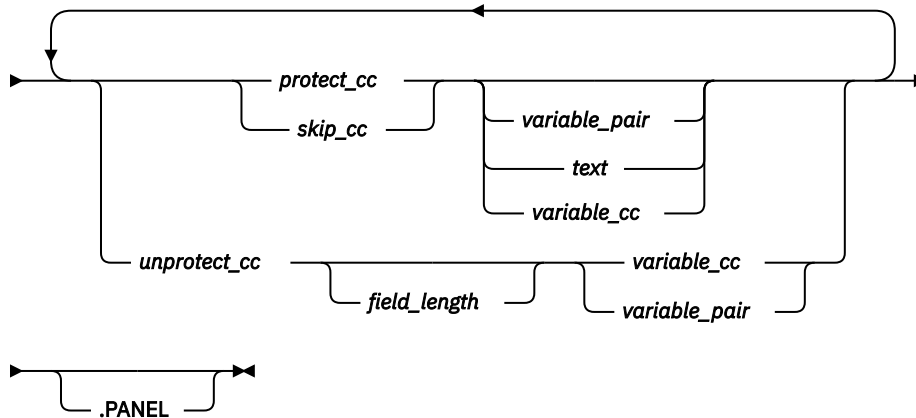
- 最初の列で始まり、後ろに、スペースを 1 つ続け、大文字でなければなりません。
- パネル標識の終わりでない限り、.PANEL と同じ行にパネル名が入っている必要があります。
- 少なくとも 1 つのフィールドを持ち、.PANEL 行に続く最初の文字が、保護、スキップ、または無保護の制御文字でなければなりません。
- 明示的な入力フィールド長が使用されている場合を除き、フィールドは、次のフィールドの始めで終了します。空のフィールドを使用して、フィールドを終了することができます。
- 制御文字を正規表示可能文字として使用するには、2 つの連続する制御文字を入力します。2 つの連続する制御文字を表示するには、4 つの連続する制御文字を入力します。制御文字のペアの後ろに 7 つ以上の連続するスペースがある場合を除き、1 つの文字として表示される制御文字のペアごとに、フィールド内の後続テキストが位置 1 つ左に表示されます。
- 保護またはスキップ・フィールドには、そのフィールドに収まるだけの任意の数の変数またはテキストを入れることができます。保護フィールド内の変数は、置換に使用できる領域があり、変数 ID 制御文字で始まり、次の制御文字またはテキストの直前で終わります。
- 変数 ID 制御文字と変数名の間にスペースを入れることは許されません。スペースを入れると、変数名がブレース・テキストとして解釈されます。
- 無保護フィールドに指定できる関連変数は 1 つだけです。
- 無保護フィールドに、フィールド長を明示的に示す数値を入れることができます。この数値は、無保護制御文字と変数制御文字の間に入れる必要があります。明示的なフィールド長が使用された場合、フィールドは終了を必要としません。明示的な長さフィールドが 1 行の最後のフィールドである場合は、フィールドの終了を強制するスキップ属性を使用して終了フィールドが作成されます。
- 明示的な入力フィールド長が使用された場合、後に続くフィールドの列位置はすべて、左寄せまたは右寄せされて、適切な位置合わせが強制されるため、明示的な長さフィールドは、直後に次の隣接するフィールドが続きます。その行のその他のフィールドのスペーシングは、そのままになります。この位置合わせは、その 1 行にのみ影響を与え、次の行のフィールドには影響しません。例えば、ある行にフィールドが 5 つあり、最初のフィールドと 4 番目のフィールドに明示的な長さとスペーシングが含まれています。フィールド 1 とフィールド 2 の間は 4 で、フィールド 2 とフィールド 3 の間は 5、フィールド 3 とフィールド 4 の間は 6、フィールド 4 とフィールド 5 の間は 7 です。この行が表示されたとき、フィールド 2 はフィールド 1 の直後から始まり (スペースなし)、フィールド 2 とフィールド 3 の間隔は 5 のまま、フィールド 3 とフィールド 4 の間隔は 6 のままで、フィールド 5 はフィールド 4 の直後に続きます (スペースなし)。明示的な長さが原因で、現行フィールドまたは後続フィールドが次のフィールドにオーバーフローする場合は、パネルの表示時にエラーが戻されます。
- フィールドが同じ行で終わらない (例えば、フィールドが複数行にまたがる) 場合、フィールド長は、パネルが表示されている画面の幅に応じて変わります。また、最後のフィールドに終了文字がない場合、そのフィールドは、最初のフィールドの開始が見つかるまで、画面を折り返します。
- パネル出力中に属性が動的に変更できるのは、変数が含まれているフィールドのみです。

- 保護フィールドかスキップ・フィールドが動的に変更されて無保護フィールドになる場合、フィールド内の最初の変数にのみ、オペレーターの入力割り当てられます。入力フィールドの内容のすべてが割り当てられます。
- パネル・ソース・ファイルは、REXX ファイル・システム内に収容されてからでないと、ランタイム・パネル機能で使用する中間ファイル (パネル・オブジェクト) に処理できません。

.PANEL

.PANEL verb の形式を示します。

➡ .PANEL — *panel_name* ➡



オペランド

panel_name

定義するパネルを指定します。これは、長さが 1 文字から 8 文字で、REXX ファイル・システム・ファイル名の規則に従うものでなければなりません。283 ページの『第 24 章 REXX/CICS ファイル・システム』を参照してください。

注: *panel_name* は、RFS ファイル名と同じでなければなりません。完全な RFS ファイル名は *panel_name*.PANSRC です。

protect_cc

保護フィールド制御文字を指定します。

skip_cc

スキップ・フィールド制御文字を指定します。

variable_pair

後ろに変数名が続いている変数制御文字を指定します。(これらの間にスペースを入れることはできません。)

text

表示可能文字。

variable_cc

変数 ID 制御文字を指定します。

unprotect_cc

無保護制御文字を指定します。

field_length

明示的な入力フィールドの長さ値を指定します。

パネル生成とパネル入出力

パネル定義は、REXX 環境の外で行うことができますが、パネル生成と入出力は、REXX EXEC または REXX 対話環境で実行されます。

REXX 対話環境は、初期パネル開発をテストするのに理想的な場所です。表示をテストするために、パネルは、TEST パネル・コマンドを使用します。これにより、パネル・オブジェクト・ファイルが作成されずにパネルが表示されます。また、パネル上の変数の置換はありません。パネル・オブジェクトを作成するには、GENERATE、SEND、または CONVERSE の各パネル・コマンドを使用します。FILE キーワードを使用して、パネル・ソースを見つめる RFS 内のディレクトリーを明示的に示すか、あるいはデフォルトの現行ディレクトリーにします。

パネル・ソース名は、ファイル名としてパネル名を持ち、ファイル・タイプとして PANSRC を持つ必要があります。パネル・オブジェクトは、ファイル名がソース・ファイル名と同じで、かつ PANOBJ というファイル・タイプを持つパネル・ソースと同じディレクトリーで作成し、ファイルされます。GENERATE は、パネル・オブジェクトを作成しますが、そのパネルを表示することはありません。SEND は、パネル・オブジェクトを作成し、パネルを表示し、変数置換を試行します。CONVERSE は、暗黙の待機と受信に関して、SEND と似ています。

注：REXX 対話環境の使用という側面的な影響があります。複数のパネル・キーワードがそれぞれ異なる働きをします。SEND でのカーソル位置は無視され、SEND および CONVERSE の場合は、キーボード・ロックも無視されます。

PANEL コマンドの特性は以下のとおりです。

- すべての引数またはキーワードが、すべてのコマンドに対して意味を持つものであったり、有効であるとは限りません。
- REXX EXEC 内の最後のパネル・コマンドは、END コマンドです。これは、それより前のパネル・コマンドが保持しているストレージをすべて解放します。このコマンドに他のオペランドは不要です。
- 関連する変数を持つフィールドだけが、属性を動的に変更することができます。
- パネルのオブジェクト・ファイルが存在しない場合、表示中のパネルに対して作成されます。このファイル名はパネル・ソース・ファイル名と同じであり、ファイル・タイプは 'PANOBJ' になります。
- フィールドが入力フィールドに変更されるときに端末オペレーターが入力した入力は、保護/スキップ・フィールド内の最初の変数にのみ割り当てられます。
- この動的属性の変更は、現在のパネル実行に影響しますが、持続することはありません。後続のパネル表示は、属性が再び動的に変更されない限り、パネル定義ステップによって静的に定義されたものに戻ります。
- SEND は RECEIVE より前に実行する必要があります、パネル名が一致している必要があります。
- 属性のみが変更を指示し、その他は、静的に定義されたままです。
- オペレーター入力が REXX 変数に正しく割り当てられるように、位置引数で送信されたパネルを同じ位置の値で受信する必要があります。
- 複数のフィールドがリストされている場合、フィールド ID リストを括弧で囲んでください。
- それぞれの異なる属性変更ごとに 1 つの ATTRIBUTE 引数を指定する必要があります。

注：

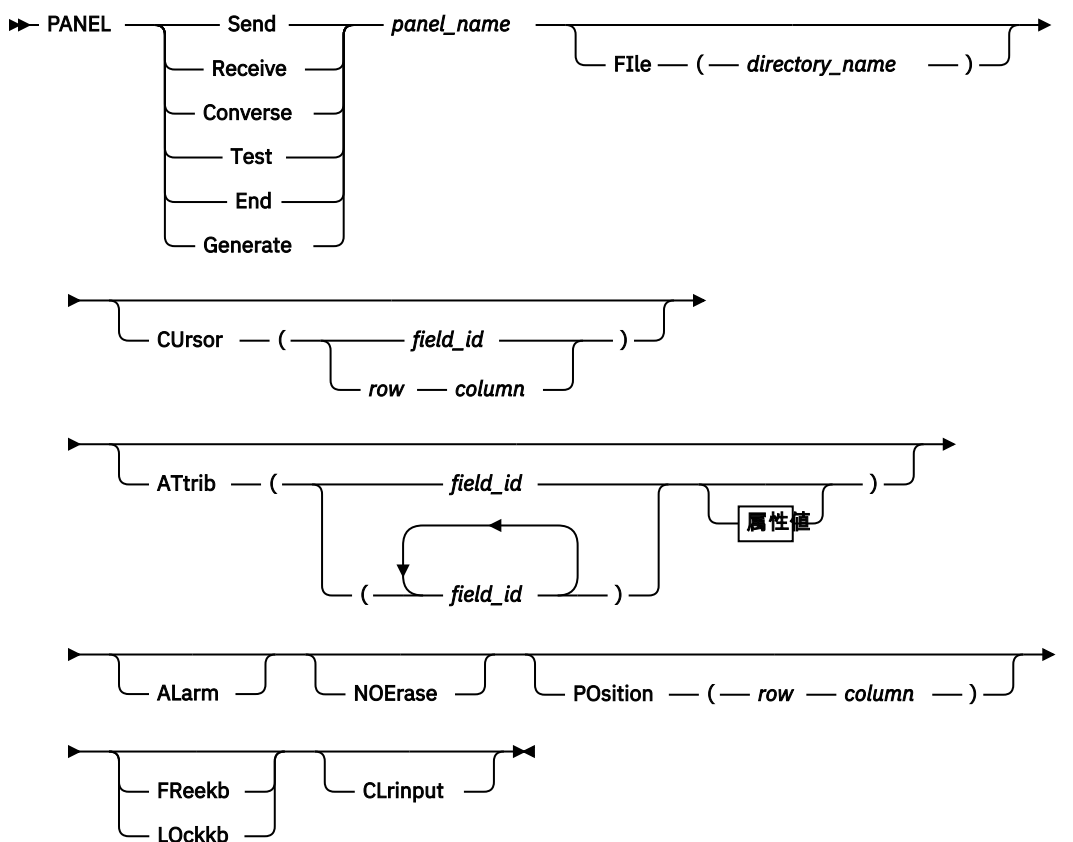
- ATTRIBUTE 引数の数を動的に変更するには、REXX 変数を使用しなければなりません (REXX 変数にリテラル属性ストリングを入れ、その変数を使用します)。例えば、オペレーターの入力に応じて、あるフィールドを青色に変更したり、あるフィールドを青色に変更して別のフィールドを明滅させたりするプログラムを作成する必要がある場合があります。次に解決例を示します。

```
field_id = 'xxxx'; /* name of field needing attribute changed*/
attr_string = 'attr(' field_id 'blue )';
if operator_input = y then
    attr_string = attr_string 'attr(' field_id2 'blink )'
'panel send panel_name' attr_string;
```

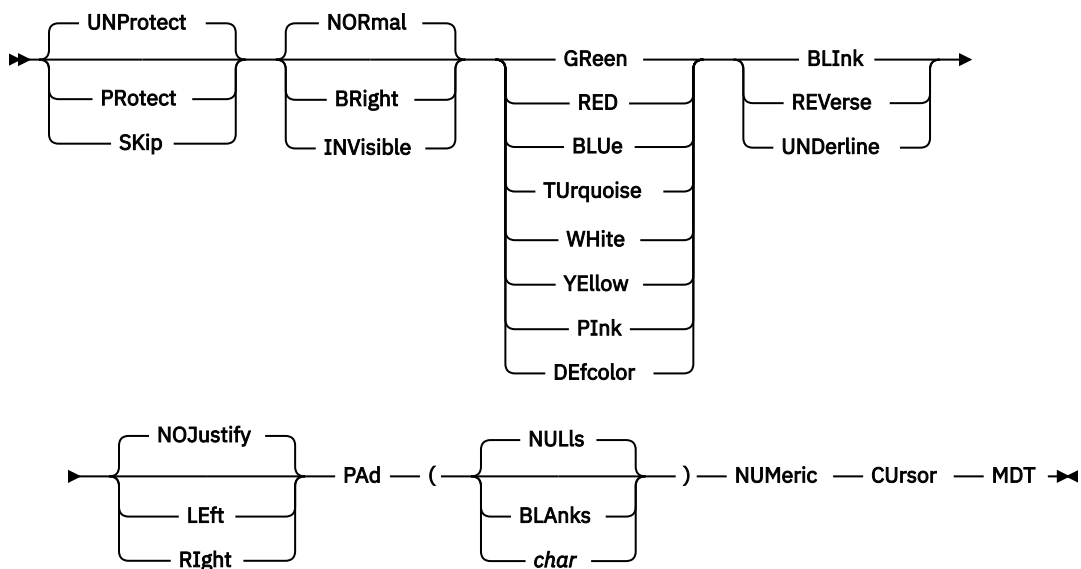
- REXX パネル機能は、オブジェクトが見つからない場合や、オブジェクトがソースより古い場合には、パネル・オブジェクトを生成します。

PANEL RUNTIME

PANEL RUNTIME コマンドの形式を示します。



属性値



オペランド

送信

パネルを送信するパネル・コマンドです。

受信

パネルを受信するパネル・コマンドです。

Converse

パネルを送信し、オペレーターの入力を待つパネル・コマンドです。

Test

パネルを表示するパネル・コマンドです。中間ファイル (パネル・オブジェクト) は作成されず、変数置換は試行されません。

終了

パネル・セッションを終了するコマンドです。コマンドは、パネル機能によって保持されているすべてのストレージを解放します。このコマンドには引数はなく、指定された引数はすべて無視されます。

Generate

パネル・オブジェクトを作成する明示コマンドです。パネルは表示されません。

panel_name

入出力または生成するパネルの名前を指定します。

File()

このパネルが含まれている RFS ディレクトリーの名前 *directory_name*) を指定します。(END 以外のすべてのパネル・コマンドに指定します。)

Cursor()

(SEND および CONVERSE の場合にのみ指定) パネル上にカーソルを配置します。

field_id

パネル上でカーソルを配置する REXX 変数名を指定します。

row

カーソルを配置する、パネル内の行を指定します。行の値は、パネルの開始行からの相対値です。パネルのデフォルトの開始行は 1 ですが、POSITION() キーワードを使用して変更できます。

column

カーソルを配置する、パネル内の列を指定します。列の値は、パネルの開始列からの相対値です。パネルのデフォルトの開始列は 1 ですが、POSITION() キーワードを使用して変更できます。

ATtrib(*field_id attribute_values*)|((*field_ids*) *attribute_values*)

(SEND および CONVERSE の場合にのみ指定) 属性を動的に設定して、パネル定義に指定されている属性をオーバーライドします。

field_id

属性が動的に設定されるフィールドを指定します。これは、そのフィールドと関連付けられている変数名でなければなりません。フィールドのリストを指定する場合は、括弧で囲む必要があります。

attribute_values

設定する属性を指定します。指定された属性のみが変更され、その他の属性は静的に定義された値にデフォルト設定されます。例えば、最初に RED および UNDERLINE として定義されたフィールドに BLUE を指定した場合、そのフィールドは BLUE および UNDERLINE となります。

属性のリストについては、[333 ページの『属性』](#)を参照してください。

ALarm

(SEND および CONVERSE の場合のみ指定) パネルを表示するときに、ベルを鳴らします。(デフォルトはアラームなしです)。

NOErase

(SEND および CONVERSE の場合のみ指定) この画面を表示する前の画面を消去しません。(デフォルトでは、パネル作成の前に消去されます)。

この 2 番目のパネルの表示中にその基礎のパネル内の属性が変更されたり、基礎のパネル内の文字が 2 番目のパネル内に表示されたりするなどの予期しない結果を防ぐために、以下の指針に従ってください。

- 2 番目のパネル内で変数を使用する場合は、その位置が基礎のパネルのブランク域内になるようにします。
- 2 番目のパネルの終端が画面の右側の境界になるように配置するか、基礎のパネルのブランク列のみが続くように配置します。

POsition()

(SEND、CONVERSE、および RECEIVE の場合にのみ指定) 出力画面上にパネルを配置します。行 (*row*) および列 (*column*) は、パネルの左上隅の開始位置を指定します。(デフォルトは行 1 列 1 です)。例えば、POS(5 10) は、パネルが行 5 と列 10 から始まることを意味し、実際の移動は下に 4 行、右に 9 列です。

FReekb

(SEND および CONVERSE の場合のみ指定) オペレーターが入力できるように、キーボードを解放します。(これはデフォルトです。)

LOckkb

(SEND および CONVERSE の場合のみ指定) キーボードをロックします。

CLrinput

(SEND および CONVERSE の場合のみ指定) パネルを表示する前にすべての入力フィールドをクリアします。変数置換は試みられず、入力域に埋め込み文字が充てんされます。

属性**UNProtect**

フィールドをオペレーター入力から保護しないことを指定します。

PRotect

フィールドをオペレーター入力から保護することを指定します。

SKip

自動スキップ機能をもつ保護フィールドを指定します。直前の無保護フィールドの最後の位置にオペレーターが文字を 1 文字入力すると、カーソルはこのフィールドをスキップします。

NORmal

フィールドを強調表示しないことを指定します。

BRight

フィールドを強調表示することを指定します。

INVisible

フィールドを非表示にすることを指定します。

GReen**RED****BLUe****TURquoise****WHite****YELlow****PInk****DEfcolor**

カラーの選択肢です。

注:

1. デフォルトのカラーを指定しない場合、カラーはフィールド・タイプと輝度の値に基づきます。protect/normal は青、protect/bright は白色、unprotect/normal は緑、および unprotect/bright は赤で表示されます。
2. パネル上の任意のフィールドにカラー (DEFCOLOR など) が明示的に指定されている場合には、DEFCOLOR が指定されるか、またはカラーが指定されていない高輝度フィールドはすべて白で表示され、DEFCOLOR が指定されるか、またはカラーが指定されていない通常のフィールドは緑で表示されます。これは、3270 ハードウェア制限であり、パネルの機能ではありません。

BLInk

フィールドが明滅することを指定します。

REVerse

フィールドが反転表示であることを指定します。

UNDerline

フィールドに下線を付けることを指定します。

NOJustify

位置調整をしないことを指定します。

入力の場合、前後のブランクや埋め込み文字は除去されません。

出力の場合、前後のブランクや埋め込み文字は除去されません。必要に応じて、右側のデータが切り捨てられます。切り捨てによって実行が中断することはありませんが、戻りコード 4 と理由コード 117 が返されます。変数データの右側のヌルの位置は埋め込み文字に置き換えられます。

LEft

フィールドを左寄せすることを指定します。

入力の場合、前後のブランクや埋め込み文字は除去されます。

出力の場合、先行ブランクが除去され、変数データの右側のヌルが埋め込み文字に置き換えられます。先行ブランクが除去された後に、必要に応じて右側のデータが切り捨てられます。切り捨てによって実行が中断することはありませんが、戻りコード 4 と理由コード 117 が返されます。

RIght

フィールドを右寄せすることを指定します。

入力の場合、前後のブランクや埋め込み文字は除去されます。

出力の場合、末尾ブランクが除去され、変数データの左側のヌルが埋め込み文字に置き換えられます。末尾ブランクが除去された後に、必要に応じて左側のデータが切り捨てられます。切り捨てによって実行が中断することはありませんが、戻りコード 4 と理由コード 117 が返されます。出力の場合、変数データの左側のヌルとブランクの位置が埋め込み文字に置き換えられます。

PAAd()

変数を持つフィールドのコンテキストでのみ指定します。無保護フィールドでは、埋め込み文字は、変数値が埋め込まれていない文字位置に充てんされます。保護フィールドでは、埋め込み文字は無保護フィールドと似ていますが、埋め込み域の範囲は無保護フィールドのようにフィールド全体ではありません。これは、フィールドの終わり、あるいは次の変数またはテキストの開始までの保護フィールド内で、変数の開始位置によってバインドされます。

NULLs

フィールドにヌル文字が埋め込まれることを指定します。

BLanks

フィールドにブランクが埋め込まれることを指定します。

char

フィールドの埋め込みに使用する単一文字を指定します。

NUMeric

フィールドが数値であることを指定します (無保護フィールドのみ)。

CUrsor

カーソルをこのフィールドの先頭に配置することを指定します。複数のカーソル・フィールドが定義されている場合は、最後に定義されたフィールドにカーソルが含まれます。カーソル・フィールドが定義されていない場合、カーソルは左上隅に置かれます。

MDT

(SEND および CONVERSE の場合のみ指定) すべての入力フィールドの変更ビット・タグをパネル上に設定します。

PANEL 変数

REXX プログラムが使用できる暗黙的に定義された PANEL 変数と、PANEL エラー関連の変数がリストされます。

PAN.AID

最後のパネル入力の原因となったアテンション ID。

ENTER	ENTER
CLEAR	CLEAR
CLRP	CLEAR PARTITION
PEN	SELECTOR PEN
OPID	OPERATOR ID

MSRE	MAGNETIC READER
STRF	STRUCTURE FIELD
TRIG	TRIGGER
PA1	
PA2	
PA3	
PF1	
PF2	
PF3	
PF4	
PF5	
PF6	
PF7	
PF8	
PF9	
PF10	
PF11	
PF12	
PF13	
PF14	
PF15	
PF16	
PF17	
PF18	
PF19	
PF20	
PF21	
PF22	
PF23	
PF24	

PAN.CURS

最後のパネル入力内でのカーソルの位置。これは、ブランクで区切られた、行/列の形式になっています。例えば、'10 5' は行 10 列 5 になります。行および列の値は、画面の始まりに対して絶対であり、POSITION() キーワードの影響を受けません。

PAN.CNAM

カーソル位置に関連付けられた REXX 変数名 (フィールド ID)。フィールドに関連付けられた変数がない場合、PAN_CNAM は更新されません。

PANEL エラー関連の変数は、以下のとおりです。

PAN.REA

警告またはエラーが発生した場合の理由コード。最初に、REXX 戻りコード RC を調べる必要があります。RC が 10 の場合、PAN.REA には、エラーの判別に役立つ状態コードと入力コードが含まれています。詳細については、[338 ページの『状態コードと入力コード』](#)を参照してください。

PAN.LOC

内部ロケーション・コード。IBM サポートが使用する 3 桁から 4 桁の数値です。REXX 変数 RC に値 10 が含まれている場合、PAN.REA と一緒に PAN.LOC を使用してエラーを判別してください。

PAN.LINE

パネル・オブジェクトの生成の検出時に発生したエラーが検出された、ソース・パネル定義内の行番号です。

パネル機能戻りコード情報

パネル・コマンドの主要戻りコードは、REXX 変数 RC で設定されます。戻りコードの各レベルには、理由コードと行番号の形式で追加情報が付加されます (該当する場合)。

パネル・ソース・コード (ロケーション・コードが 11xx または 12xx) の処理中にエラーが検出され、RC が 12 でも 16 でもない場合、REXX 変数 PAN.LINE に、誤った行の番号が含まれています。

戻りコード

4

警告。パネル機能は処理を続行します。他の戻りコード値については、処理は停止します。

8

プログラマー・エラー

10

プログラマー・エラーです。PAN.REA に、このエラーの原因を判別するのに役立つ詳しい情報が含まれています。詳細については、[338 ページの『状態コードと入力コード』](#)を参照してください。

12

CICS コマンド・エラー。CICS EIBRESP がパネル理由コードに戻されます。このエラーをプログラマーが解決できない場合は、エラーの再現に必要な情報を可能な限り保管および収集して、IBM サポートに連絡してください。

14

RFS エラー。理由コードに RFS 戻りコードが含まれています。

16

内部システム・エラー。エラーの再現に必要な情報を可能な限り保管および収集して、IBM サポートに連絡してください。

システム・エラー理由コード

401

コマンドの処理中に、パネル機能はストレージ・スペースを使い尽くしました。

402

内部制御文字 ID テーブルと制御文字情報テーブルが同期していません。

403

パネル・オブジェクト・データが破損しています。最初に、ファイルが正しいか、また、それがパネル・オブジェクトであるかを確認してください。

404

CICS 受信バッファが破損しています。

405

検証要求に誤りがあります。

406

ストレージ解放要求が失敗しました。

407

ストレージ取得要求が失敗しました

408

REXX 変数を取得または入れようとしたが、失敗しました。

409

援助プログラムが不明です

410

動的一致エラー。

プログラマー提供の警告およびエラーの理由コード

101

キーワードが反復されているか、または類似カテゴリ内のキーワードが反復されています。例えば、RED と BLUE、UNDERLINE と REVERSE です。

102

同時指定できないキーワードです。例えば、PROTECT と NUMERIC です。

103

キーワードまたはパネル名が欠落しています。

104

定義される制御文字が無効であるか、または欠落しています。

105

パネルが、画面に対して大きすぎます。

108

括弧が欠落しています。

- 109**
埋め込み文字が無効です。
- 110**
タスクに関連付けられた端末がありません。
- 111**
パネルに、受信時に定義された入力フィールドがありません。
- 112**
パネル名が無効です。
- 115**
REXX 変数名 (またはフィールド ID) が無効です。
- 116**
数値が誤って指定されています。パネル・ソースまたは行/列の値に明示的な長さの値が入っています。
- 117**
変数値が長すぎるため、出力フィールドに合わせて切り捨てられました。
- 118**
テキスト・フィールドは切り捨てられました。明示的な長さにより、後続のフィールドが別のフィールドにオーバーレイしていないかどうかを確認してください。
- 119**
パネル・コマンドに誤りがあるか、または欠落しています。SEND、RECEIVE、CONVERSE、TEST、またはENDでなければなりません。
- 122**
変更されたフィールドを受け取りましたが、対応する入力フィールド定義がありませんでした。
- 124**
受信したバッファは空です。CLEAR キー、ENTER キー、および PA キーがこの原因です。
- 125**
ファイル名が無効です。
- 126**
ATTRIBUTE キーワードのフィールド ID がパネルにありませんでした。
- 129**
コマンドで指定された引数が多すぎます。
- 130**
SEND は、RECEIVE より前に実行する必要があります。
- 131**
パネル・ソースにパネル定義がありません。おそらく、.PANEL が列 1 に入っていないか、後ろにスペースがなかったためです。
- 132**
不明なキーワードが検出されました。
- 133**
現在アクティブでない制御文字に対して DROP が要求されました。
- 136**
継続は有効になっていますが、ソースの終了が検出されました。
- 137**
指定された行または列が、エラーの原因である可能性がある現行表示 CURSOR() または POSITION() に対して大きすぎます。
- 138**
変数 ID 制御文字が定義されており、複数の語幹名がリストされました。
- 139**
制御文字定義にリストされているよりも多くの変数フィールドが定義されています。
- 140**
明示的な入力フィールド値が原因で、フィールドが画面の終わりを超えてしまいました。

143

ファイル内のパネル名が、パネル・ランタイム・コマンドのパネル名と一致しません。

144

パネルの行数が、現在の画面で対応できる行数を超えています。

145

パネルの列数が、現在の画面で対応できる列数を超えています。

状態コードと入力コード

戻りコード 10 の場合、理由コード (PAN.REA 内) は、特別な意味を持ちます。

理由コードは、2 桁の数字コード 2 つで構成され、最初の数値は、処理されるキーワード (状態コード) を示し、2 番目の数値は、エラー状態の原因となった入力 (入力コード) を示します。PAN.LOC のロケーション・コードを使用して、使用する状態コード・リストと入力コード・リストを決定できます。

以下に例を示します。

```
RC = 10  
PAN.REA = 0199  
PAN.LOC = 1177
```

11xx ロケーション・コードの状態コードおよび入力コードのリストを使用した場合、処理されるキーワードは、状態コード 01 の .DEFINE であり、入力コードに、入力コード 99 の不明なシンボルが含まれています。

11xx ロケーション・コードの状態コード

01

.DEFINE および制御文字

02

フィールド・タイプ (protect/skip/unprotect)

03

カラー (red/blue/green/...)

04

輝度 (bright/normal/invisible)

05

位置調整 (left/right/nojustify)

06

数値

07

拡張強調表示 (blink/reverse/underline)

08

MDT

09

Cursor

10

Pad()

11

可変

12

Drop

20xx ロケーション・コードの状態コード

01

パネル・コマンド (send/receive/converse/...)

- 02**
File()
 - 03**
Cursor()
 - 04**
Position()
 - 05**
Alarm
 - 06**
Noerase
 - 07**
キーボード・ロック (lockkb/freekb)
 - 08**
Clrinput
 - 09**
属性
 - 10**
フィールド・タイプ (protect/skip/unprotect)
 - 11**
カラー (red/blue/green/...)
 - 12**
輝度 (bright/normal/invisible)
 - 13**
位置調整 (left/right/nojustify)
 - 14**
数値
 - 15**
拡張強調表示 (blink/reverse/underline)
 - 16**
MDT
 - 17**
Cursor
 - 18**
Pad()
 - 19**
ATTRIBUTE 引数の右括弧
- 11xx および 20xx のロケーション・コードの入力コード**
- 01**
フィールド・タイプ (protect/skip/unprotect)
 - 02**
カラー (red/blue/green/...)
 - 03**
輝度 (bright/normal/invisible)
 - 04**
位置調整 (left/right/nojustify)
 - 05**
数値
 - 06**
拡張強調表示 (blink/reverse/underline)

- 07** MDT
- 08** Cursor
- 09** Pad()
- 10** 可変
- 11** Drop
- 12** (未使用)
- 13** File()
- 14** Cursor()
- 15** Position()
- 16** Alarm
- 17** Noerase
- 18** キーボード・ロック (lockkb/freekb)
- 19** Clrinput
- 20** 属性
- 21** ATTRIBUTE 引数の右括弧
- 98** コマンドの終わり。さらに多くのオペランドが予想されていました
- 99** 不明なシンボル。キーワードまたは制御文字が予想されていました。
-
- 12xx ロケーション・コードの状態コード**
- 01** パネル名
- 02** 保護/スキップ・フィールド
- 03** 無保護フィールド
- 04** 保護/スキップ・フィールド内のテキスト
- 05** (未実装)
- 06** 明示的な入力フィールドの長さ番号
- 07** 無保護変数

08

保護/スキップ変数

12xx ロケーション・コードの入力コード

01

プレーン表示テキスト

02

明示的な長さ番号

03

保護フィールド制御文字

04

無保護フィールド制御文字

05

変数制御文字

07

パネルの終わり

08

無効または不明な入力

ロケーション・コード

1000 未満の番号は、メインライン処理プログラムに記載されています。

10xx

パネル生成プログラムの共通処理プログラム

11xx

.DEFINE verb 処理プログラム

12xx

.PANEL verb 処理プログラム

20xx

パネル・ランタイム・コマンド・プロセッサ

21xx

動的属性解決処理プログラム

30xx

出力 3270 データ・ストリーム処理プログラム

40xx

入力 3270 データ・ストリームおよび REXX 変数割り当て処理プログラム

90xx

CICS インターフェース 処理プログラム

サンプル・パネルの例

例 1

```
.DEFINE > prot blue
.DEFINE ? prot red
.DEFINE # unprot num green
.DEFINE < unprot invisible num
.DEFINE @ protect turq
.DEFINE + prot blue underline
.PANEL signon
> Panel signon                                &companyname

?&message

@                                Welcome to ACME On-Line Tax Services
```

```
+Please enter your Account Number and Personal ID Number and press ENTER>
```

```
>Account Number :#7&account_num
```

```
>PIN :<4&pin
```

例 2

```
.DEFINE > prot green
.DEFINE < unprot underline white
.DEFINE + var service.
.DEFINE % skip turq
.PANEL service
> Panel service &disp_date &companyname

% &salutation
% Tab the cursor to the type of service wanted and press the ENTER key.

<+> Itemized tax preparation
<+> Non-itemized tax preparation
<+> Query return status
<+> Show calendar
<+> Exit
```

例 3

```
.DEFINE # protect bright
.DEFINE + protect
A panel to display a static message without erasing previous panel.
Notice the position of the escape sequence in lines 1 and 6.
See product information for an explanation about escape sequences.
.PANEL msgbox1
#++-----+++
#|                                     |+
#| We are sorry but the service you have |+
#| chosen is not available at this time. |+
#| Press ENTER to continue.             |+
#|                                     |+
#++-----+++
```

例 4

```
.DEFINE ) protect bright
.DEFINE + drop
.DEFINE & var msg.
A panel to display output dynamic messages.
.PANEL msgbox2
)+-----+##
)|                                     |##
)| &                                 |##
)| &                                 |##
)|                                     |##
)+-----+##
```

例 5

```
.DEFINE > skip blue
.DEFINE < skip green right
.DEFINE % var center_days.
.DEFINE + var right_days.
.DEFINE # VAR left_days.
.DEFINE @ var pf3 pf7 pf8
.PANEL calendar
> Panel calendar &disp_date &companyname
```

> &disp_left_mon	&disp_center_mon	&disp_right_mon
>su mo tu we th fr sa	su mo tu we th fr sa	su mo tu we th fr sa
<# <# <# <# <# <# >	<% <% <% <% <% <% >	<+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# >	<% <% <% <% <% <% >	<+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# >	<% <% <% <% <% <% >	<+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# >	<% <% <% <% <% <% >	<+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# >	<% <% <% <% <% <% >	<+ <+ <+ <+ <+ <+ >
<# <# >	<% <% >	<+ <+ >

>@ = Leave Calendar >@ = Backup a month >@ = Go forward a month

注: 文字 @ # \$ ¢ は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。[CICS 資料](#)で使用されている表記規則および用語も参照してください。

REXX パネル・プログラムの例

```

/* REXX program sample using Panel Facility */
/* data base */
ACCOUNT.1234561 = '1231 John W. Smith Mr.'
ACCOUNT.1234562 = '1232 Jane M. Brown Miss'
ACCOUNT.1234563 = '1233 Mary R. Scott Mrs.'

MESSAGE = '' /* no output message yet */
COMPANYNAME = 'ACME On-Line Tax Services'
CURS_NAME = 'ACCOUNT_NUM' /* put cursor on LNAME field */
ATTR_STRING = '' /* no dynamic attributes on first send */
PATH_NAME = 'FILE(PPOOL1:¥USERS¥BLAKELY)'
CLR_INP_FIELDS = 'CLR'

DO FOREVER
  'PANEL SEND SIGNON' CLR_INP_FIELDS PATH_NAME ,
  'CURSOR(' CURS_NAME ') ' ATTR_STRING
  IF RC > 4 THEN /* more than a warning */
    SIGNAL ERROR /* clean up and exit */
  'PANEL RECEIVE SIGNON '
  IF RC > 4 THEN
    SIGNAL ERROR /* clean up and exit */

  /* check return code and reason code to see if any data received */
  IF RC=4 & PAN.REA = 124 THEN /* warning and no input received */
    ITERATE /* redisplay panel */

  CLR_INP_FIELDS = '' /* display input fields with variable values */
  IF &not;SEARCH(ACCOUNT_NUM) THEN /* search for account number */
    DO;
      MESSAGE = ' Account Number not found, Please re-ENTER Number'
      CURS_NAME = 'ACCOUNT_NUM' /* put cursor in ACCOUNT field */
      ATTR_STRING = 'ATTR( ACCOUNT_NUM REV)'
      ITERATE
    END;

  IF WORD(ACCOUNT.ACCOUNT_NUM,1) ~= PIN THEN /* pin mismatch ? */
    DO;
      MESSAGE = ' PIN Number is incorrect, Please check to see your',
      'Account Number is correct and re-ENTER your PIN';
      CURS_NAME = 'PIN';
      ATTR_STRING = 'ATTR( PIN REV)';
      ITERATE /* display the panel again */
    END;
  LEAVE ;
END ; /* forever */

```

```
SERVICE. = '';
DISP_DATE = DATE('U');      /* set to display current date */
MSG.1 = 'Be sure cursor is in the first column!';
MSG.2 = 'Press ENTER or and PF key to continue.';
SALUTATION = 'Hi' WORD(ACCOUNT.ACCOUNT_NUM,5) ,
              WORD(ACCOUNT.ACCOUNT_NUM,4) ||,
              ', How may we be a service to you?';
```

```
DO FOREVER;
  PAN.CNAM = '';
  'PANEL SEND SERVICE CURSOR(SERVICE.1)' PATH_NAME
  IF RC > 4 THEN
    SIGNAL ERROR; /* clean up and exit */
  'PANEL RECEIVE SERVICE'
  IF RC > 4 THEN
    SIGNAL ERROR; /* clean up and exit */
  SALUTATION = ''; /* greeting only once */
```

```
SELECT;
  WHEN PAN.CNAM = 'SERVICE.1' THEN
    CALL ITEMIZE_ROUTINE;
  WHEN PAN.CNAM = 'SERVICE.2' THEN
    CALL NON_ITEMIZE_ROUTINE;
  WHEN PAN.CNAM = 'SERVICE.3' THEN
    CALL QUERY_RET_ROUTINE;
  WHEN PAN.CNAM = 'SERVICE.4' THEN
    CALL CAL;
  WHEN PAN.CNAM = 'SERVICE.5' THEN
    CALL EXIT_ROUTINE;
```

```
OTHERWISE
  DO;
    'PANEL SEND MSGBOX2 POS(7 10) NOERASE' PATH_NAME
    IF RC > 4 THEN
      SIGNAL ERROR;
    'PANEL RECEIVE MSGBOX2'
    IF RC > 4 THEN
      SIGNAL ERROR;
    END;
  END; /* select */
END; /* do forever */
EXIT
```

```
SEARCH: ; ARG ACC_NUM ;
IF SYMBOL('ACCOUNT.ACC_NUM') == 'VAR' THEN
  RETURN(1)
ELSE
  RETURN(0);
```

```
ITEMIZE_ROUTINE:
NON_ITEMIZE_ROUTINE:
QUERY_RET_ROUTINE:
  'PANEL SEND MSGBOX1 POS(7 10) NOERASE' PATH_NAME
  IF RC > 4 THEN
    SIGNAL ERROR;
  'PANEL RECEIVE MSGBOX1'
  IF RC > 4 THEN
    SIGNAL ERROR;
  RETURN;
```

```
CAL: PROCEDURE
COMPANYNAME = 'ACME On-Line Tax Service';
PATH_NAME   = 'FILE(POOL1:¥USERS¥BLAKELY¥)';
DISP_DATE = DATE('U');
```

```
/* calling date function in on statement ensures consistent date */
/* data save has format of YYYYMMDDNNNNNNN */
DATE_SAVE = DATE('S') || DATE('B');
```

```
/* set number of days each month has */
NUM_OF_DAYS.1 = 31;
NUM_OF_DAYS.3 = 31;
NUM_OF_DAYS.4 = 30;
NUM_OF_DAYS.5 = 31;
```

```

NUM_OF_DAYS.6 = 30;
NUM_OF_DAYS.7 = 31;
NUM_OF_DAYS.8 = 31;
NUM_OF_DAYS.9 = 30;
NUM_OF_DAYS.10 = 31;
NUM_OF_DAYS.11 = 30;
NUM_OF_DAYS.12 = 31;

```

```

/* set names of each month for panel display */
MONTH_NAME.1 = 'January';
MONTH_NAME.2 = 'February';
MONTH_NAME.3 = 'March';
MONTH_NAME.4 = 'April';
MONTH_NAME.5 = 'May';
MONTH_NAME.6 = 'June';
MONTH_NAME.7 = 'July';
MONTH_NAME.8 = 'August';
MONTH_NAME.9 = 'September';
MONTH_NAME.10 = 'October';
MONTH_NAME.11 = 'November';
MONTH_NAME.12 = 'December';

```

```

/* get number of complete days from year 1900 to 1st of month */
TOT_DAYS = SUBSTR(DATE_SAVE,9,6)-SUBSTR(DATE_SAVE,7,2) +1;

/* save current year and month to highlight today date on display */
CUR_YEAR = SUBSTR(DATE_SAVE,1,4);
/* get month part of date. adding 0 strips the leading zero */
CUR_MONTH = SUBSTR(DATE_SAVE,5,2) +0;

YEAR = CUR_YEAR; /* these variables will change with whats displayed */
MONTH = CUR_MONTH;

IF YEAR // 400 &not;= 0 & YEAR // 4 = 0 THEN /* leap year? */
    NUM_OF_DAYS.2 = 29;
ELSE
    NUM_OF_DAYS.2 = 28;

```

```

/* find 1st weekday of the month. Sun = 0 AND Sat = 6 */
FIRST_WEEKDAY = (TOT_DAYS+1) // 7;
FIRST_WEEKDAY_SAVE = FIRST_WEEKDAY;

DISP_CENTER_MON = MONTH_NAME.MONTH; /* center display month name */
CENTER_DAYS. = ''; /* null out all unused month days */
/* starting at the first weekday of the month fill center month */
DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.MONTH + FIRST_WEEKDAY ;
    CENTER_DAYS.I = I - FIRST_WEEKDAY;
END;

```

```

/* set up to fill in the left side month */
IF MONTH = 1 THEN
    LEFT_MONTH = 12;
ELSE
    LEFT_MONTH = MONTH - 1;

DISP_LEFT_MON = MONTH_NAME.LEFT_MONTH; /* left display month name */
FIRST_WEEKDAY = (TOT_DAYS - NUM_OF_DAYS.LEFT_MONTH+1) // 7;
LEFT_DAYS. = '';
DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.LEFT_MONTH + FIRST_WEEKDAY ;
    LEFT_DAYS.I = I - FIRST_WEEKDAY;
END;

```

```

/* set up to fill in the right side month */
FIRST_WEEKDAY = (TOT_DAYS + NUM_OF_DAYS.MONTH +1) // 7;
IF MONTH = 12 THEN
    RIGHT_MONTH = 1;
ELSE
    RIGHT_MONTH = MONTH + 1;
DISP_RIGHT_MON = MONTH_NAME.RIGHT_MONTH; /* right display month name */
RIGHT_DAYS. = '';
DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.RIGHT_MONTH + FIRST_WEEKDAY ;
    RIGHT_DAYS.I = I - FIRST_WEEKDAY;
END;

```

```

CUR_DAY_FIELD = 'CENTER_DAYS.' || (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE)
ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )';

```

```
'PANEL SEND CALENDAR' PATH_NAME ATTR_STRING  
'PANEL RECEIVE CALENDAR'
```

```
DO FOREVER;  
  IF PAN.AID = 'PF3' THEN  
    RETURN;  
  
  IF PAN.AID = 'PF7' THEN /* go back one month request */  
    DO;  
      IF MONTH = 1 THEN /* always keep track of center month */  
        DO;  
          MONTH = 12;  
          YEAR = YEAR - 1;  
          IF YEAR // 400 &not;= 0 & YEAR // 4 = 0 THEN /* leap year? */  
            NUM_OF_DAYS.2 = 29;  
          ELSE  
            NUM_OF_DAYS.2 = 28;  
          END;  
        END;  
      ELSE
```

```
        ELSE  
          MONTH = MONTH - 1;  
          TOT_DAYS = TOT_DAYS - NUM_OF_DAYS.MONTH;  
          DISP_RIGHT_MON = DISP_CENTER_MON;  
          DISP_CENTER_MON = DISP_LEFT_MON;  
          DO I = 1 TO 37;  
            RIGHT_DAYS.I = CENTER_DAYS.I;  
            CENTER_DAYS.I = LEFT_DAYS.I;  
          END;  
        END;
```

```
      IF MONTH = 1 THEN  
        LEFT_MONTH = 12;  
      ELSE  
        LEFT_MONTH = MONTH - 1;  
        FIRST_WEEKDAY = (TOT_DAYS - NUM_OF_DAYS.LEFT_MONTH + 1) // 7;  
        DISP_LEFT_MON = MONTH_NAME.LEFT_MONTH;  
        LEFT_DAYS. = '';  
        DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.LEFT_MONTH + FIRST_WEEKDAY ;  
          LEFT_DAYS.I = I - FIRST_WEEKDAY;  
        END;  
      END; /* if pan.aid = 'pf7' */  
    ELSE
```

```
      IF PAN.AID = 'PF8' THEN /* go forward one month request */  
        DO;  
          TOT_DAYS = TOT_DAYS + NUM_OF_DAYS.MONTH;  
          IF MONTH = 12 THEN /* always keep track of center month */  
            DO;  
              MONTH = 1;  
              YEAR = YEAR + 1;  
              IF YEAR // 400 &not;= 0 & YEAR // 4 = 0 THEN /* leap year? */  
                NUM_OF_DAYS.2 = 29;  
              ELSE  
                NUM_OF_DAYS.2 = 28;  
              END;  
            END;  
          ELSE  
            MONTH = MONTH + 1;  
          END;  
        END;
```

```
      DISP_LEFT_MON = DISP_CENTER_MON;  
      DISP_CENTER_MON = DISP_RIGHT_MON;  
      DO I = 1 TO 37  
        LEFT_DAYS.I = CENTER_DAYS.I; /* shift the months to left */  
        CENTER_DAYS.I = RIGHT_DAYS.I;  
      END;
```

```
      IF MONTH = 12 THEN /* need a new right month */  
        RIGHT_MONTH = 1;  
      ELSE  
        RIGHT_MONTH = MONTH + 1;  
      END;
```

```
      DISP_RIGHT_MON = MONTH_NAME.RIGHT_MONTH;  
      FIRST_WEEKDAY = (TOT_DAYS + NUM_OF_DAYS.MONTH + 1) // 7;  
      RIGHT_DAYS. = '';  
      DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.RIGHT_MONTH + FIRST_WEEKDAY;  
        RIGHT_DAYS.I = I - FIRST_WEEKDAY;
```



```

END;
END; /* if pan.aid = 'pf8' */

/* see if today's date is in any of the three month being displayed */
/* and set it to red. */

ATTR_STRING = ''; /* assume current day not on screen */
IF YEAR = CUR_YEAR THEN
  SELECT;
  WHEN MONTH = CUR_MONTH THEN /* current month in middle */
    DO;
      CUR_DAY_FIELD = 'CENTER_DAYS.' ||,
        (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE);
      ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;
    END;

  WHEN MONTH = CUR_MONTH + 1 THEN /* current month in left display */
    DO;
      CUR_DAY_FIELD = 'LEFT_DAYS.' ||,
        (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE);
      ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;
    END;

  WHEN MONTH = CUR_MONTH - 1 THEN /* current month in right display */
    DO;
      CUR_DAY_FIELD = 'RIGHT_DAYS.' ||,
        (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE);
      ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;
    END;

  OTHERWISE;
END; /* select */

'PANEL SEND CALENDAR' PATH_NAME ATTR_STRING
'PANEL RECEIVE CALENDAR'

END; /* do forever loop */

ERROR:
  SAY 'RETURN CODE ' RC
  SAY 'REA CODE ' PAN.REA
  SAY 'LOC CODE ' PAN.LOC
  EXIT;
EXIT_ROUTINE:
  'PANEL END';
  SENDE;
  EXIT;

***** panel definitions *****
each definition needs to be in a separate RFS file.
*****

```


第 30 章 REXX/CICS コマンド

すべての REXX/CICS コマンドについて詳しいリファレンス情報を用意しています。すべてのコマンドの戻りコード情報は、コマンドの実行後に、特殊 REXX 変数 RC で戻されます。

このセクションのコマンドはすべて、コマンド環境名 REXXCICS で使用できます。これは、デフォルトでもあります。ただし、コマンドの定義方法により、さらに具体的な環境名 (CICS など) を使用できます。別のコマンド環境を使用していたためにコマンド環境をリセットする必要がある場合は、REXX/CICS コマンドを発行する前に次のコマンドを入力してください。

```
ADDRESS REXXCICS
```

REXX/CICS は、以下を除くすべての EXEC CICS コマンドをサポートします。

- システム・プログラミング (SPI) コマンド
- HANDLE CONDITION
- HANDLE AID
- HANDLE ABEND
- IGNORE CONDITION
- PUSH
- POP

REXX/CICS での CICS コマンドの構文については、[アプリケーション開発のリファレンス](#)で説明しています。既存の EXEC CICS コマンド定義と REXX コマンドの間のマッピングは、以下のとおりです。

- CICS コマンドの接頭部には、EXEC CICS ではなく CICS を使用します。
- データ値フィールドはすべて、リテラル文字ストリングとして、あるいは REXX 変数名として指定することができます。
- データ域フィールドはすべて、対象データのソースまたはターゲットのいずれかである REXX 変数名として指定できます。
- CICS コマンドのソース・フィールドとターゲット・フィールドの両方に同じ REXX 変数を使用しないでください。そうした場合、コマンドの実行結果は予測不能になります。
- LENGTH オプションを指定しない場合、長さは、関連する REXX 変数または文字ストリングの長さから自動的に決定されます。
- REXX/CICS から CICS ENQ コマンドを使用する場合は、LENGTH パラメーターを使用してください。そうしないと、予測不能な結果になる可能性があります。
- NOHANDLE は、すべての CICS コマンドについて自動的に指定されます。各コマンドの実行からの EIBRESP 値は、REXX 特殊変数 RC で戻されます。また、EIB フィールドは、REXX 変数の DFHEIBLK、EIBRESP、EIBRESP2、および EIBRCODE に入れられます。
- 戻りコード値の説明については、[アプリケーション開発のリファレンス](#)を参照してください。負の値のある戻りコードについては、[423 ページの『第 32 章 戻りコード』](#)を参照してください。

EXEC CICS コマンドから REXX/CICS コマンドへのマッピング例

- REXX 以外:

```
EXEC CICS XCTL PROGRAM('PGMA') COMMAREA(COMA) LENGTH(COMAL)
```

- REXX/CICS:

```
"CICS XCTL PROGRAM('PGMA') COMMAREA(COMA)"
```

注: EXEC CICS READ コマンド、WRITE コマンド、および DELETE コマンドは、それぞれの使用を制御するために、デフォルトで、REXX/CICS 許可コマンドとしてインプリメントされます。[REXX/CICS 許可コマンドと許可コマンド・オプションおよびセキュリティ](#)を参照してください。

ALLOC

ALLOC は、データ・セットをデータ定義名と関連付けます。

これは、許可コマンドです。

➡ ALLOC — *ddname* — *dsname* — (— *member* —) — SHR
OLD ➡

オペランド

DD 名

データ定義名を指定します。

dsname

完全修飾データ・セット名を指定します。

member

データ・セット内のメンバー名を指定します。

SHR

データ・セットが存在しますが、排他制御は必要ありません。

OLD

データ・セットが存在し、排他制御が必要です。

戻りコード

SVC 99 によって与えられた戻りコード

詳しくは、[z/OS MVS Programming: Authorized Assembler Services Guide](#) を参照してください。

1702

無効なオペランド

注:

- セキュリティー上の理由から、DS 名にはユーザー ID が上位接頭部として含まれます。また、データ定義名およびデータ・セット名のセキュリティ処理を追加できるようにユーザー出口が用意されています。ALLOC コマンドからメンバー名を省略した場合は、順次データ・セットが割り振られます。パフォーマンス上の理由から、データ・セットがマイグレーションされた場合、割り振り要求は拒否されます。
- インストールでセキュリティ検査を行えるように、割り振りサンプル出口 CICSECX1 が、このコマンドと共に用意されています。

AUTHUSER

AUTHUSER はユーザー ID のリストを許可します。

これは、許可コマンドです。

➡ AUTHUSER — *userid* ➡

オペランド

userid

REXX/CICS 許可となる、CICS サインオン・ユーザー ID です。

戻りコード

0

Normal return

2602

オペランドが無効であるか、または欠落しています

2621

ユーザー ID の無効な長さを指定します

2642

ユーザー ID の格納中にエラーが発生した

例

```
'AUTHUSER USER2 SYSPGMR'
```

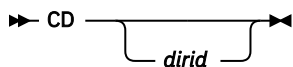
この例は、USER2 および SYSPGMR の REXX/CICS 許可ユーザーを作成します。

注:

1. 許可ユーザーであれば、許可 REXX/CICS ライブラリーからロードされた EXEC で実行しているかどうかに関係なく、REXX/CICS 許可コマンドを使用できます。
2. ユーザー ID のリストでエラーが検出されると、エラーのあるユーザー ID が REXX 特殊変数に置かれ、リストの RESULT および処理は停止します。
3. AUTHUSER コマンドは累積的です。CICS 領域がリサイクルされるまで、過去の AUTHUSER コマンド定義は有効なままです。

CD

CD は、RFS ファイル・システム・ディレクトリーを変更します。



オペランド

dirid

ユーザーの新しい現行作業ディレクトリーになる、REXX ファイル・システム・ディレクトリーの一部または全体を指定します。

dirid が指定されない場合、現行作業ディレクトリーが変更されるのではなく、現行作業ディレクトリーは取得され、REXX 特殊変数 RESULT に入られます。

完全ディレクトリー ID の形式は、次のとおりです。 *poolid*:¥*dirid1*¥...¥*diridn*

完全ディレクトリー ID が指定されると、それが、以前のディレクトリー設定に完全にとって代わります。

部分ディレクトリー ID は *poolid* では始まりません。この場合、部分ディレクトリー ID が既存のディレクトリー ID の末尾に付加されます。部分ディレクトリー ID が 2 つのピリオドで始まる場合、1 つのディレクトリー・レベルが除去 (右から) されてから、新しい部分ディレクトリー ID が末尾に付加されます。この場合、ディレクトリー ID の前に円記号 (¥) が必要です。

例: 現行ディレクトリーが *POOL1:¥USERS¥USER1¥ABC* で、*CD ..¥XYZ* と入力すると、新しい現行ディレクトリーは *POOL1:¥USERS¥USER1¥XYZ* になります。

ユーザーのデフォルトのディレクトリー ID は、*poolid*:¥*USERS¥userid*¥ です。ここで、*poolid* は最初の定義済み RFS ファイル・プールのファイル・プール ID であり、*userid* はユーザーの CICS サインオン・ユーザー ID です。ユーザーが CICS にサインオンしていない場合には、*userid* は、デフォルトで CICS DFLTUSER 内の値になります。

戻りコード

0

Normal return

521

ファイル・プール定義の取得中にエラーが発生した

522

デフォルトの RFS ディレクトリーの作成中にエラーが発生した

523

現行 RFS ディレクトリー情報の格納中にエラーが発生した

524

RFS ディレクトリーが存在しないか、アクセス権限が許可されていない

525

ディレクトリー情報の取得中にエラーが発生した

526

ファイル・プール/ディレクトリーが無効

527

過去のルート・ディレクトリーに戻ることはできない

528

結果値の設定中にエラーが発生した

例

```
'CD ¥USERS¥USER2¥XYZ'
```

この例は、現行作業ディレクトリーを、以前のディレクトリー設定に関係なく、現在使用しているファイル・プール内の ¥USERS¥USER2¥XYZ に変更します。

ユーザーの現行ディレクトリーが ¥USERS¥USER2 の場合は、次のように入力します。

```
'CD XYZ'
```

現行ディレクトリーが ¥USERS¥USER2¥XYZ に移動します。

注: CD コマンドは、REXX EXEC の実行のために検索順序を識別する PATH コマンドと連携して動作します。現行ディレクトリー (CD コマンドで指定) は、常に最初に EXEC について検索されます。次に、PATH コマンドでリストされたディレクトリーと MVS 区分データ・セットが検索されます。最後に、CICS 領域の始動 JCL で割り振られた MVS 区分データ・セットが検索されます。

CEDA

リソース定義オンライン (RDO) のために CEDA コマンドを実行します。

➡ CEDA — *RDO_Command* ➡

オペランド

RDO_Command

入力として CEDA トランザクション・プログラムに渡されるコマンド・ストリングを指定します。

戻りコード

n

エラーが検出された場合に CICS によって戻される戻りコード。

0

Normal return

-101

無効なコマンド

警告メッセージまたはエラー・メッセージはすべて、変数 CEDATOUT に入れます。実行の結果は、存在する場合、変数 CEDAEOUT に入れます。CEDAEOUT で戻される最大長はおよそ 28K バイトです。各変数の形式は、以下のとおりです。

- フィールドの包括的長さを含むバイナリー・ハーフワード。
- 生成されたメッセージの数を含むバイナリー・ハーフワード。
- 最高のメッセージの重大度を含むバイナリー・ハーフワード。0 および 4 は実行を続行し、8 および 12 は実行を続行しません。
- 以下を含む可変長データ
 - CEDATOUT の場合: 診断メッセージ
 - CEDAEOUT の場合: 通常、CEDA 画面に表示されるデータ (メッセージを含む)。各行の先頭は改行 (NL) 文字です。そうでない場合は、ブランクと大文字の英数字で構成されます。

このデータの形式は、リリースが異なる場合は保証されませんが、CEDA によって表示されるものと同じです。

例

```
'CEDA INSTALL PROGRAM(XYZ) GROUP(ABC)'
```

この例は、実行のために CEDA トランザクション・プログラムに渡される CICS コマンドを示します。

CEMT

CEMT を使用すると、REXX から CICS マスター端末コマンドを実行できます。

➡ CEMT — *master_term_cmd* ➡

オペランド

master_term_cmd

入力として CEMT トランザクション・プログラムに渡されるコマンド・ストリングを指定します。

戻りコード

n

エラーが検出された場合に CICS によって戻される戻りコード。REXX からの CEMT の呼び出しでは、DFHEMTA (CEMT プログラマブル・インターフェース) が使用されるので、戻されるコードは、CEMT コマンドの戻りコードです。考えられる戻りコードは以下のとおりです。

- 1 NOT FOUND
- 2 CLASS NOT FOUND
- 3 ERROR
- 4 BEGINS WITH DFH
- 5 CHANGE INVALID
- 6 CANNOT NEWCOPY

- 7** NOT AUTHORIZED
- 8** OPTION CONFLICT
- 9** PRIORITY > 255
- 10** NOT FOR CONSOLE
- 11** PROGRAM NOT FOUND
- 12** INVALID AUTHID
- 13** INVALID ATI / TTI
- 14** IS NOT INTRA
- 15** OPEN/SWITCH FAIL
- 16** SDUMP BUSY
- 17** NOT SUCCESSFUL
- 18** 0>=TRIGGER>32767
- 19** NOT FOR INDIRECT
- 20** 0>MAXIMUM>999
- 21** IS NOT EXTRA
- 22** OPEN/CLOSE FAILED
- 23** SDUMP SUPPRESSED
- 24** BEGINS WITH C
- 25** NOT ACTIVE
- 26** NOT CLOSED
- 27** NOT FOR SYS LOG
- 28** ALREADY EXISTS
- 29** CATALOG I/O ERROR
- 30** 1>MAXTASKS>2000
- 31** NOT FOR REMOTE

32 NOSTG DSALIMIT
33 0>AGING>65535
34 AKP NOT IN SYSTEM
35 MAXT. < AMAXT.
36 NOT IN SYSTEM
37 INVALID COMAUTHID
38 50>AKP>65535
39 CATALOG FULL
40 2M>DSALIMIT>16M
41 1>MAXACTIVE>999
42 INVALID DUMPCODE
43 INVALID DB2ID
44 500>RUN. >2700000
45 100>TIME>3600000
46 BAD TRANSAC CLASS
47 TIME < SCANDELAY
48 CEILING REACHED
49 1>MROBATCH>255
50 48M>EDSALIM>2047M
51 CLOSE FAILED
52 NOT FOR BDAM
53 CLOCK INOPERATIVE
54 NOT VALID VTAM®
55 NOT FOR PATH

56	OPEN FILE
57	BEING CLOSED
58	BEING IMMCLOSED
59	0>SCANDELAY>5000
60	NOT FOR SNASVCMG
61	NOT FOR THIS TASK
62	NOT FOR YOUR TERM
63	INVALID MSGQUEUE
64	NOT FOR YOUR LINE
65	QUEUE IS DISABLED
66	NOSTG EDSALIMIT
67	PARTIAL DUMP
68	DDNAME NOT FOUND
69	BEING ACQUIRED
70	BEING FORCECLOSED
71	NOT FOR HOLD PROG
72	LOAD FAILED
73	SDUMP FAILED
74	EMPTY OR NOT CLSD
75	NOT DISABLED
76	CLOSE REQUESTED
77	BEING OPENED
78	BEING UNENABLED
79	BEING DISABLED
80	BEING QUIESCED

- 81**
SEE MSG DFHIR3793
- 82**
START/SWITCH FAIL
- 83**
INVALID PLAN
- 84**
INVALID INTERVAL
- 85**
OUT &→REL INVALID
- 86**
SEE MSG DFHIR3768
- 87**
SEE MSG DFHIR3786
- 88**
NOT FOR MAPSET
- 89**
SEE MSG DFHIR3771
- 90**
NOT FOR PARTITION
- 91**
SEE MSG DFHIR3773
- 92**
INV DSRTPROGRAM
- 93**
SEE MSG DFHIR3775
- 94**
SEE MSG DFHIR3776
- 95**
SEE MSG DFHIR3777
- 96**
SEE MSG DFHIR3778
- 97**
SEE MSG DFHIR3779
- 98**
SEE MSG DFHIR3780
- 99**
SEE MSG DFHIR3781
- 100**
SEE MSG DFHIR3791
- 101**
ONLY FOR VTAM
- 102**
GOING OUT
- 103**
SPECIFY NUMBER
- 104**
NUMBER ERROR
- 105**
NEGPOLL INVALID

106
>20000

107
INVALID ENDOFDAY

108
MAX | SHUTDOWN

109
LINE DCB NOT OPEN

110
INV PLANEXITNAME

111
SET FAILED

112
REMOVE FAILED

113
INVALID SIGNID

114
FILECOUNT > 0

115
INV STATSQUEUE

116
BACKOUT FAILED

117
INV COMTHREADLIM

118
>MAXIMUM

119
INV PURGECYCLE

120
IS INDOUBT

121
4>TCBLIMIT>2000

122
CONNECTION →ACQD

123
DATASET QUIESCING

124
IN PROGRESS

125
DATASET UNAVAIL

126
DATASET QUIESCED

127
BACKUP OCCURRING

128
RESOURCE MISSING

129
STATS MISSING

130
IS SIT PARAMETER

131
CANNOT LOAD PLT

132
INV THREADLIMIT

133
NO DATASET

134
RECOVERY REQUIRED

135
PURGE FAILED

136
FILE IN USE

137
ONLY FOR BDAM

138
STOPPED

139
SEE MSG DFHIR3798

140
PROGRAM IS URM

141
IN USE

142
IN USE BY ICE

143
IN USE BY PCT

144
NOT RELEASED

145
DELETE FAILED

146
INVALID RECOVSTAT

147
NOT OUTSERVICE

148
INV PROTECTNUM

149
INVALID DB2ENTRY

150
IS ENABLED EXIT

151
INV DTRPROGRAM

152
IN USE BY AID

153
NOT FOR SESSION

154
NOT FOR PIPELINE

155
NOT DISCARDABLE

- 156**
NOT LOCAL SYSTEM
- 157**
NOT FOR SYSTEM
- 158**
NOT FOR MODEL
- 159**
NO ACTION
- 160**
CANNOT LOAD XLT
- 161**
ISC NOT DEFINED
- 162**
ALREADY ACTIVE
- 163**
INVALID TRANSID
- 164**
DUPLICATE TRANSID
- 165**
NO BWO SUPPORT
- 166**
DSNB IS INVALID
- 167**
DSNB BDAM OR PATH
- 168**
UPDATE COUNT > 0
- 169**
DSN → SMS MANAGED
- 170**
BWO ERROR
- 171**
DFHTMP ERROR
- 172**
PRESET SIGNON ERR
- 173**
NOT FOR CPSVCMG
- 174**
PRM UNAVAILABLE
- 175**
NOT WHEN XLN DONE
- 176**
INV PROGAUTOINST
- 177**
INV PROGAUTOCTLG
- 178**
INV PROGAUTOEXIT
- 179**
FREE NOT COMPLETE
- 180**
BEING RELEASED

181
REUSE DEFINED

182
UNBLOCKED DEFINED

183
0< MAX. <99999999

184
FORMAT NOT VARBLE

185
NO LSRPOOL

186
CONNECTING

187
INVALID FREQUENCY

188
0>PURGET.>1000000

189
INVALID PSDINT

190
NOT WITH XRF

191
SETLOGON FAILURE

192
BACK LEVEL VTAM

193
ACB CLOSED

194
RECOVERY ERROR

195
DEFERRED

196
NOT FOR LOCAL SYS

197
SEE MSG DFHIR3799

198
ONLY FOR APPC

199
ESM INACTIVE

200
REGISTER ERROR

201
DEREGISTER ERROR

202
AIDS CANCELED

203
WAITING FOR DB2

204
DB2 INACTIVE

205
INVALID INITPARM

206
NOT REQUIRED

207
STILL CLOSING

208
NO AIDS CANCELED

209
DFSMS CATLG ERROR

210
DSNB REMOVED

211
NO RLS SUPPORT

212
SYSTEM LOCKED

213
SDTRAN NOT FOUND

214
SDTRAN DISABLED

215
SDTRAN SHUT DIS

216
DSN → DFSMS VSAM

217
DATASET MIGRATED

218
INVALID IDLE

219
NOT LU61 OR LU62

220
NETID 0 USE PRFRM

221
SEE MSG DFHZC0178

222
NOT GR REGISTERED

223
ENTER NETID

224
NO AFFINITY FOUND

225
SESSIONS IN USE

226
SEE MSG DFHZC0176

227
DELETE INFLIGHT

228
USED BY INDIRECTS

229
IRC IS OPEN

230
IS ERROR CONSOLE

231
SDTRAN IS REMOTE

232
XRF NOT ACTIVE

233
SYSID IN ERROR

234
ERR: SHUNTED UOWS

235
SEE MSG DFHZC0173

236
RLS AND CMT

237
DISCONNECTING

238
KEYLENGTH ERROR

239
RECORDSIZE ERROR

240
MISSING POOL NAME

241
INVALID NAME

242
POOL NOT FOUND

243
CONTEN AND RECOV

244
INVALID ACTION

245
LASTUSED<INTERVAL

246
WAITING

247
NO AUDITLOG

248
PARAM MISMATCH

249
NO CFDT SERVER

250
TCPIP CLOSED

251
PORT IN USE

252
PORT NOT AUTH

253
INVALID STATUS

254
PROFILE NOT FOUND

255
ADDRESS UNKNOWN

256
INVALID Q-TYPE

257
1>MAXOPENTCBS>2000

258
NO JVMCLASS SET

259
NOT A JVM PROGRAM

260
INVALID JVMCLASS

261
INVALID DSNAME

262
1>MAXSOCKETS>65535

263
EXCEEDS HARD LIMIT

264
AT MAXSOCKETS

265
TCIPSERVICE NOT OPENED

266
SESSBEANTIME > 143999

267
RESOURCE NOT INSERVICE

268
MISSING CORBASERVER NAME

269
DJAR IS PENDING RESOLUTION

270
INVALID DB2GROUPID

271
DB2ID & GROUPID ENTERED

272
1>MAXJVMTCBS>999

273
1>MAXXPTCBS>999

274
NOT IIOPLISTENER

275
DSNAPRH NOT FOUND

276
MAXOPENTCBS < DB2CONN TCBLIMIT

277
TCBLIMIT > MAXOPENTCBS

278
DB2 GROUPID NOT FOUND

279
DB2 ID NOT FOUND

280
DJAR CLASH (CORBASERVER SCAN)

281
JVMPROFILE INVALID..SET PROGRAM

282
BEING STARTED

283
BEING RELOADED

284
BEING ENABLED

285
BEING DISCARDED

286
NOT STARTED

287
NOT STOPPED

288
DB2 RESTART-LIGHT

289
CACHESIZE INVALID

290
TCPIP INACTIVE

291
ATTEMPT FORCE PURGE

292
1>MAXSSLTCBS>1024

293
PIPELINE NOT ENABLED

294
MISSING JVMPROFILE

295
NOT VALID FOR DFHRPL

296
RANKING 10 RESERVED FOR DFHRPL

297
RANKING OUT OF RANGE

298
NO ACQ FOR SEND=0

299
MAXJVMTCBS EXCEEDED

300
PSTYPE=NOPS AND PSDI > 0

301
INV FILELIMIT

302
INV PROGRAMLIMIT

303
INV TSQUEUELIMIT

304
MQNAME IN USE

305
INVALID MQNAME

- 306**
MQNAME NOT FOUND
- 307**
WAITING FOR QMGR
- 308**
1>THREADLIMIT>256
- 309**
NO MORE THREADS
- 310**
THREADS LIMITED
- 311**
DRAINING
- 312**
SEE MSG DFHPI2024
- 313**
EXCESSIVE NUMBER OF ELEMENTS
- 314**
1M>TSMALIM>32G
- 315**
ATTEMPT PURGE 1ST
- 316**
USING JVMSERVER
- 317**
INV REUSELIMIT
- 318**
> 25.00% OF MEMLIMIT
- 319**
DEFINED BY BUNDLE
- 320**
SEE MSG DFHSO0123
- 321**
NOT FOR JVM PROG
- 322**
USED BY MGMTPART

注：CICS リリースによって、戻されない戻りコードもあります。

0

Normal return

-102

無効なコマンド

警告メッセージまたはエラー・メッセージはすべて、変数 CEMTTOUT に入れます。実行の結果は、存在する場合、変数 CEDAEOUT に入れます。CEMTEOUT で戻される最大長はおよそ 28K バイトです。各変数の形式は、以下のとおりです。

- フィールドの包括的長さを含むバイナリー・ハーフワード。
- 生成されたメッセージの数を含むバイナリー・ハーフワード。
- 最高のメッセージの重大度を含むバイナリー・ハーフワード。0 および 4 は実行を続行し、8 および 12 は実行を続行しません。
- 以下を含む可変長データ
 - CEMTTOUT の場合: 診断メッセージ

- CEMTEOUT の場合: 通常、CEMT 画面に表示されるデータ (メッセージを含む)。各行の先頭は改行 (NL) 文字です。そうでない場合は、ブランクと大文字の英数字で構成されます。

このデータの形式は、リリースが異なる場合は保証されませんが、CEMT によって表示されるものと同じです。

CEMTEOUT の内容を解析するために以下の例を使用することができます。

```
PARSE VAR CEMTEOUT BUFFLEN 3 MSGCOUNT 5 MAXRC REST
```

この例では、以下のように割り当てられます。

- 1 目目のハーフワード (CEMTEOUT の合計長を含む) が REXX 変数 BUFFLEN に
- 2 目目のハーフワード (生成されたメッセージ数を含む) が REXX 変数 MSGCOUNT に
- 3 目目のハーフワード (最高のメッセージ重大度を含む) が REXX 変数 MAXRC に
- CEMTEOUT の残りが REXX 変数 REST に

最初の 3 つのハーフワードに含まれる値は、印刷不能な 2 進数フィールドなので、通常はピリオドとして表示されます。これらの値を表示するには、初めに変換しなければなりません。例えば、BUFFLEN の値を表示するには、以下のいずれかを使用します。

- SAY C2X(BUFFLEN)
- VAR1 = C2X(BUFFLEN)
SAY VAR1

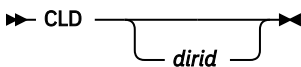
例

```
'CEMT SET PROGRAM(XYZ) NEWCOPY'
```

この例では、CICS コマンドを CEMT トランザクション・プログラムに渡して実行しています。

CLD

CLD は、ユーザーの現行 RLS リスト・ディレクトリーを変更します。



オペランド

dirid

ユーザーの新しい現行作業ディレクトリーになる、REXX List System ディレクトリーの一部または全体を指定します。

dirid が指定されない場合、現行作業ディレクトリーが変更されるのではなく、現行作業ディレクトリーは取得され、REXX 変数 RESULT に入れられます。

完全ディレクトリー ID は、円記号で始まり、形式は次のとおりです。¥*dirid1*¥...¥*diridn*

完全ディレクトリー ID が指定されると、それが、以前のディレクトリー設定に完全に取って代わります。

部分ディレクトリー ID は円記号では始まりません。この場合、部分ディレクトリー ID が既存のディレクトリー ID の末尾に付加されます。部分ディレクトリー ID が 2 つのピリオドで始まる場合、1 つのディレクトリー・レベルが除去 (右から) されてから、新しい部分ディレクトリー ID が末尾に付加されます。この場合、ディレクトリー ID の前に円記号 (¥) が必要です。

例: 現行ディレクトリーが ¥USERS¥USER1¥ABC で、CLD ..¥XYZ と入力すると、新しい現行ディレクトリーは ¥USERS¥USER1¥XYZ になります。

ユーザーのデフォルト・ディレクトリー ID は ¥USERS¥*genid*¥ です。ここで、*genid* はユーザーの CICS サインオン・ユーザー ID です。ユーザーが CICS にサインオンしていない場合には、*genid* は、デフォルトで DFLTUSER 内の値になります。

戻りコード

0

Normal return

923

現行 RLS ディレクトリー情報の格納中にエラーが発生した

924

RLS ディレクトリーが存在しないか、アクセス権限が許可されていない

925

ディレクトリー情報の取得中にエラーが発生した

926

無効なディレクトリー

927

過去のルート・ディレクトリーに戻ることはできない

928

結果値の設定中にエラーが発生した

例

```
'CLD ¥USERS¥USER2¥XYZ'
```

この例は、以前のディレクトリー設定に関係なく、現行作業リスト・ディレクトリーを ¥USERS¥USER2¥XYZ に変更します。

ユーザーの現行ディレクトリーが ¥USERS¥USER2 の場合は、次のように入力します。

```
'CLD XYZ'
```

現行ディレクトリーが ¥USERS¥USER2¥XYZ に移動します。

注：

1. 現行ディレクトリー (CLD コマンドにより指定) は、常に最初に検索され、適切なリスト名を持つ RLS リストを探し出そうとします。
2. 完全修飾 RLS ファイル名は、ユーザーのディレクトリーの検索を迂回します。

CONVTMAP

CONVTMAP は、MVS 順次ファイルまたは PDS メンバーを読み取り、DSECT (以前にアセンブルされた BMS マップによって作成されたもの) を 1 つの構造に変換し、その結果を REXX ファイル・システムのファイルに格納します。

➡ CONVTMAP — *mvs_dataset_name* — *rfs_fileid* →

CONVTMAP への入力として使用される BMS マップは、アセンブラー言語形式でなければなりません。結果として生じる出力ファイルは、REXX ファイル構造として形式設定されます。[473 ページの『第 36 章 基本マッピング・サポートの例』](#)を参照してください。

オペランド

mvs_dataset_name

完全修飾 MVS データ・セット名を指定します。データ・セットは物理順次データ・セットでなければなりません。PDS の場合は、括弧で囲んで PDS メンバーを指定しなければなりません。

rfs_fileid

完全修飾 REXX ファイル・システム・ファイル、つまり REXX ファイル名のみを指定します。完全修飾名が指定されない場合、ファイルを格納するために現在の REXX ディレクトリーが使用されます。

戻りコード

n

MVS データ・セットの処理試行からの戻りコード。

0

Normal return

-302

無効なオペランド

-321

無効な入力レコード

-322

出力ファイルの書き込み中の RFS エラー

例

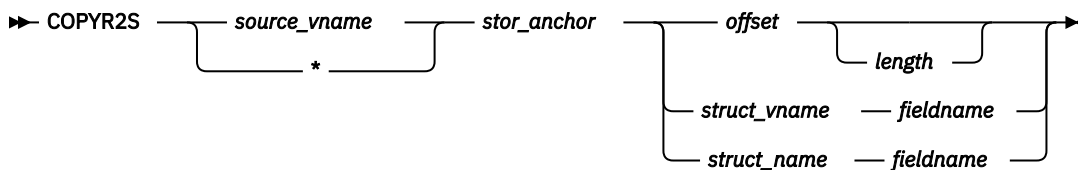
```
'CONVTMAP USER1.TEST.DATA(MAP1) POOL1:\USERS\USER1\MAP1.DATA'
```

この例は、入力 BMS マップ DSECT である MAP1 を示します。これは、形式設定され、RFS ファイルである POOL1:¥USERS¥USER1¥MAP1.DATA に書き込まれる、MVS PDS データ・セット USER1.TEST.DATA のメンバーです。

COPYR2S

COPYR2S は、REXX 変数の内容を、GETMAIN 要求で事前に取得した 31 ビットのストレージにコピーします。

これは、許可コマンドです。



stor_anchor でアドレス指定した対象ストレージ域が 32767 バイトを超えている場合は、GETMAIN 要求で FLENGTH オプションを指定していただく確認してください。

注：COPYR2S は 31 ビット・アドレスに対して実行できます。COPYR2S は 64 ビット・アドレスに対しては実行できません。

オペランド

source_vname

事前に GETMAIN 要求で取得したストレージ域にコピーされる値が含まれている REXX 変数を指定します。

注：この値は、置換が行われないように、引用符で囲む必要があります。

すべての REXX 変数がコピーされることを指定します。アスタリスク (*) を指定した場合、*fieldname* を指定することはできません。

stor_anchor

事前に GETMAIN 要求で取得したターゲット・ストレージ域のアンカーが含まれている REXX 変数を指定します。このアンカーは 4 バイトで構成され、GETMAIN 要求で取得したストレージの 31 ビット・

アドレスが含まれているものです。アンカーが 31 ビット・アドレスを指すことを確認してください。そうでない場合は、予測不能な結果になる可能性があります。

offset

REXX 変数の内容のコピー先である (GETMAIN 要求で取得した) ストレージ域への変位を指定します。この領域の最初のバイトは、ゼロの変位で示されます。

長さ

実行されるコピーの長さを 10 進のバイトで指定します。この長さが指定されると、ソース REXX 変数の内容は、切り捨てられるか、またはこの長さに一致するようブランクが埋め込まれてから、コピーされます。ただし、ソース REXX 変数がこのプロセスで変更されることはありません。この長さを省略すると、ソース REXX 変数の現在の長さが使用されます。

struct_vname

GETMAIN 要求で取得したストレージ域内のフィールドの構造定義 (またはマッピング) が含まれている REXX 変数を指定します。この変数内でのデータの形式は、*field1_name length ... fieldn_name length* です。この機能は、フィールド変位が、中央の位置から容易に計算および変更できるように提供されています。

struct_name

GETMAIN 要求で取得したストレージ域内のフィールドの構造定義 (またはマッピング) が含まれている構造ファイル ID を指定します。

構造は、*fieldname location length type* という形式のレコードで構成されます。ここで、それぞれの意味は以下のとおりです。

fieldname

フィールドの、1 文字から 12 文字のシンボル名を指定します。

location

このフィールドが始まる構造内の位置を指定します (最初の位置は 1)。

長さ

このフィールドの 10 進数の長さをバイト数で指定します。

type

フィールド・データ・タイプを、C (文字)、F (フルワード)、または H (ハーフワード) で指定します。

fieldname

このコピーの宛先フィールドと関連付けられた、1 文字から 12 文字のシンボル名です。この名前が、上記の指定 REXX 変数または構造定義ファイルに存在している必要があります。*fieldname* の指定が必要なのは、*struct_vname* または *struct_name* を指定する場合です。

戻りコード

0

Normal return

2002

無効なオペランド

2021

無効な構造定義

2022

無効な変数構造定義

2023

フィールド名が見つかりません

2025

GETVAR 要求の処理に失敗しました

2026

無効な数値入力

2027

RFS 読み取りエラー

2028

無効なオフセット

2029

無効な長さ値

例

次の例は、200 バイトの仮想記憶域を要求し、16 進値 '00000000'x をその領域の 4 バイトから 7 バイトにコピーします。

```
/* Needed if entering example from the REXXTRY utility */
'PSEUDO OFF'
'CICS GETMAIN SET(WORKANC) LENGTH(200)' /* get 200 bytes of working storage */
VAR1 = '00000000'x /* set a REXX variable with 4 bytes of hex */
'COPYR2S VAR1 WORKANC 4'
```

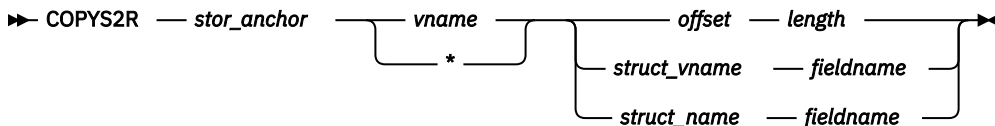
次の例は、200 バイトの仮想記憶域を要求し、GETMAIN 要求で取得した、アンカー WORKANC で参照される領域の位置 7 に文字ストリング ABC をコピーします。

```
'CICS GETMAIN SET(WORKANC) LENGTH(200)' /* get 200 bytes of working storage */
VAR1 = 'ABC' /* set a REXX variable with 3 characters */
struct1 = 'flda 4 flldb 2 fldc 3 fldd 8 flde 5'
'COPYR2S VAR1 WORKANC STRUCT1 FLDC'
```

COPYS2R

COPYS2R は、GETMAIN 要求で事前に取得した 31 ビットのストレージのデータを、REXX 変数にコピーします。

これは、許可コマンドです。



stor_anchor でアドレス指定した対象ストレージ域が 32767 バイトを超えている場合は、GETMAIN 要求で FLENGTH オプションを指定していたか確認してください。

注：COPYS2R は 31 ビット・アドレスに対して実行できます。COPYS2R は 64 ビット・アドレスに対しては実行できません。

オペランド

stor_anchor

事前に GETMAIN 要求で取得したターゲット・ストレージ域のアンカーが含まれている REXX 変数を指定します。このアンカーは 4 バイトで構成され、事前に GETMAIN 要求で取得した 31 ビット・ストレージのアドレスが含まれているものです。アンカーが 31 ビット・アドレスを指すことを確認してください。そうでない場合は、予測不能な結果になる可能性があります。

vname

事前に GETMAIN 要求で取得したストレージ域から値をコピーする REXX 変数を指定します。

注：この値は、置換が行われないように、引用符で囲む必要があります。

*

すべての REXX 変数がコピーされることを指定します。アスタリスク (*) を指定した場合、fieldname を指定することはできません。

offset

REXX 変数の内容のコピー元である (GETMAIN 要求で取得した) ストレージ域への変位を指定します。この領域の最初のバイトは、ゼロの変位で示されます。

長さ

実行されるコピーの長さを 10 進のバイトで指定します。この長さが指定されると、ソース REXX 変数の内容は、切り捨てられるか、またはこの長さに一致するようブランクが埋め込まれてから、コピーされます。ただし、ソース REXX 変数がこのプロセスで変更されることはありません。この長さを省略すると、ソース REXX 変数の現在の長さが使用されます。

struct_vname

GETMAIN 要求で取得したストレージ域内のフィールドの構造定義 (またはマッピング) が含まれている REXX 変数を指定します。この変数内でのデータの形式は、*field1_name length ... fieldn_name length* です。この機能は、フィールド変位が、中央の位置から容易に計算および変更できるように提供されています。

struct_name

GETMAIN 要求で取得したストレージ域内のフィールドの構造定義 (またはマッピング) が含まれている構造ファイル ID を指定します。

構造は、*fieldname location length type* という形式のレコードで構成されます。ここで、それぞれの意味は以下のとおりです。

fieldname

フィールドの、1 文字から 12 文字のシンボル名を指定します。

location

このフィールドが始まる構造内の位置を指定します (最初の位置は 1)。

長さ

このフィールドの 10 進数の長さをバイト数で指定します。

type

フィールド・データ・タイプを、C (文字)、F (フルワード)、または H (ハーフワード) で指定します。

fieldname

このコピーの宛先フィールドと関連付けられた、1 文字から 12 文字のシンボル名です。この名前が、上記の指定 REXX 変数または構造定義ファイルに存在する必要があります。*fieldname* の指定が必要なのは、*struct_vname* または *struct_name* を指定する場合です。

戻りコード

0

Normal return

2102

無効なオペランド

2121

無効な構造定義

2122

無効な変数構造定義

2123

フィールド名が見つかりません

2125

GETVAR 要求の処理に失敗しました

2126

無効な数値入力

2127

RFS 読み取りエラー

2128

無効なオフセット

2129

無効な長さ値

例

```
var1 = '' /* set REXX variable VAR1 to null */
struct1 = 'flda 4 fldb 2 fldc 3 fldd 8 flde 5'
'COPYS2R WORKANC VAR1 STRUCT1 FLDC'
```

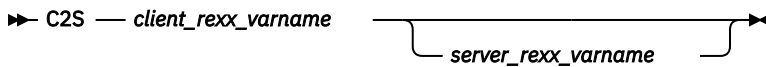
この例は、フルワード・アドレスによって固定された、以前に GETMAIN の対象となったストレージ域の位置 7 から位置 9 の 3 バイトのデータを REXX 変数 WORKANC にコピーし、それを REXX 変数 VAR1 へコピーします。

注:

1. アンカー・アドレスは、GETMAIN コマンドによって設定されるものに限られません。例えば、COPYS2R を使用して、GETMAIN 要求で取得した領域のフルワード・アドレスを、後続の COPYS2R または COPYR2S でアンカーとして使用する REXX 変数にコピーすることができます。
2. GETMAIN 要求で取得した 1 つの領域に複数の構造 (フィールド) 定義を指定できます。複数の定義をオーバーラップし、ネストして使用して、フィールドを再定義することができます。
3. アンカー・アドレスは、GETMAIN 要求で取得した領域の始めを示すものでなくてもかまいません。しかし、GETMAIN 要求で取得した、ユーザーが所有する領域内にあり、アンカー・アドレスに対する操作が領域の境界を超えないようにすることが重要です。

C2S

C2S は、クライアント REXX 変数をサーバー REXX 変数にコピーします。



オペランド

client_rexx_varname

コピー元にするクライアント REXX 変数を指定します。

server_rexx_varname

コピー先にするサーバー REXX 変数を指定するオプションの名前です。これが指定されない場合、デフォルトは、*client_rexx_varname* と同じになります。

戻りコード

0

Normal return

2440

変数名が指定されていません

2441

変数の取得中にエラーが発生しました

2442

変数の格納中にエラーが発生しました

2448

使用可能なクライアントがありません

例

```
'C2S VARA VARB'
```

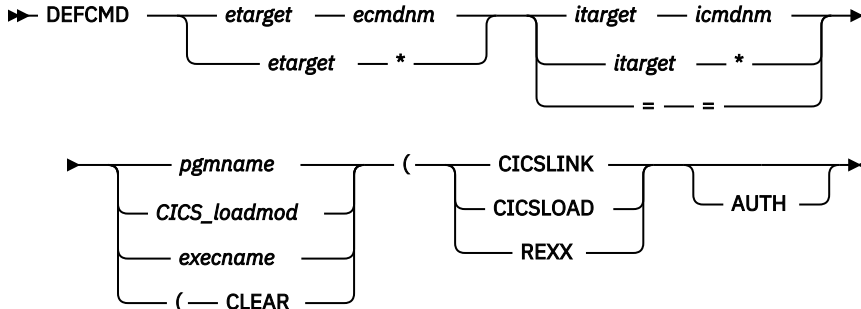
この例は、クライアント REXX 変数 VARA の内容がサーバー REXX 変数 VARB にコピーされることを示します。VARB の長さは、VARA の長さと同じです。

注:

1. このコマンドに対してサポートされる変数名の最大長は 250 文字です。
2. このコマンドは、REXX/CICS サーバー EXEC (例えば、DEFECMD または DEFSCMD によって定義された EXEC) のみが使用することを意図しています。

DEFECMD

DEFECMD は、REXX ユーザー・コマンドを定義 (または再定義) します。



オペランド

etarget

このコマンドを発行する REXX EXEC で使用された外部ターゲット環境の、1 文字から 8 文字の名前です。これは、コマンド・ストリングの送信先として使用された外部環境名です。この環境名は、テーブル内でルックアップされ、コマンド名と一緒に使用されて、処理のためにコマンド・ストリングが送信される REXX プログラムを決定します。

注: 外部ターゲットは、ADDRESS キーワード命令の環境名と同じものであってよく、また、REXXCICS が現行環境 (これがデフォルト条件) であれば、コマンド・ストリングの最初のトークンとして指定することができます。

ecmdnm

このコマンドを発行するのにユーザーが使用した最初のコマンド名トークンです。これは、ユーザーが認識しているコマンド名の最初のワードです。特殊値アスタリスク (*) が (この定義の一部として) 指定された場合、このコマンド定義は、ユーザーが環境名 *etarget* を使用して発行し、他の場所でこれ以上明示的に定義されていないすべてのコマンドをカバーします。コマンド名の長さは最大 16 文字です。

itarget

コマンド・ストリングを処理するエージェントにこのコマンド定義が渡す内部環境名を指定します。これが必要であるのは、ユーザーが認識している外部環境名を、これらのコマンドを処理するエージェントの破損なしに再定義できるようにするためです。内部名と外部名が同じ場合、内部名を指定する必要はありません。特殊値「=」は、*itarget* が *etarget* と同じであることを表します。

icmdnm

内部コマンド名の最初のワードです。これは、処理されるコマンドを指定するために REXX コマンド・エージェントに渡されるコマンド名の最初の部分です。これは、*ecmdnm* と異なる場合にのみ、指定します。特殊値「=」は、*icmdnm* が *ecmdnm* と同じであることを表します。

pgmname

コマンドを処理するために EXEC CICS LINK によって呼び出される CICS プログラムを指定します。

CICS_loadmod

CICSLOAD オプションが指定されたために呼び出される CICS プログラムの名前を指定します。

注: プログラムはコマンドの最初のインスタンスでのみロードされ、そのアドレスが後続のコマンドに引き継がれます。

execname

このコマンド (単数または複数) を処理する REXX コマンド・サーバーとして呼び出される EXEC を指定します。この Server EXEC がすでに実行中であると、このコマンドはその実行中のサーバーに経路指定されます。この名前の REXX サーバーが実行していない場合は、自動サーバー始動 (Automatic Server

Initiation (ASI)) を使用してサーバーが自動的に始動されます。*execname* は、ファイル名 (ファイル・タイプのデフォルトは EXEC) を使用できますが、*filename.filetype* という形式を取ることもできます。

CICSLINK

定義された REXX コマンドの処理エージェントが、EXEC CICS LINK によって呼び出される標準 CICS プログラムであることを示すキーワードです。

CICSLOAD

処理エージェントが、EXEC CICSLOAD によってロードされる CICS プログラムであることを示すキーワードです。

REXX

この REXX コマンドの処理エージェントが、コマンド・サーバーとして機能する REXX EXEC であることを示すキーワードです。

AUTH

これは許可オプションです。

これが許可 REXX/CICS コマンドであることを示すキーワードです。許可 REXX/CICS ユーザー (AUTHUSER コマンドで指定) によってのみ、あるいは許可ライブラリーからロードされた EXEC 内からのみ実行できるコマンドです。

CLEAR

この DEFCMD の目的が、指定された外部ターゲット環境名およびコマンド名の以前の定義をすべてクリアすることにあることを示すキーワードです。

戻りコード

0

Normal return

1001

無効なコマンド

1021

プログラムをロードできない

1023

エントリーが見つからない

1048

使用可能なクライアントがない

1099

内部エラー

例

```
'DEFCMD CICS SEND = = SENDPGM (CICSLINK'
```

この例は、このユーザーに対してのみ SEND というコマンドを定義します。ユーザーは、次のように入力することにより、REXXCICS のデフォルトのコマンド環境でこのコマンドを発行できます。

```
'CICS SEND arg1 arg2 ... argn'
```

この例は、このコマンドを処理するために EXEC CICS LINK コマンドによって呼び出されるプログラム SENDPGM を示しています。

注:

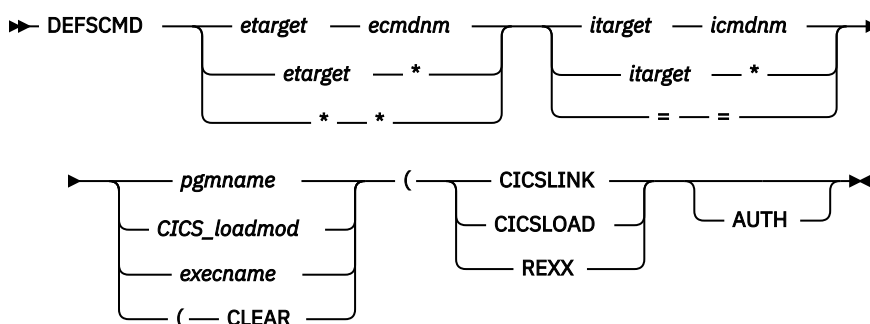
1. REXX/CICS 環境名が REXXCICS (すべての EXEC またはマクロが呼び出された場合のデフォルト) の場合、コマンド・ストリングの最初のトークンは、ADDRESS 環境 REXX 命令と一緒に使用できる可能性のある環境名です。これにより、さらに統合されたコマンド環境が提供され、ADDRESS 命令による恒常的な環境切り替えは必要なくなります。

2. EXEC CICS LINK および Assembler BASSM 命令 (CICSLD オプション) によって制御を受け取るコマンド・プログラムの呼び出しとパラメーター受け渡しのシーケンスは似ています。コマンド・プログラムの作成について詳しくは、[307 ページの『第 26 章 REXX/CICS コマンド定義』](#)を参照してください。
3. DEFCMD を使用して、ユーザー単位またはアプリケーション単位で、ユーザーのコマンド・セットを動的に調整することができます。DEFCMD コマンドは、ユーザーの PROFILE EXEC にも、あるいはアプリケーション EXEC にも入れることができます。DEFCMD を使用して、システム・コマンド定義を指定変更することもできます。
4. DEFCMD REXXCICS * は許可されません。
5. ユーザー・コマンド定義が検索されてから、システム・コマンド定義 (指定変更不能な DEFCMD は除く) が検索されます。
6. REXX コマンドは、REXX で作成することができます。これらの REXX コマンドは交代で、REXX で作成された他の REXX コマンドをビルディング・ブロック方式で呼び出します。DEFCMD により REXX ユーザー (プログラマー) からのインプリメンテーションの詳細情報は非表示にされるため、パフォーマンスが重要になった場合には、コマンドを REXX で即時に作成し、後で別の言語で透過的に作成し直すことができます。

DEFSCMD

DEFSCMD は、REXX システム・コマンドを定義 (または再定義) します。

これは、許可コマンドです。



オペランド

etarget

このコマンドを発行する REXX EXEC で使用された外部ターゲット環境の名前です。これは、コマンド・ストリングの送信先として使用された外部環境名です。この環境名は、テーブル内でルックアップされ、コマンド名と一緒に使用されて、処理のためにこのコマンド・ストリングが送信されるプログラム、REXX EXEC、またはキューを決定します。

注：外部ターゲットは、ADDRESS キーワード命令の環境名に一致するものであるか、あるいは REXXCICS が現行環境（これがデフォルト）の場合は、コマンド・ストリングの最初のトークンとして指定することができます。

ecmdnm

このコマンドを発行するのにユーザーが使用した最初のコマンド名トークンです。これは、ユーザーが認識しているコマンド名の最初のワードです。特殊値アスタリスク (*) が (この定義の一部として) 指定された場合、このコマンド定義は、ユーザーが環境名 *etarget* を使用して発行し、他の場所でこれ以上明示的に定義されていないすべてのコマンドをカバーします。

itarget

コマンド・ストリングを処理するエージェントにこのコマンド定義が渡す内部環境名を指定します。これが必要であるのは、ユーザーが認識している外部環境名を、これらのコマンドを処理するエージェントの破損なしに再定義できるようにするためです。内部名と外部名が同じ場合、内部名を指定する必要はありません。特殊値「=」は、*itarget*が*etarget*と同じであることを表します。

icmdnm

内部コマンド名の最初のワードです。これは、処理されるコマンドを指定するために REXX コマンド・エージェントに渡されるコマンド名の最初の部分です。これは、*ecmdnm* と異なる場合にのみ、指定します。特殊値「=」は、*icmdnm* が *ecmdnm* と同じであることを表します。

pgmname

コマンドを処理するために EXEC CICS LINK によって呼び出される CICS プログラムを指定します。

CICS_loadmod

CICSLOAD オプションが指定されたために呼び出される CICS プログラムの名前を指定します。

注: プログラムはコマンドの最初のインスタンスでのみロードされ、そのアドレスが後続のコマンドに引き継がれます。

execname

このコマンド (単数または複数) を処理する REXX コマンド・サーバーとして呼び出される EXEC を指定します。この Server EXEC がすでに実行中であると、このコマンドはその実行中のサーバーに経路指定されます。この名前の REXX サーバーが実行していない場合は、自動サーバー始動 (Automatic Server Initiation (ASI)) を使用してサーバーが自動的に始動されます。*execname* は、ファイル名にすることも (ファイル・タイプはデフォルトで EXEC になる)、*filename.filetype* の形式にすることもできます。

CICSLINK

定義された REXX コマンドの処理エージェントが、EXEC CICS LINK によって呼び出される標準 CICS プログラムであることを示すキーワードです。

CICSLOAD

処理エージェントが、EXEC CICSLOAD によってロードされる CICS プログラムであることを示すキーワードです。

REXX

この REXX コマンドの処理エージェントが、コマンド・サーバーとして機能する REXX EXEC であることを示すキーワードです。

AUTH

これが許可 REXX/CICS コマンドであることを示すキーワードです。許可 REXX/CICS ユーザー (AUTHUSER コマンドで指定) によってのみ、あるいは許可ライブラリーからロードされた EXEC 内からのみ実行できるコマンドです。

CLEAR

この DEFSCMD の目的が、指定された外部ターゲット環境名およびコマンド名の以前の定義をすべてクリアすることにあることを示すキーワードです。

戻りコード

0

Normal return

1101

無効なコマンド

1121

プログラムをロードできない

1123

エントリーが見つからない

1148

使用可能なクライアントがない

1199

内部エラー

例

```
'DEFSCMD CICS SEND = = SENDPGM (CICSLINK'
```

この例は、このユーザーに対してのみ SEND というコマンドを定義します。ユーザーは、次のように入力することにより、REXXCICS のデフォルトのコマンド環境でこのコマンドを発行できます。

```
'CICS SEND arg1 arg2 ... argn'
```

この例は、このコマンドを処理するために EXEC CICS LINK コマンドによって呼び出されるプログラム SENDPGM を示しています。

注：

1. REXX/CICS 環境名が REXXCICS (すべての EXEC またはマクロが呼び出された場合のデフォルト) の場合、コマンド・ストリングの最初のトークンは、ADDRESS 環境 REXX 命令と一緒に使用できる可能性のある環境名です。これにより、さらに統合されたコマンド環境が提供され、ADDRESS 命令による恒常的な環境切り替えは必要なくなります。
2. EXEC CICS LINK および Assembler BASSM 命令 (CICSLOAD オプション) によって制御を受け取るコマンド・プログラムの呼び出しとパラメーター受け渡しのシーケンスは似ています。コマンド・プログラムの作成について詳しくは、[307 ページの『第 26 章 REXX/CICS コマンド定義』](#)を参照してください。
3. DEFSCMD の最初の 2 つのオペランドがすべてアスタリスク (*) の場合、これは、それ以上具体的なコマンド定義の範囲内にはない (一致しない) REXX コマンドについて発行されるコマンド処理エージェントを指定する総称的な定義です。
4. ユーザー・コマンド定義が検索されてから、システム・コマンド定義 (指定変更不能な DEFSCMD は除く) が検索されます。
5. REXX コマンドは、REXX で作成することができます。これらの REXX コマンドは交代で、REXX で作成された他の REXX コマンドをビルディング・ブロック方式で呼び出します。DEFSCMD により REXX ユーザー (プログラマー) からのインプリメンテーションの詳細情報は非表示にされるため、パフォーマンスが重要になった場合には、コマンドを REXX で即時に作成し、後で別の言語で透過的に作成し直すことができます。

DEFTRNID

DEFTRNID は、領域全体の許可コマンドであり、これを使用すると、特定の CICS トランザクション ID に対して呼び出される EXEC の名前を定義できます。

これは、許可コマンドです。

➡ DEFTRNID — *trnid* — *execname* — CLEAR

オペランド

trnid

1 文字から 4 文字の CICS トランザクション ID を指定します。

execname

REXX ファイル・システム内では、1 文字から 17 文字の REXX/CICS EXEC 名を形式 *filename.filetype* で指定します。MVS 区分データ・セット内に EXEC がある場合は、1 文字から 8 文字までの名前です。

CLEAR

このトランザクション ID について定義を除去することを示すキーワードです。

戻りコード

0

Normal return

1202

無効なオペランド

1222

無効なオプション

1223

trantable 情報の取得中にエラーが発生した

1225

trantable 情報の取得中にエラーが発生した

1226

EXEC 名の長さエラーです

1228

trantable 値の設定中にエラーが発生した

1233

トランザクションがテーブルに見つかりません

例

XYZ という名前の新規 CICS トランザクション ID を定義し、その開始時に EXEC TESTEXEC を呼び出すようにした後で、次の手順を実行します。

1. DD 名 CICUSER に割り振られるか連結されるデータ・セット内に TESTEXEC を作成します。
2. REXX/CICS REXXTRY ユーティリティの下で、コマンド DEFTRNID XYZ TESTEXEC を入力して、REXXTRY ユーティリティを終了します。
3. RDO の下で、REXX Development System for CICS の場合、以下のように入力します。

```
CEDA DEFINE TRAN(XYZ) PROGRAM(CICREXD) GROUP(CICREXX)
```

REXX Runtime Facility for CICS の場合、以下のように入力します。

```
CEDA DEFINE TRAN(XYZ) PROGRAM(CICREXR) GROUP(CICREXX)
```

次に、以下のように入力します。

```
CEDA INSTALL TRAN(XYZ) GROUP(CICREXX)
```

4. 画面をクリアし、CICS トランザクション ID XYZ を入力して Enter を押します。TESTEXEC EXEC が実行するはずですが。

注：

1. DEFTRNID 定義は、通常、REXX/CICS の始動時に実行される CICSTART EXEC に配置する必要があります。
2. 提供されている CICREXD プログラムまたは CICREXR プログラムを呼び出すには、このトランザクション ID を定義する必要があります。例のステップ [379 ページの『3』](#) を参照してください。

DIR

DIR は、現行ディレクトリーの内容を表示するか、オプションで、ディレクトリーの内容を REXX 複合変数に戻します。



オペランド

dirid

表示される一部または完全な REXX ファイル・システム・ディレクトリーを指定します。これを省略した場合、現行ディレクトリーが表示されます。

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#)を参照してください。Stem.0 には、エントリー内のエレメント数が含まれています。これを省略すると、内容が画面に表示されます。

戻りコード

0

Normal return

321

現行 RFS ディレクトリー情報にアクセスできない

322

無効な語幹名

325

RFS ディレクトリーの取得中にエラーが発生した

例

```
'DIR ¥USERS¥USER2 (X.'
```

この例は、¥USERS¥USER2 のディレクトリーの内容を、REXX 複合変数の X.1 から X.n までに入れます。X.0 には、戻されるエレメントの数が入ります。

注：現行ディレクトリーは、CD コマンドによって指定されます。

EDIT

EDIT は、新規編集セッションを開きます。



オペランド

NONAME

ファイル ID は指定されません。これはデフォルトです。

fileid

編集しているファイルのファイル ID。

PDS_name(mem)

完全修飾 MVS PDS 名とメンバー名を指定します。

PDS

PDS の編集時に、PDS メンバー名の後に続くキーワード。

MACRO

編集中のファイルに適用される命令のグループを指定するキーワードです。

macroname

プロファイル・マクロ・ファイル ID (REXX EXEC 名) のファイル名部分。

戻りコード

0

Normal return

201

無効なコマンド

211

ファイル ID が無効です

226

ファイルは現在、編集集中です

299

内部エラー

例

```
'CD ¥USERS¥USER2¥' /* specify current working directory */
'EDIT TEST.EXEC'    /* edit an existing file in the PATH */
                    /* or create new file in current dir */
```

この例は、ディレクトリー ¥USERS¥USER2 内のメンバー TEST.EXEC を編集します。

REXX/CICS エディターについて詳しくは、[255 ページの『第 23 章 REXX/CICS テキスト・エディター』](#)を参照してください。

EXEC

EXEC は、より低いレベルで REXX EXEC を (ネストされた EXEC として) 呼び出します。この新しい EXEC の変数はすべて、上位レベル EXEC とは別個のものとされ、ネストされた EXEC が終了するまで中断されます。

➡ EXEC — *execid* — *args* ➡

オペランド

execid

name.modifier という形式の EXEC の 1 から 17 文字の ID。 *name* と *modifier* はそれぞれ最大 8 文字です。 *modifier* を指定しない場合、EXEC が想定されます。

args

呼び出された EXEC の引数ストリング。

戻りコード

n

呼び出された EXEC の出口によって設定された戻りコード。[172 ページの『EXIT』](#)を参照してください。

0

Normal return

-3

EXEC が見つかりません

-10

EXEC 名が指定されていません

-11

EXEC 名が無効です

-12

GETMAIN error

-99

内部エラー

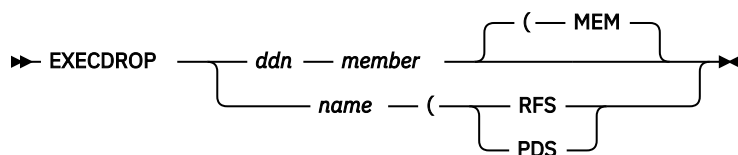
例

```
'EXEC ABC'
```

この例は、EXEC の ABC.EXEC を実行します。

EXECDROP

EXECDROP は、EXECLOAD コマンドを使用して以前にロードされた仮想記憶域から EXEC を除去します。
これは、許可コマンドです。



オペランド

ddn

読み取られる DD 名を指定します。

member

使用される区分データ・セットのメンバーを指定します。

name

完全修飾 RFS ファイル名または完全修飾 PDS 名とメンバーを指定します。

MEM

DD 名とメンバー名が指定されていることを示します。

RFS

RFS ファイルが指定されていることを示します。

PDS

区分データ・セット名とメンバーが指定されていることを示します。

戻りコード

0

Normal return

1401

無効なコマンド

1402

無効なオペランド

1423

EXECLOAD 情報の格納中にエラーが発生した

1425

EXECLOAD 情報の取得中にエラーが発生した

1448

使用可能なクライアントがありません

例

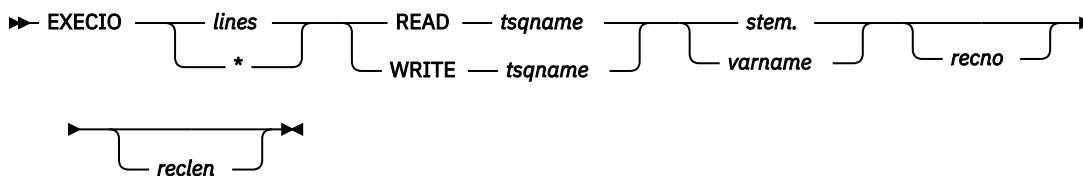
```
'EXECDROP POOL1:¥USERS¥USER2¥TEST.EXEC (RFS'
```

この例は、仮想記憶域から RFS ファイルを除去します。

注：部分ディレクトリー ID が指定されると、完全修飾ディレクトリー ID を取得するために、現行作業ディレクトリー値の末尾に一時的に付加されます。

EXECIO

EXECIO は、CICS 一時記憶域キューへのファイル入出力を実行します。



オペランド

lines

読み取りまたは書き込む行の数を指定します。アスタリスク (*) は、READ 操作についてのみ指定される特殊なケースであり、ファイルがターゲット行 (ターゲット行が指定されていない場合は行 1) からファイルの終わりまで読み取られることを示します。

READ

CICS 一時記憶域キュー (TSQ) から 1 つ以上のレコードを読み取ります。

WRITE

CICS 一時記憶域キュー (TSQ) へ 1 つ以上のレコードを書き込みます (または再書き込みします)。

tsqname

1 文字から 8 文字の一時記憶域キュー名を指定します。

stem

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#)を参照してください。

varname

この EXECIO 操作のソースまたはターゲットである REXX 変数名を指定します。

recno

READ または WRITE の先頭に置く一時記憶域キュー内のレコード番号を指定します。

reclen

CICS 一時記憶域に書き込まれるレコードの長さを指定します。reclen を省略した場合、長さはデフォルトで 80 バイトになります。

戻りコード

n

エラーが検出された場合に CICS によって戻される戻りコード。

0

Normal return

-202

無効なオペランド

-221

指定したオペランドが多すぎます

-222

recno オペランドが範囲外です

-224

lines オペランドが無効です

例

```
x.1 = 'line 1'
x.2 = 'Line Two'
EXECIO 2 WRITE QUEUE1 X.
```

この例は、CICS 一時記憶域キューにデータを書き込みます。

```
'EXECIO 2 READ QUEUE1 Y.'  
say y.0 /* ==> 2 */  
say y.1 /* ==> 'line 1' */  
say y.2 /* ==> 'Line Two' */
```

この例は、一時記憶域キューからデータを読み取ります。

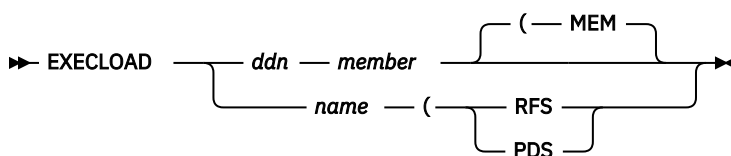
注：

1. 許可される最大レコード長は 256 バイトです。
2. READ 操作に *stem* が指定される場合 (複数のレコードが読み取られる場合には、さらに *stem* の指定が必要)、読み取られるレコードの実際のは *stem.0* に置かれます。
3. CICS 提供の CEBR トランザクションを使用して、一時記憶域キューをブラウズします。例えば、例で作成されたキューを表示するには、CEBR QUEUE1 と入力します。
4. CEBR には、PUT 関数と GET 関数が用意されています。これらの関数を使用して、CICS 一時データ・キューと CICS 一時記憶域キューの間でコピーすることができます。

EXECLOAD

EXECLOAD は、EXEC を仮想記憶域にロードします。

これは、許可コマンドです。



オペランド

ddn

読み取られる DD 名を指定します。

member

使用される区分データ・セットのメンバーを指定します。

name

完全修飾 RFS ファイル名または完全修飾 PDS 名とメンバーを指定します。

MEM

DD 名とメンバー名が指定されていることを示します。

RFS

RFS ファイルが指定されていることを示します。

PDS

区分データ・セット名とメンバーが指定されていることを示します。

戻りコード

0

Normal return

1501

無効なコマンド

1502

無効なオペランド

1523

EXECLOAD 情報の格納中にエラーが発生した

1525

EXECLOAD 情報の取得中にエラーが発生した

1530

CICPDS ルーチンにリンクできません

1531

CICPDS ルーチンからエラーが戻されました

1532

RFS READ からエラーが戻されました

1532

PDS ビルドからエラーが戻されました

1547

GETMAIN error

1548

使用可能なクライアントがない

1599

内部エラー

例

```
'EXECLOAD POOL1:¥USERS¥USER2¥TEST.EXEC (RFS'
```

この例は、EXEC の TEST.EXEC を RFS から仮想記憶域にロードします。TEXT.EXEC の後続の呼び出しでは、ロードされたコピーを使用します。

注：

1. EXEC が仮想記憶域にロードされると、その EXEC は自動的にすべてのユーザーによって共有されます。
2. ストレージ内にある EXEC をより新しいコピーに置き換えるために EXECLOAD が実行されると、以前のコピーに基づくすべてのアクティブ EXEC が終了するまで、それらコピーのシャドウが仮想記憶域で保持されます。これは、使用回数によって実行されます。
3. 仮想記憶域にロードされる EXEC は、常に他の EXEC の前に使用されます。2つの EXEC が同じ名前で、1つが RFS の現行ディレクトリーにあり、もう1つが仮想記憶域にロードされていると、RFS コピーを実行できないため、プログラムには慎重に名前を付けてください。
4. EXEC が仮想記憶域にロードされると、EXEC のコピーは、ロード元のセキュリティ特性を継承します。例えば、すべてのユーザーが CICEXEC データ・セットの EXEC を実行でき、CICEXEC からロードされた EXEC はすべて REXX/CICS 許可コマンドを使用できます。[CICS 初期設定 JCL の更新](#)を参照してください。

EXECMAP

EXECMAP は、DD 名とメンバー、ユーザーの数、記述子テーブル開始 (16 進) のほか、EXECLOAD を使用してロードされた EXEC に必要なストレージの量を戻します。

➡ EXECMAP ➡

戻りコード**0**

Normal return

1623

EXECLOAD ディレクトリーが見つかりません

例

```
'EXECMAP'
```

EXEC の POOL1:\USERS\USER1\TEST.EXEC が以前に EXECLOAD を使用してロードされた場合、結果の表示は次のようになります。

EXEC name	Use	Location	Size	Fully Qualified Name
TEST.EXEC	0	09369083	287	POOL1:\USERS\USER1\TEST.EXEC

EXPORT

EXPORT は、RFS ファイルを MVS データ・セットにエクスポートします。

➡ EXPORT — *rfs_fileid* — *dsn* →

オペランド

dsn

完全修飾した (引用符なしの) MVS 物理順次データ・セットまたは区分編成データ・セット名を指定します。区分データ・セット (DSORG=PO) の場合は、括弧でメンバー名を囲まなければなりません。

rfs_fileid

完全修飾 REXX ファイル・システム・ファイル ID を指定します。

戻りコード

0

Normal return

1701

無効なコマンド

1702

無効なオペランド

1723

RFS 書き込みエラー

1724

RFS 読み取りエラー

1725

動的割り振りが失敗しました

1726

動的解放が失敗しました

1727

一時データ・キューのオープンが失敗しました

1728

メンバーがエンキューされていません

1729

メンバーは使用中です

1730

レコードが切り捨てられました

1733

エクスポートする入力が見つかりません

1735

一時データ・エラー

1736

予期しない CICS エラーです。

1737

Invalid request

1738

データ・セット名が無効です

1739

無効な後処理

1741

DSORG がサポート対象外

1742

一時データ・プールのビルド・エラー

1743

REXX 一時データ・キューが使用不可です/見つかりません

1744

ユーザーがサインオンしていません/データ・セットへのアクセスを許可されていません

1745

データ・セットが空

1746

REXX キューが見つかりません

1748

Task Input/Output Table (TIOT) に DD 名のエントリーがありません

1749

複合データ・セットにはエクスポートできません

1750

DD 名として複数のデータ・セットが連結されています

1799

内部エラー

例

```
'EXPORT POOL1:\USERS\USER1\TEST.DATA USER1.TEST.DATA'
```

この例では、POOL1:¥USERS¥USER1¥TEST.DATA を RFS ファイルから MVS 順次データ・セットにコピーします。

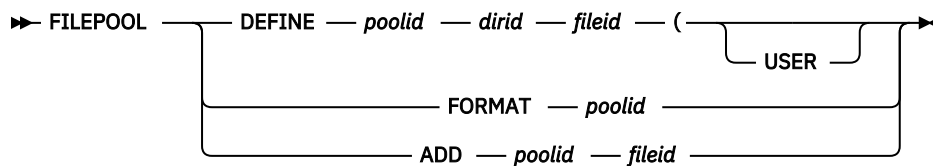
注:

1. このコマンドは SVC 99 を実行するのでパフォーマンスが懸念されます。そのため、頻繁に使用することはお勧めしません。
2. REXX/CICS 提供の CICSECX1 セキュリティー出口が呼び出され、CICS サインオン・ユーザー ID が、指定された MVS データ・セットへのアクセスを許可されているか検証されます。デフォルトでは、データ・セット名の高位接頭部が CICS サインオン・ユーザー ID と一致していなければなりません。CICS 領域も、指定された MVS データ・セットへのアクセス許可を (例えば、外部セキュリティが有効な場合は外部セキュリティ・マネージャーから) 受けていなければなりません。
3. MVS データ・セットは DISP=OLD を割り振られます。
4. DSN に指定する MVS データは、既に割り振られていなければなりません。順次データ・セットを使用する場合は、rfs_fileid に指定した RFS ファイルの内容に置き換えられます。区分データ・セットや既存のメンバー名を指定した場合は、RFS ファイルの内容に置き換えられます。
5. CICS 領域に定義されている MVS データ・セットにエクスポートした場合、CICS 領域がリサイクルされるまでそのデータ・セットのファイル属性指定は OLD に変更されます。

FILEPOOL

FILEPOOL は、RFS ファイル・プール管理アクティビティーを実行します。

これは、許可コマンドです。



オペランド

DEFINE

新しい RFS ファイル・プールを定義します。

poolid

ターゲット・ファイル・プールの名前を指定します。

dirid

ファイル・プール・ディレクトリーの CICS ファイル ID を指定します。

fileid

ファイル・プールの VSAM ファイルの CICS ファイル ID を指定します。

USER

ユーザー・ファイル・プールであることを示すオプションのキーワードです。ユーザーは作成された ¥USERS ディレクトリーを自動的に持つので、複数のユーザーが、ディレクトリーへのアクセスを制御するために RFS セキュリティー (CICS セキュリティーと比較して) を使用してこのファイル・プールを共用します。

FORMAT

新しい RFS ファイル・プール内の最初のファイルをフォーマット設定します。

ADD

既存のファイル・プールに追加の VSAM ファイルを追加します。

戻りコード

0

Normal return

1802

無効なオペランド

1821

無効なファイル・プール・サブコマンド

1822

ファイル・プール・サブコマンドが指定されていません

1823

ファイル・プール情報の格納中にエラーが発生した

1824

ファイル・プール ID が指定されていません

1825

ファイル・プール情報の取得中にエラーが発生した

1826

ファイル・プール ID が無効です

1827

取得されたファイル・プール・データが無効です

1828

ファイル・プールが定義されていません

1829

RFS はライブラリーをファイル・プールに追加できませんでした

1830

RFS はユーザー・ディレクトリーを作成できませんでした

1831

ファイル・プールの DDNAME を指定する必要があります

1832

DDNAME が無効です

1833

ファイル・プール変数が破損しています

1834

プール ID は既に存在します

1835

DDNAME は既に使用されています

1836

ファイル・プールをフォーマットできませんでした

1837

ファイル・プールは最初にフォーマットする必要があります

1838

ファイル・プールの ADD レコードがいっぱいです

1839

ファイル ID が見つかりません

例

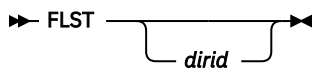
```
'FILEPOOL DEFINE POOL1 REXXDIR1 REXXLIB1 (USER'
```

この例は、ファイル・プール POOL1 を定義し、使用する CICS ファイル定義が REXXLIB1 であることを RFS に指示します。また、RFS MKDIR を発行して、¥USERS ディレクトリーを作成するよう FILEPOOL FORMAT コマンドに指示します。

注：これは、REXX/CICS 管理者またはシステム・プログラマーが実行する許可コマンドです。

FLST

FLST は、ファイル・リスト・ユーティリティーを呼び出して、ファイルを処理します。

**オペランド*****dirid***

ファイル・リストが表示されるオプションの完全ディレクトリー ID または部分ディレクトリー ID を指定します。*dirid* を指定しない場合、デフォルトで現行作業ディレクトリーに設定されます。

戻りコード

FLST は、RFS によって与えられた戻りコードを戻します。

例

```
'FLST'
```

この例は、現行作業ディレクトリーのメンバーのファイル・リストを表示します。

注：

1. FLST が呼び出そうとするデフォルトのユーザー・プロファイル・マクロは EXEC ID FLSTPROF です。FLSTPROF マクロは、ユーザー (またはユーザーのグループ) が固有のデフォルトを指定できるようにします。
2. REXX ファイル・システムについて詳しくは、[283 ページの『第 24 章 REXX/CICS ファイル・システム』](#)を参照してください。

FREE

FREE は、データ定義名の割り振りを解除します。

➡ FREE — DDNAME — *ddname* ➡

オペランド

DDNAME

FREE が DD 名によるものであることを示すキーワードです。

DD 名

データ定義名を指定します。

戻りコード

SVC 99 によって与えられた戻りコード

詳しくは、[z/OS MVS Programming: Authorized Assembler Services Guide](#) を参照してください。

1702

無効なオペランド

例

```
'ALLOC USER1A USER1.REXX(TEST)'  
'FREE DDNAME USER1A'
```

この例は、データ・セット USER1.REXX のメンバー TEST を DD 名 USER1A に割り振った後、別のデータ・セットでできるように DD 名 USER1A を解放します。

GETVERS

GETVERS は、現行 REXX/CICS、プログラム名、バージョン、およびコンパイル時刻情報を取得し、それを REXX 変数 VERSION に入れます。

➡ GETVERS ➡

戻される情報の形式は、*pgmnameVxRyMmmmm mm/dd/yyyy hh:mm* です。ここで、各部の意味は次のとおりです。

pgmname

REXX/CICS Development System の場合は CICREXD が、REXX/CICS Runtime Facility の場合は CICREXR が示されます。

x

REXX/CICS のバージョン番号を指定します。

y

REXX/CICS のリリース番号を指定します。

mm/dd/yyyy

REXX/CICS 基本プログラムのコンパイル日付を指定します。

hh.mm

REXX/CICS 基本プログラムのコンパイル時刻を指定します。

- 1723**
RFS 書き込みエラー
- 1724**
RFS 読み取りエラー
- 1725**
動的割り振りが失敗しました
- 1726**
動的解放が失敗しました
- 1727**
一時データ・キューのオープンが失敗しました
- 1728**
メンバーがエンキューされていません
- 1729**
メンバーは使用中です
- 1730**
レコードが切り捨てられました
- 1733**
エクスポートする入力が見つかりません
- 1735**
一時データ・エラー
- 1736**
予期しない CICS エラーです。
- 1737**
Invalid request
- 1738**
データ・セット名が無効です
- 1739**
無効な後処理
- 1741**
DSORG がサポート対象外
- 1742**
一時データ・プールのビルド・エラー
- 1743**
REXX 一時データ・キューが使用不可です/見つかりません
- 1744**
ユーザーがサインオンしていません/データ・セットへのアクセスを許可されていません
- 1745**
データ・セットが空
- 1746**
REXX キューが見つかりません
- 1799**
内部エラー

例

```
'IMPORT USER1.TEST.DATA POOL1:\USERS\USER1\TEST.DATA'
```

この例では、USER1.TEST.DATA を MVS 順次データ・セットから RFS ファイル POOL1:¥USERS ¥USER1¥TEST.DATA にコピーします。


注：

- ## LISTCMD

►► LISTCMD —————►►

envname cmdname

LISTPOOL

► LISTPOOL 

- ・ユーザーのディレクトリーが含まれているかどうか

戻りコード

O

Normal return

2225

ファイル・プール情報の取得中にエラーが発生した

2226

語幹変数名が無効です

例

```
'LISTPOOL LST.'
```

この例は、RFS ファイル・プールに関する情報を REXX 複合変数の LST.1 から LST.n までに入れます。LST.0 には、戻されたエレメントの数が含まれています。

注: これは、すべての定義済み RFS ファイル・プールを表示する一般ユーザー・コマンドです。

LISTTRNID

LISTTRNID は、DEFTRNID コマンドによって作成された現行のトランザクション ID 定義をリストします。

これは、許可コマンドです。

➤ LISTTRNID ➤

戻りコード

O

Normal return

2325

| trantable | 情報の取得中にエラーが発生した |

例

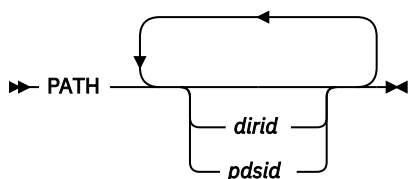
'LISTTRNID'

CICSTART EXEC は、デフォルトのトランザクションおよびそれぞれの EXEC 名を定義します。結果として生じる表示は以下のとおりです。

TRNID	EXEC name
REXX	CICRXTRY
EDIT	CICEDIT
FLST	CICFLST
*SIB	CICOVSIB
End of Transaction table list.	

PATH

PATH は、REXX EXEC の検索パスを定義します。



注：現行パスの設定を取得するには、SET コマンド (パラメーターなし) を使用します。

オペランド

dirid

実行する EXEC を見つけようとするときに検索対象にする、1 つ以上の完全修飾 REXX ファイル・システム・ディレクトリーを指定します。

完全な RFS ディレクトリー ID は、プール ID で始まり、POOL1:\dirid1\...\diridn という形式になります。

複数のディレクトリー ID を指定する場合には、それらを区切るためにブランクを使用します。

pdsid

1 つ以上の MVS 区分データ・セット名を指定します。

戻りコード

0

Normal return

625

パス情報の取得中にエラーが発生した

626

RFS ディレクトリー名が無効です

627

PDS 名が無効です

628

結果値の設定中にエラーが発生した

629

データ・セット名が無効です

630

パス情報の格納中にエラーが発生した

631

現在定義されているパスはありません

632

結果の PATH に RFS ディレクトリーや PDS 名が含まれていません

例

```
'PATH POOL1:¥USERS¥USER2 POOL2:¥USERS¥USER2¥PROJECT1'
```

この例は、このリスト内のディレクトリーが指定された順序で (左から右へ) 検索されることを示しています。

```
'PATH MYUSERID.REXX1.EXECS MYUSERID.REXX2.EXECS'
```

この例では、最初の区分データ・セットから検索が開始されます。

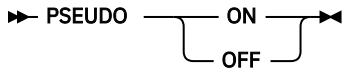
注：

1. *dirid* と *pdsid* を単一の PATH ステートメントで混用できます。
2. 複数のストリングを結合して 1 つの変数に入れ、その変数を PATH コマンドに指定することで、非常に長い PATH ディレクトリー・リストを作成できます。
3. PATH コマンド定義は、CICS の複数回の再始動にわたって引き継がれることはありません。PATH 定義を恒久的に変更するには、PATH コマンドを、ユーザー ID の基本ディレクトリー内の PROFILE EXEC に挿入してください。

4. この PATH コマンドは累積されません。つまり、最後の PATH コマンドが前の PATH 定義に取って代わります。
5. *dirid* も *pdsid* も指定しない場合は、ユーザーの現行作業パスが取得されて、REXX 特殊変数 RESULT に入れます。
6. ゼロ以外の戻りコードを受信した場合、RESULT 特殊変数の内容は予測できません。

PSEUDO

PSEUDO は、疑似会話型モードをオンまたはオフにします。



オペランド

ON

現行 EXEC で以下のいずれかの条件が検出されると、会話型の端末読み取りが即時に行われるのではなく、EXEC CICS RETURN TRANSID を使用して、端末入力が行われるまでこの EXEC を中断し、その後、端末読み取りが行われるように、自動疑似会話型サポートを使用可能にします。

- REXX PULL 命令
- REXX/CICS WAITREAD コマンド
- PANEL RECEIVE コマンド
- 画面がいっぱいで、画面の右下隅に MORE と表示される (暗黙的な READ、画面のクリアを待機中)

OFF

自動疑似会話型サポートを使用不可 (オフ) にします。

戻りコード

0

Normal return

2502

無効なオペランド

2521

オペランドが指定されていません

例

```
'PSEUDO ON'
```

この例は、疑似会話型モードをオンにします。

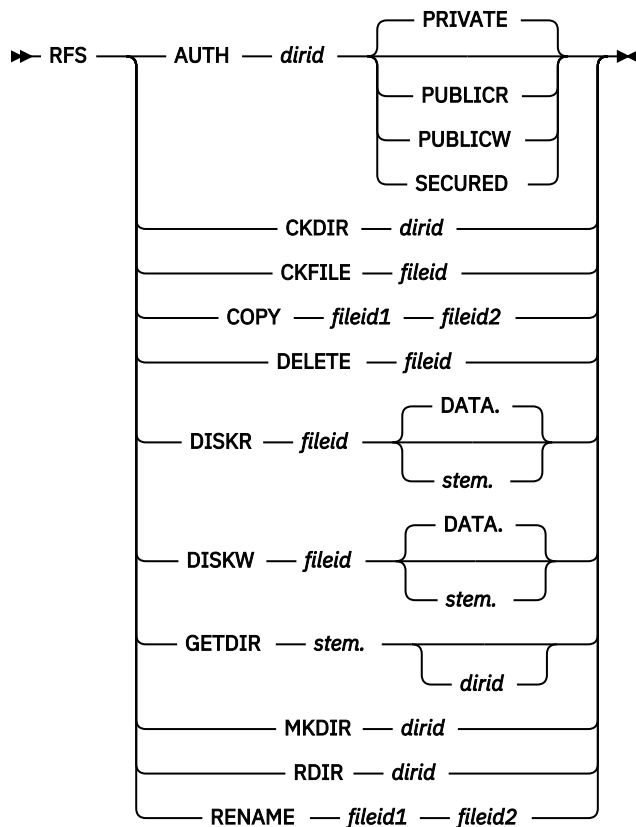
注: PSEUDO ON/OFF 設定は一時的なものです。疑似設定は、現行 EXEC が実行している間は存在します。ネストされた EXEC はすべて、PSEUDO の現行設定を継承します。現行 EXEC が終了すると、この EXEC への入り口にある PSEUDO の値が復元されます。SETSYS PSEUDO コマンドは、PSEUDO のシステム・デフォルト設定値を定義します。



注意: PSEUDO ON により、ユーザーの REXX EXEC は即時に疑似会話型になり、これにより、CICS はすべてのファイル変更をコミットし、次の疑似会話型端末読み取りの際に非共用 GETMAINEd 領域をすべて開放します。

RFS

RFS は、REXX ファイル・システムに対してファイル入出力を実行します。



オペランド

AUTH

RFS ディレクトリーへのアクセスを許可するコマンドです。指定されたアクセスは、このディレクトリー内のすべてのファイルに適用されます。

dirid

REXX ファイル・システムのディレクトリー ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CD コマンドの [351 ページ](#) の『CD』を参照してください。

PRIVATE

ディレクトリーの所有者のみが、ファイルへの読み取り/書き込みアクセスを持つことを指定します。これはデフォルトです。

PUBLICR

あらゆるユーザーがディレクトリー内のファイルに対して読み取り専用アクセス権限を持つことを指定します。

PUBLICW

あらゆるユーザーがディレクトリー内のファイルに対して読み取り/書き込みアクセス権限を持つことを指定します。

SECURED

外部セキュリティー・マネージャーがディレクトリー内のファイルに対するアクセス権限を付与することを指定します。

CKDIR

既存の RFS ディレクトリー・レベルを検査するコマンドです。

CKFILE

指定されているか、部分修飾なのか完全修飾なのか、あるいはファイル ID が存在するかどうかを検査するコマンドです。

fileid

ファイル ID を指定します。

COPY

ファイルをコピーし、あらゆる既存のファイルを置き換えるコマンドです。

fileid1

ソース・ファイル ID を指定します。完全修飾または部分修飾のディレクトリーとファイル ID を使用できます。

fileid2

ターゲット・ファイル ID を指定します。完全修飾または部分修飾のディレクトリーとファイル ID を使用できます。

DELETE

RFS ディレクトリー・レベルまたは RFS ファイルを削除するコマンドです。

fileid

ソース・ファイル ID を指定します。完全修飾または部分修飾の形式を使用できます。

DISKR

RFS ファイルからレコードを読み取るコマンドです。

stem

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#)を参照してください。デフォルトの語幹は DATA. です。

DISKW

レコードを RFS ファイルに書き込むコマンドです。

GETDIR

現行ディレクトリーまたは指定されたディレクトリーの内容のリストを戻して、指定された REXX 配列に入れるコマンドです。

MKDIR

新しい RFS ディレクトリー・レベルを作成するコマンドです。

RDIR

指定された RFS ディレクトリーを除去するコマンドです。

RENAME

RFS ファイルの名前を新しいものに変更するコマンドです。

fileid1

ソース・ファイル ID を指定します。完全修飾または部分修飾のディレクトリーとファイル ID を使用できます。

fileid2

ソース・ターゲット・ファイル ID を指定します。完全修飾または部分修飾のディレクトリーとファイル ID を使用できます。

戻りコード

0

Normal return

101

無効なコマンド

102

無効なオペランド

103

ファイルが見つかりません

104

許可されません

105

ファイルは既に存在します

107

ファイル・プールのスペースが不十分です

110

要求は失敗しました

111

ファイル ID が無効です

113

ディレクトリーが見つかりません

115

ディレクトリーは既に存在しています

116

ディレクトリーが指定されていません

121

ファイルが破損しています

122

stem.0 は無効または範囲外です

126

パス・エラー

127

CICS 入出力エラー

128

この場所からのコマンドは無効です

130

Directory not empty

131

オペランドが欠落しています

132

ファイル・プール・データ・レコードが欠落しています。ファイル・プールがフォーマットされていない可能性があります。

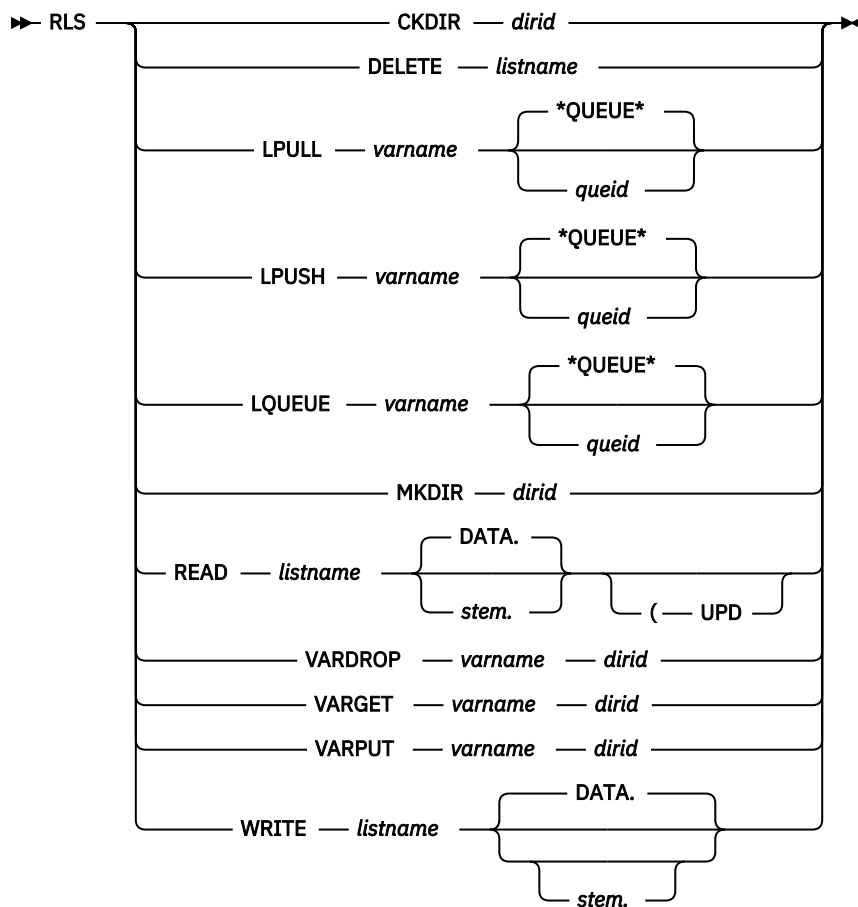
199

内部エラー

注：ファイル・アクセス・セキュリティ検査は、ファイル・レベルではなく、ディレクトリー・レベルで実行されます。指定されたファイル ID が完全修飾 ID でない場合、現行ディレクトリーまたは PATH ディレクトリーは、部分名を完全修飾名に解決するために使用されます。この場合、これ以上の検査は不要です。完全修飾ファイル ID が使用されている場合、それが収容されているディレクトリーは、ファイルのオープン時に、適切なアクセス許可の有無を検査されます。

RLS

RLS は、REXX List System に対してリスト入出力を実行します。



オペランド

CKDIR

既存の RLS ディレクトリー・レベルについて検査するコマンドです。

dirid

REXX List System のディレクトリー・レベル ID を指定します。これは、部分修飾または完全修飾です。詳しくは、CLD コマンドの [367 ページの『CLD』](#) を参照してください。

DELETE

RLS ディレクトリー・レベルまたは RLS リストを削除するコマンドです。

listname

REXX List System のリスト ID を指定します。これは、部分修飾または完全修飾です。

LPULL

キューの先頭からレコードをプルするコマンドです。

varname

単純 REXX 変数名を指定します。これはピリオドで終わらず、それにより、変数名と語幹名が区別されます。

QUEUE

特殊デフォルト名を指定します。

queid

LPULL、LPUSH、または LQUEUE によってアクセスされる特殊タイプの RLS リストの ID です。

LPUSH

キューの先頭にレコードをプッシュする (LIFO) コマンドです。

LQUEUE

キューの終わりにレコードを追加する (FIFO) コマンドです。

MKDIR

新しい RLS ディレクトリー・レベルを作成するコマンドです。

READ

RLS リストから語幹にレコードを読み取るコマンドです。

listname

リスト ID を指定します。

stem.

語幹の名前を指定します。語幹はピリオドで終了する必要があります。[155 ページの『語幹』](#)を参照してください。デフォルトの語幹は DATA. です。

UPD

更新のためにリストにエンキューします。

VARDROP

RLS 変数が削除されることを示すキーワードです。

VARGET

RLS 変数を使用し、その変数を同じ名前の REXX 変数にコピーするコマンドです。

VARPUT

REXX 変数を使用し、その変数を同じ名前の RLS 変数にコピーするコマンドです。

WRITE

レコードを語幹から RLS リストに書き込むコマンドです。

戻りコード**0**

Normal return

701

無効なコマンド

702

無効なオペランド

713

ディレクトリーが見つかりません

715

ディレクトリーは既に存在しています

716

ディレクトリーが指定されていません

723

リストが見つかりません

726

リストが指定されていません

728

リストが更新モードになっています

729

リストが更新モードになっていません

730

ユーザーがサインオンしていません

732

キューが空です

733

指定されたキューが見つかりませんでした

- 736**
語幹または変数が指定されていません
- 737**
語幹名または変数名が長すぎます。
- 738**
語幹または変数のカウントが無効です
- 743**
ブロックが見つかりません
- 746**
CICGETV エラー
- 747**
GETMAIN error
- 748**
FREEMAIN error
- 749**
ENQ エラー
- 750**
DEQ エラー
- 751**
動的領域 GETMAIN のエラー
- 752**
保管済み変数データにエラーがあります
- 753**
保管済み変数は見つかりませんでした
- 754**
ユーザーはリストの所有者ではありません

SCRNINFO

SCRNINFO は、2 桁の 10 進数の画面の高さ (行数で) を変数 SCRNHHT で戻し、3 桁の 10 進数の画面幅 (列数で) を変数 SCRNWWD で戻します。

►► SCRNINFO ◄◄

戻りコード

- n***
エラーが検出された場合に CICS によって戻される戻りコード。
- 0**
Normal return
- 499**
内部エラー

例

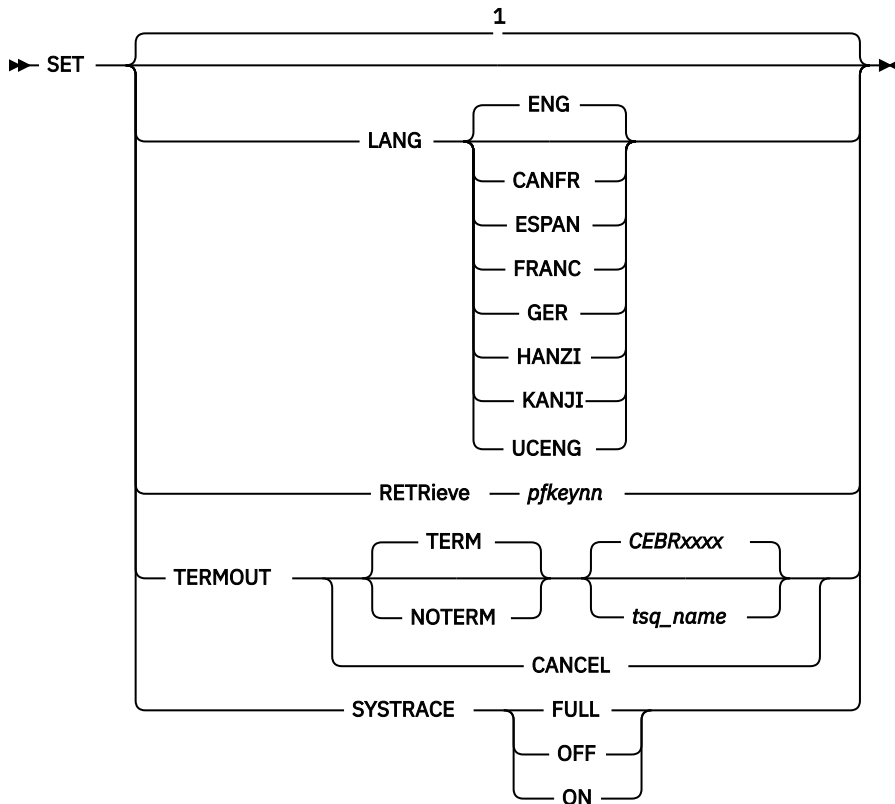
```
'SCRNINFO'
```

注:

- 戻される画面情報は、EXEC CICS ASSIGN SCRNHHT(*scrnhht*) SCRNWWD(*scrnwwd*) を実行すると取得されます。
- 接続されている端末がない場合、画面の高さおよび画面幅の値は 0 です。

SET

SET は現行ユーザー用の REXX/CICS 処理オプションを設定します。必要なすべての SET コマンドをユーザー・プロファイル EXEC に配置することをお勧めします。



注:

¹ SET コマンドにパラメーターを渡さない場合、SET は、SET、SETSYS、または PATH コマンドでユーザーのために作成されたすべての処理オプションが含まれている語幹変数 (SET.) を作成します。

オペランド

LANG

REXX ランタイム環境でメッセージおよび日付に使われる言語を指定します。

複数の REXX/CICS ユーザーが、同一領域内で同時に異なる言語を使用できます。

ENG

英語。これはデフォルトです。

CANFR

カナダ・フランス語

ESPAÑ

スペイン語

FRANC

フランス語

GER

ドイツ語

HANZI

中国語 (繁体字)

KANJI

漢字

UCENG

大文字の英語

RETRieve

最後に入力された行を取り出すための PF キーを指定します。

pfkeynn

PF キー番号を指定します。

TERMOUT

端末が接続されている場合でも、行モード出力を CICS 一時ストレージ・キュー (例えば、SAY 出力や TRACE 出力) に送信します。

CANCEL

行モード出力を CICS 一時記憶域キューに送信しないことを指定します。

NOTERM

端末に端末回線モード出力を表示しないことを指定します。

TERM

行モード出力を端末に送信することを指定します。

tsq_name

CICS 一時記憶域キューの名前を指定します。デフォルト *tsq_name* は CEBRxxxx であり、ここで、xxxx はご使用の端末 ID です。(TSQ 名を指定せずに CEBR トランザクションを入力した場合、これが、使用されるデフォルト名になります。)

SYSTRACE

REXX/CICS システム・トレースをオンにするかどうかを指定します。このオプションは、IBM サービス担当員の指示のもとでのみ使用してください。

REXX/CICS システム・トレースは、CICS ユーザー・トレースもアクティブな場合にのみ出力されます。

FULL

例外トレースに加えて、基本的な REXX/CICS システム・トレース (入り口および戻り) および補足トレースがオンであることを指定します。

OFF

REXX/CICS システム・トレースをオフにするように指定します。例外トレースのみがオンになります。

ON

例外トレースに加えて、基本的な REXX/CICS システム・トレースがオンであることを指定します。

戻りコード

0

Normal return

421

SET サブコマンドが無効です

422

変数の格納中にエラーが発生しました

423

言語が無効です

426

RETRIEVE PFkey オペランドが無効または欠落しています

427

TERMOUT オペランドが無効です

428

SYSTRACE オペランドが無効または欠落しています

例

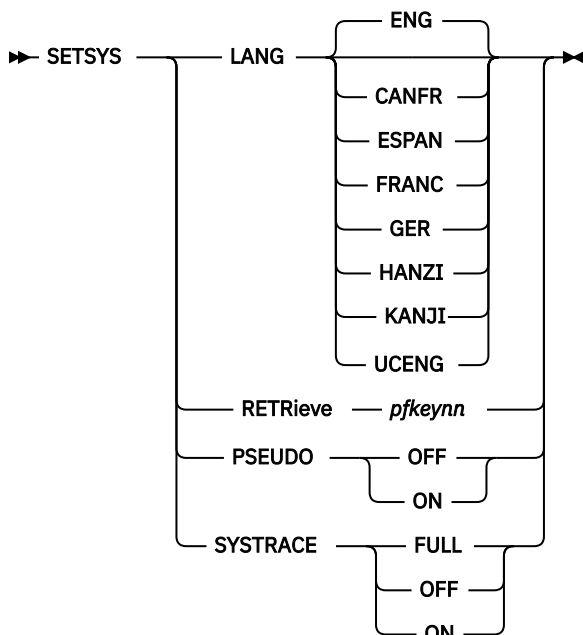
```
'SET TERMOUT TERM TSQ1'
```

この例は、端末行モード出力を一時記憶域キュー TSQ1 に送信する処理オプションを設定します。

SETSYS

SETSYS は、システムの REXX/CICS 処理オプションを設定します。システム全体にわたる SETSYS コマンドを CICSTART exec 内に入れることをお勧めします。

これは、許可コマンドです。



オペランド

LANG

REXX ランタイム環境でメッセージおよび日付に使われる言語を指定します。

ENG

英語。これはデフォルトです。

CANFR

カナダ・フランス語

ESPAÑ

スペイン語

FRANC

フランス語

GER

ドイツ語

HANZI

中国語 (繁体字)

KANJI

漢字

UCENG

大文字の英語

RETRieve

最後に入力された行を取り出すための PF キーを指定します。

pfkeynn

PF キー番号を指定します。

PSEUDO

デフォルトの領域全体にわたる REXX/CICS 自動疑似会話型設定を指定します。396 ページの『PSEUDO』を参照してください。

このオプションが以前に設定されていない場合は、自動疑似会話型設定がオンになります。

OFF

自動疑似会話型設定がオフであることを指定します。

ON

自動疑似会話型設定がオンであることを指定します。

SYSTRACE

REXX/CICS システム・トレースをオンにするかどうかを指定します。このオプションは、IBM サービス担当員の指示のもとでのみ使用してください。

REXX/CICS システム・トレースは、CICS ユーザー・トレースもアクティブな場合にのみ出力されます。このオプションが以前に設定されていない場合、REXX/CICS システム・トレースはオフになります。

FULL

例外トレースに加えて、基本的な REXX/CICS システム・トレース (入り口および戻り) および補足トレースがオンであることを指定します。

OFF

REXX/CICS システム・トレースをオフにするように指定します。例外トレースのみがオンになります。

ON

例外トレースに加えて、基本的な REXX/CICS システム・トレースがオンであることを指定します。

戻りコード

0

Normal return

2721

SETSYS サブコマンドが無効です

2722

変数の格納中にエラーが発生しました

2723

無効な言語

2726

RETRIEVE PFkey オペランドが無効または欠落しています

2728

SYSTRACE オペランドが無効または欠落しています

2732

PSEUDO オペランドが無効または欠落しています

例

```
'SETSYS RETRIEVE 12'
```

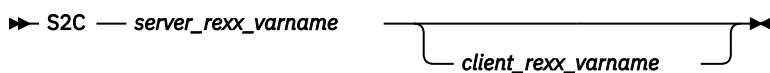
この例は、PF キー 12 を検索キーとして設定します。

```
SETSYS SYSTRACE ON
```

この例では、すべてのユーザーに関して基本的な REXX/CICS システム・トレースをオンに設定します。

S2C

S2C は、サーバー REXX 変数をクライアント REXX 変数にコピーします。



オペランド

server_rexx_varname

コピー元のサーバー REXX 変数の名前です。

client_rexx_varname

コピー先のクライアント REXX 変数のオプションの名前です。これを指定しない場合、デフォルトで **server_rexx_varname** と同じものに設定されます。

戻りコード

0

Normal return

2840

変数名が指定されていません

2841

変数の取得中にエラーが発生しました

2842

変数の格納中にエラーが発生しました

2848

使用可能なクライアントがありません

例

```
'S2C VARA VARB'
```

この例は、クライアント REXX 変数 VARB へのサーバー REXX 変数 VARA コピーの内容を示しています。VARB の長さは VARA の長さと同じです。

注：

1. このコマンドで許可される *varname* の最大長は 250 文字です。それより長い名前が指定された場合には、最初の 250 文字だけが使用されます。
2. S2C によってコピーされる変数名の最大長は 6000 バイトです。
3. S2C コマンドは、サーバー EXEC からのみ使用できます。

TERMID

TERMID は、CICS フィールド EIBTRMID からの 4 文字の CICS 端末 ID を変数 TERMID で戻します。

➡ TERMID ➡

戻りコード

0

Normal return

2921

端末 ID の取得中にエラーが発生した

2928

TERMID 値の設定中にエラーが発生した

例

```
'TERMID'
```

この例は、CICS フィールド EIBTRMID からの CICS 端末 ID を変数 TERMID に入れます。

WAITREAD

WAITREAD は、フルスクリーン端末入力を実行し、その結果を複合変数に入れます。

▶▶ WAITREAD ◀◀

複合変数内の結果は次のようになります。

WAITREAD.0

戻されたエレメントの数が含まれます。

WAITREAD.1

AID 記述が含まれます。

WAITREAD.2

カーソル位置が含まれます。

WAITREAD.3 から WAITREAD まで。n

変更された、残りの 3270 フィールドが含まれます。

戻りコード

0

Normal return

3021

端末が接続されていません

3099

内部エラー

例

```
'WAITREAD'
```

この例は、端末画面から入力を読み取り、その結果を、REXX 複合変数 WAITREAD.1 から WAITREAD に入れます。n.

注: 変更読み取りが端末に対して実行されると、この読み取りからの情報と一緒に配列が戻されます。これらのエレメントの形式は、*field_row field_column data* です。

WAITREQ

WAITREQ は、サーバーが要求を待機する原因となる REXX サーバーでのみ使用されます。要求は、受信された後、REXX 変数 REQUEST に入られます。

▶▶ WAITREQ ◀◀

戻りコード

0

Normal return

3121

WAITREQ が使用不可です

3122

EXEC がサーバーではありません

3123

要求変数の保管中にエラーが発生した

3199

内部エラー

クライアント・プログラムに示される戻りコードは、WAITREQ コマンドへの入り口またはサーバー EXEC の出口での REXX サーバー変数の値です。

例

```
'WAITREQ'
```

この例は、サーバーに対する別の要求が検出されるまで、サーバー EXEC を中断させます。要求が検出されると、サーバー EXEC は、その中断前の状況に、新しい要求値で復元されます。

第 31 章 エラー番号とメッセージ

エラー・コードとそれに関連する CICS メッセージをリストしています。

言語処理プログラムに対する外部インターフェースは、言語処理プログラムに制御を渡す前、または言語処理プログラムから制御が戻された後に、3つのエラー・メッセージを生成する可能性があります。したがって、以下のエラーを SIGNAL ON SYNTAX でトラップすることはできません。

- 3 および 5 (ストレージに関する初期要件が満たされなかった場合)
- 26 (出口において、戻されたストリングを有効な戻りコード形式に変換することができなかった場合)

エラー 4 をトラップできるのは、SIGNAL ON HALT または CALL ON HALT だけです。

言語処理プログラムが検出する以下の 5 つのエラーを、SIGNAL ON SYNTAX によってトラップすることはできません。ただし、プログラム内において、エラーのある文節よりも SYNTAX ラベルが先にある場合を除きます。

- 6
- 12
- 13
- 22
- 30

表 5. エラー・コードと CICS メッセージ

エラー・コード	CICS メッセージ	エラー・コード	CICS メッセージ
番号なし	CICREX255T	エラー 26	CICREX466E
エラー 3	CICREX451E	エラー 27	CICREX467E
エラー 4	CICREX452E	エラー 28	CICREX486E
エラー 5	CICREX450E	エラー 29	CICREX487E
エラー 6	CICREX453E	エラー 30	CICREX468E
エラー 7	CICREX454E	エラー 31	CICREX469E
エラー 8	CICREX455E	エラー 32	CICREX492E
エラー 9	CICREX456E	エラー 33	CICREX488E
エラー 10	CICREX457E	エラー 34	CICREX470E
エラー 11	CICREX458E	エラー 35	CICREX471E
エラー 12	CICREX459E	エラー 36	CICREX472E
エラー 13	CICREX460E	エラー 37	CICREX473E
エラー 14	CICREX461E	エラー 38	CICREX489E
エラー 15	CICREX462E	エラー 39	CICREX474E
エラー 16	CICREX463E	エラー 40	CICREX475E
エラー 17	CICREX465E	エラー 41	CICREX476E
エラー 18	CICREX491E	エラー 42	CICREX477E
エラー 19	CICREX482E	エラー 43	CICREX478E
エラー 20	CICREX483E	エラー 44	CICREX479E

表 5. エラー・コードと CICS メッセージ (続き)

エラー・コード	CICS メッセージ	エラー・コード	CICS メッセージ
エラー 21	CICREX464E	エラー 45	CICREX480E
エラー 22	CICREX449E	エラー 46	CICREX218E
エラー 23	CICREX1106E	エラー 47	CICREX219E
エラー 24	CICREX484E	エラー 48	CICREX490E
エラー 25	CICREX485E	エラー 49	CICREX481E

これらのメッセージでは、用語「言語処理プログラム」は、REXX/CICS インタープリターを指します。

以下のセクションのエラー・メッセージのほかに、言語処理プログラムは端末 (リカバリー不能) メッセージ 412 ページの『CICREX255T』を発行します。

CICREXnnn エラー・メッセージ

CICREX218E Error 46 Invalid variable reference

説明

ARG、DROP、PARSE、PULL、または PROCEDURE の各命令内で、変数参照 (名前が括弧で囲まれており、使用される値を持つ変数) の構文が正しくありません。変数名の直後にあるべき右括弧が欠落している可能性があります。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

CICREX219E Error 47 Unexpected label

説明

INTERPRET 命令について評価中の式、または対話式デバッグの過程で入力された式の中に、使用法が正しくないラベルが検出されました。

システムの処置

実行は停止します。

ユーザーの処置

これらの式の中ではラベルを使用しないでください。

CICREX255T Insufficient storage for Exec interpreter

説明

言語処理プログラムそのものを初期設定するためのストレージが不足しています。

システムの処置

実行はエラーの発生時点で終了されます。

ユーザーの処置

ストレージを再定義して、コマンドを再発行してください。

CICREX449E Error 22 running *fn ft*, line *nn*: Invalid character string

説明

OPTIONS ETMODE を有効にしてスキャンされた文字ストリングで、以下のいずれかがあてはまります。

- シフトアウト (SO) 制御文字とシフトイン (SI) 制御文字が対応していない
- シフトアウト (SO) 文字とシフトイン (SI) 文字の間のバイト数が奇数である

システムの処置

実行は停止します。

ユーザーの処置

EXEC ファイル内の誤りのある文字ストリングを訂正します。

CICREX450E Error 5 running *fn ft*, line *nn*: User storage exhausted or request exceeds limit

説明

プログラムを処理しようとしたましたが、続行するために必要なリソースを言語処理プログラムが取得できませんでした。(例えば、作業域や変数に必要なスペースを取得できませんでした。) 言語処理プログラムを呼び出したプログラムが使用可能なストレージをほとんど使

い尽くしたか、または、実装上の最大値を超えるストレージ要求が行われた可能性があります。

システムの処置

実行は停止します。

ユーザーの処置

プログラム自体で EXEC またはマクロを実行するか、または、NUCXLOAD を出して、適切に終了せずにループしている可能性がないかプログラムを検査します。追加のストレージ要件については、システム管理者に問い合わせてください。

CICREX451E Error 3 running *fn ft*: Program is unreadable

説明

REXX プログラムを読み取ることができませんでした。これは、EXEC ファイル内のデータに問題があるか、または入出力エラーによるものと考えられます。

システムの処置

実行は停止します。

ユーザーの処置

EXEC ファイルを調べて、訂正します。

注: REXX EXEC 内の列 73 から列 80 では、シーケンス番号は使用できません。

CICREX452E Error 4 running *fn ft*, line *nn*: Program interrupted

説明

システムは、ユーザーの REXX プログラムの実行を中断しました。この状態は、特定のユーティリティー・モジュールが重大なエラー状態を検出した場合に強制される可能性があります。

システムの処置

実行は停止します。

ユーザーの処置

ユーザーの EXEC またはマクロで呼び出されたユーティリティー・モジュールに問題がないか調べます。

CICREX453E Error 6 running *fn ft*, line *nn*: Unmatched /* or quote

説明

コメントまたはリテラル・ストリングが開始されていますが、終了していません。これは、言語処理プログラムが以下のことを検出したためと考えられます。

- コメントの終わり「*/」またはリテラル・ストリングの終了引用符を見つけることなく、ファイルの終わり(または、INTERPRET ステートメント内のデータの終わり)を検出しました。
- リテラル・ストリングの行の終わりを検出しました。

システムの処置

実行は停止します。

ユーザーの処置

EXEC を編集し、結びの「*/」または引用符を追加します。プログラムの初めに TRACE SCAN ステートメントを挿入して、再実行することもできます。結果として生じる出力に、エラーがどこに存在するかが示されます。

CICREX454E Error 7 running *fn ft*, line *nn*: WHEN or OTHERWISE expected

説明

言語処理プログラムでは、SELECT ステートメント内に一連の WHEN と 1 つの OTHERWISE があるものと予想しています。このメッセージは、他の命令が検出されたか、あるいは、すべての WHEN 式が偽であり、OTHERWISE が存在しないことが検出された場合に出されます。このエラーは、WHEN に続く命令のリストの前後に DO 命令と END 命令を入れ忘れたことが原因である場合が少なくありません。以下に例を示します。

WRONG	RIGHT
Select When a1=b1 then Say 'A1 equals B1' B1' exit Otherwise nop end	Select When a1=b1 then DO Say 'A1 equals exit end Otherwise nop end

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

CICREX455E Error 8 running *fn ft*, line *nn*: Unexpected THEN or ELSE

説明

言語処理プログラムは、対応する IF 文節に適合しない THEN または ELSE を検出しました。この状況は、複合 IF-THEN-ELSE 構造の THEN 部分に間違った DO-END を使用したことが原因で起こる場合が少なくありません。その例を次に示します。

WRONG	RIGHT
If a1=b1 then do; Say EQUALS exit else Say NOT EQUALS	If a1=b1 then do; Say EQUALS exit end else Say NOT EQUALS

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX456E Error 9 running *fn ft*, line *nn*:
Unexpected WHEN or OTHERWISE**

説明

言語処理プログラムは、SELECT 構造の外側で WHEN 命令または OTHERWISE 命令を検出しました。END 命令をはずしたままにしたことにより DO-END 構造内でこの命令を誤って囲んでしまったか、あるいは SIGNAL ステートメントを使用してこの END 命令にブランチしようとした(その時、SELECT が終了しているため、これは機能しません)可能性があります。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX457E Error 10 running *fn ft*, line *nn*:
Unexpected or unmatched END**

説明

言語処理プログラムが、プログラム内で END が DO または SELECT よりも多いことを検出したか、あるいは DO または SELECT に対応していない END を検出しました。反復ループを終了させる END に制御変数の名前を入れると、この種のエラーを突きとめるのに役立ちます。

このメッセージは、ユーザーがループの中にシグナルを出そうとした場合に出されることがあります。この場合、先行する DO が実行されていないために、END は予想されません。また、SIGNAL は現行ループを終了させるので、SIGNAL を使用してループ内のある場所から別の場所に制御を移すことはできないことに留意してください。

このメッセージは、THEN 構造または ELSE 構造の直後に END を置いた場合、または DO の後の *name* に一致しない *name* を END キーワードに指定した場合にも出されます。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。TRACE スキャンを使用してプログラムの構造を表示すると、エラーが見つけやすくなる場合があります。反復ループを終了させる END に制御変数の名前を入れると、この種のエラーを突きとめるのに役立ちます。

**CICREX458E Error 11 running *fn ft*, line *nn*:
Control stack full**

説明

このメッセージは、250 という、制御構造 (DO-END、IF-THEN-ELSE など) のネスティング・レベル制限を超えたか、ユーザー・ストレージ制限に達したかのいずれか少ない方の限界に該当したときに outされます。

このメッセージは、次のような INTERPRET 命令のループが原因で出されることがあります。

```
line='INTERPRET line'
INTERPRET line
```

上記の行はネスティング・レベルの限界を超えるまでループし、その後、このメッセージが出されます。同様に、正しく終了しない再帰的サブルーチンがループして、このメッセージが出されることもあります。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX459E Error 12 running *fn ft*, line *nn*:
Clause too long**

説明

文節の内部表記の長さの制限を超えました。実際の限界は、単一の要求で取得できるストレージの量です。

このメッセージが出された原因が明らかでない場合は、引用符が欠落していたために複数の行が 1 つの長いストリングになったことが原因である可能性があります。その場合は、おそらく、文節トレースバックに含まれているデータの始め (コンソールに +++ フラグ付きで表示されます) でエラーが発生しました。

文節の内部表記には、ストリングの外側にあるコメントまたは複数ブランクは含まれません。また、シンボル (名前) またはストリングはいずれも、内部表記で 2 文字の長さを使用することにも注意してください。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX460E Error 13 running *fn ft*, line *nn*:
Invalid character in program**

説明

言語処理プログラムが、リテラル (引用符付き) スtringの外側で無効な文字を検出しました。有効な文字は次のとおりです。

- A-Z a-z 0-9 (英数字)
- @ # \$ % . ? ! _ (名前の文字)
- & * () - + = \ ~ ' " ; < , > / | (特殊文字)

X'0E' (シフトアウト) と X'0F' (シフトイン) で囲まれており、ETMODE がオンの場合は、以下の文字も有効です。

- X'41' - X'FE' (DBCS 文字)

このエラーの原因のなかには、次のものがあります。

1. アクセント記号付き文字やその他の言語特有の文字が、シンボルに使用されている。
2. ETMODE を有効にせずに DBCS 文字を使用している。

注: 文字 @ # \$ % は、エミュレーター構成で使用されているコード・ページによっては、REXX オンライン・ヘルプで異なる文字として表示される場合があります。CICS 資料で使用されている表記規則および用語も参照してください。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX461E Error 14 running *fn ft*, line *nn*:
Incomplete DO/SELECT/IF**

説明

言語処理プログラムはファイルの終わり (または INTERPRET 命令の場合はデータの終わり) に到達し、検出した DO または SELECT に一致する END がないか、検出した IF の後に THEN 文節が続いていません。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。TRACE Scan を使用してプログラムの構成を示すことで、END または THEN を入れるべき個所を簡単に見つけられます。反復ループを終了させる END に制御変数の名前を入れると、この種のエラーを突きとめるのに役立ちます。

**CICREX462E Error 15 running *fn ft*, line *nn*:
Invalid hexadecimal or binary string**

説明

2 進ストリングは REXX では新規に採用されたものであるため、言語処理プログラムがステートメント内の該当ストリングを、ユーザーの意図ではないのに 2 進数と見なしている可能性があります。

言語処理プログラムの場合は、16 進ストリングに先行または末尾のブランクを含めることはできず、組み込みブランクを入れられるのはバイト境界だけです。使用できるのは、0 から 9 までの数字と、a から f および A から F までの文字のみです。同様に、2 進ストリングにブランクを入れられるのは 4 つの 2 進数字のグループの境界だけであり、数字の 0 と 1 しか使用できません。

以下は、有効な 16 進定数または 2 進定数です。

```
'13'x                    '0101 1100'b  
'A3C2 1c34'x           '001100'B  
'1de8'x                "0 11110000"b
```

たとえば、0 の代わりに文字 o を入力するなど、数字の 1 つを誤って入力した可能性があります。あるいは、ストリングを 16 進または 2 進として指定するつもりではないのに、1 文字の記号 X、x、B、または b (それぞれ、変数 X または B の名前) を、リテラル・ストリングの後にに入れてしまったということも考えられます。この場合は、明示の連結演算子 (||) を使用して、ストリングを記号の値に連結してください。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX463E Error 16 running *fn ft*, line *nn*:
Label not found**

説明

言語処理プログラムは、SIGNAL 命令によって指定されたラベル、または対応する (トラップされた) イベントの発生時の使用可能条件に一致するラベルを見つけられませんでした。ラベルを誤って入力したか、それを

含めるのを忘れたか、あるいは大文字にする必要があるのに大文字小文字混合で入力した場合が考えられます。

システムの処置

実行を停止します。脱落したラベルの名前は、エラー・トレースバックに入ります。

ユーザーの処置

必要な訂正を行います。

**CICREX464E Error 21 running *fn ft*, line *nn*:
Invalid data on end of clause**

説明

SELECT または NOP などの文節の後に、コメント以外のデータが続いています。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX465E Error 17 running *fn ft*, line *nn*:
Unexpected PROCEDURE**

説明

言語処理プログラムは、PROCEDURE 命令を正しくない位置に検出しました。この原因としては、アクティブな内部ルーチンがない、内部ルーチン内で PROCEDURE 命令がすでに出てきた、あるいは、CALL または関数呼び出しの後で最初に実行される命令が PROCEDURE 命令でない、のいずれかが考えられます。このエラーは、CALL または関数呼び出しによって内部ルーチンが呼び出されたためではなく、内部ルーチンを呼び出すまでの個所が"全体的に脱落"しているために起きた可能性があります。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX466E Error 26 running *fn ft*, line *nn*:
Invalid whole number**

説明

言語処理プログラムは、NUMERIC 命令、解析定位置パターン、または累乗 (**) 演算子の右側の項の中に、計算結果が整数にならない式、あるいはこれらの個所で使

用する場合の限度である 999999999 を超えている式を検出しました。

EXIT または RETURN 命令 (REXX プログラムがコマンドとして呼び出される場合) から戻される戻りコードが整数でない場合、または汎用レジスターに収まらない場合にも、このメッセージが出ることがあります。このエラーの原因として、シンボルの名前を誤って入力したために、該当するステートメントの式の変数名になっていないことが考えられます。たとえば、"EXIT RC"ではなく"EXIT CR"を入力すると、このエラーになります。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX467E Error 27 running *fn ft*, line *nn*:
Invalid DO syntax**

説明

言語処理プログラムは、DO 命令に構文エラーを検出しました。BY、TO、FOR、WHILE、OR UNTIL を 2 回使用しているか、WHILE と UNTIL を使用しています。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX468E Error 30 running *fn ft*, line *nn*:
Name or string > 250 characters**

説明

言語処理プログラムは、最大長を超えた変数またはリテラル (引用符で囲んだ) スtringを検出しました。

置換を行った後の名前の長さの最大値は、250 文字です。このエラーの原因として考えられるのは、名前内でピリオド (.) を使用したために、予期しない置換が発生したことです。

リテラル・Stringの最大の長さは 250 文字です。終了の引用符がない (またはString内に単一引用符がある) ために、数個の文節が 1 つのStringと見なされた結果、このエラーになった可能性があります。たとえば、Stringの 'don't' は、'don' 't' または "don't" と書いてください。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX469E Error 31 running *fn ft*, line *nn*:
Name starts with number or .**

説明

言語処理プログラムは、名前が数字またはピリオド (.) で始まる記号を検出しました。REXX 言語の規則では、名前が数字またはピリオドで始まる記号に値を割り当てることはできません。このような名前を使用すると、数値定数を再定義できることになり、致命的なエラーとなるためです。

システムの処置

実行は停止します。

ユーザーの処置

変数の名前を正しく付け直します。変数名は、英字で始めるのが最適ですが、それ以外の文字を使用してもかまいません。

**CICREX470E Error 34 running *fn ft*, line *nn*:
Logical value not 0 or 1**

説明

言語処理プログラムが、IF 句、WHEN 句、DO WHILE 句、または DO UNTIL 句で、結果が 0 にも 1 にもならない式を検出しました。論理演算子 (¬、\、|、&、または &&) によって演算された値はすべて、0 または 1 にならなければなりません。例えば、「If result then exit rc」という句は、result の値が 0 でも 1 でもなければ失敗します。したがって、この句は If result¬=0 then exit rc と書いたほうが適切です。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX471E Error 35 running *fn ft*, line *nn*:
Invalid expression**

説明

言語処理プログラムは、式の中に文法エラーを検出しました。これには、以下の理由が考えられます。

- 演算子で式を終了した。
- 式内に 2 つの演算子を、間になにも指定せずに続けて指定した。
- 必要な場所に式を指定しなかった。

- 必要な場所に右括弧を指定しなかった。
- 文字式の内部に特殊文字 (演算子など) を使用したが、その特殊文字を引用符で囲んでいない。

最後に示した場合は、たとえば、LISTFILE *** を、LISTFILE '* * *'(LISTFILE が変数でない場合) と書くか、あるいは 'LISTFILE * * *' と書くようにしてください。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX472E Error 36 running *fn ft*, line *nn*:
Unmatched (in expression**

説明

言語処理プログラムは、式の中でペアになっていない括弧を検出しました。このメッセージは、コマンドで単一の括弧を使用し、それを引用符で囲まなかった場合に発生します。例えば、COPY A B C A B D (REP は COPY A B C A B D '(' REP と記述します。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX473E Error 37 running *fn ft*, line *nn*:
Unexpected , or)**

説明

言語処理プログラムは、ルーチン呼び出しの外側にコンマ (,) があること、または式の中に右括弧が多過ぎることを検出しました。このメッセージは、文字式でコンマを使用し、それを引用符で囲まなかった場合に発生します。例えば、次の命令を考えてみましょう。

```
Say Enter A, B, or C
```

これは次のように書くべきです。

```
Say 'Enter A, B, or C'
```

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX474E Error 39 running *fn ft*, line *nn*:
Evaluation stack overflow**

説明

式が複雑過ぎたために言語処理プログラムが式を評価できませんでした (ネストされた括弧や関数が多いなど)。

システムの処置

実行は停止します。

ユーザーの処置

副次式を一時変数に割り当てることで、式を分割します。

**CICREX475E Error 40 running *fn ft*, line *nn*:
Invalid call to routine**

説明

言語処理プログラムは、ルーチンの呼び出し方が正しくないものを検出しました。考えられる原因として、以下のものがあります。

- 正しくないデータ (引数) を組み込みルーチンまたは外部ルーチンに渡しました (このどちらかは、実際のルーチンによって決まります)。
- 組み込みルーチン、外部ルーチン、または内部ルーチンに渡した引数が多すぎます。
- 呼び出されたモジュールが言語処理プログラムと互換性がありませんでした。

ルーチンを呼び出そうとしていたのではないのに、シンボルまたはストリングをスペースまたは演算子で切り離すつもりで、シンボルまたはストリングの隣に括弧 (を指定した可能性があります。この書き方は、関数呼び出しと見なされます。たとえば、TIME*(4+5) と書くべきところを、TIME(4+5) としているなどです。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX476E Error 41 running *fn ft*, line *nn*: Bad
arithmetic conversion**

説明

言語処理プログラムは、有効な数値ではない項または許容範囲 (-999999999 から +999999999 まで) 外の指数を使用した項を、算術式から検出しました。

変数名を誤って入力した可能性があります。つまり、算術演算子を、引用符で囲まずに文字式の中に書いていま

す。たとえば、コマンド MSG * Hi! は、'MSG * Hi!' と書く必要があります。そうしないと、言語処理プログラムは、"MSG" と "Hi!" の乗算を行おうとします。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX477E Error 42 running *fn ft*, line *nn*:
Arithmetic overflow/underflow**

説明

言語処理プログラムは、算術演算の結果を表すのに 9 桁という限度よりも大きい (999999999 より大きいか -999999999 より小さい) 指数が必要となるものを検出しました。

このエラーが起こるのは、式の計算中 (たいていは、数値の 0 による除算を行った場合) か、あるいは DO ループ制御変数の処理中です。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX478E Error 43 running *fn ft*, line *nn*:
Routine not found**

説明

言語処理プログラムは、プログラム内で呼び出されたルーチンを見つけることができませんでした。式の中、または CALL によって呼び出されたサブルーチン内で関数を呼び出しましたが、指定されたラベルがプログラム内にないか、あるいは組み込み関数の名前ではないため、REXX/CICS が外部からそのラベルを見つけられません。

このエラーの原因で最も単純、かつ最も一般的なものは、名前への入力誤りです。この他の可能性として、いずれかの標準関数パッケージが使用不能である場合も考えられます。

ルーチンの呼び出しをしようとしていなかった場合は、記号またはストリングをスペースまたは演算子で分けるつもりのところを、その記号またはストリングに隣接して "(" を書いたことが考えられます。言語処理プログラムは、それを関数呼び出しと見なします。たとえば、3*(4+5) と書くはずのものをストリングの 3(4+5) としています。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX479E Error 44 running *fn ft*, line *nn*:
Function did not return data**

説明

言語処理プログラムは、式の内部から外部ルーチンを呼び出しました。このルーチンは、エラーにならずに終了したように見えますが、式の中で使用するデータを戻しませんでした。

この原因として、REXX 関数として使用することが意図されていないモジュールの名前を指定したことが考えられます。このモジュールは、コマンドまたはサブルーチンとして呼び出すようにしてください。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX480E Error 45 running *fn ft*, line *nn*: No
data specified on function RETURN**

説明

REXX プログラムが関数として呼び出されましたが、データを戻さずに (RETURN; 命令によって) 戻ろうとしています。同様に、関数として呼び出された内部ルーチンは、式を指定した RETURN ステートメントによって終了しなければなりません。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX481E Error 49 running *fn ft*, line *nn*:
Language processor failure**

説明

言語処理プログラムは、多くの内部自己整合性検査を実行します。このメッセージが出るのは、重大エラーが検出された場合です。

システムの処置

実行は停止します。

ユーザーの処置

このメッセージが現れた場合は、IBM 担当者に連絡してください。

**CICREX482E Error 19 running *fn ft*, line *nn*:
String or symbol expected**

説明

言語処理プログラムは、CALL 命令または SIGNAL 命令の後に記号を予定していますが、何も検出されませんでした。ストリングまたは記号を省略したか、あるいは特殊文字 (括弧など) をストリングまたは記号に挿入した可能性があります。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX483E Error 20 running *fn ft*, line *nn*:
Symbol expected**

説明

言語処理プログラムは、CALL ON、CALL OFF、END、ITERATE、LEAVE、NUMERIC、PARSE、PROCEDURE、SIGNAL ON、または SIGNAL OFF キーワードの後に記号を期待しているか、DROP、UPPER、または (EXPOSE オプションを指定した) PROCEDURE キーワードの後に記号のリストまたは変数参照を期待しているのかの、いずれかです。あるべき個所に記号がなかったか、あるいは他の文字が検出されました。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX484E Error 24 running *fn ft*, line *nn*:
Invalid TRACE request**

説明

言語処理プログラムは、次の場合にこのメッセージを出します。

- TRACE 命令に指定された処置または TRACE 組み込み関数への引数において、1 文字目が有効な英字オプションのいずれかに一致しない場合。有効なオプションは、A、C、E、F、I、L、N、O、R、または S です。
- いずれかの制御構造の内部から、または対話式デバッグの間に、TRACE Scan 要求が出された場合。

- 対話式トレースにおいて、整数以外の数値を入力した場合。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX485E Error 25 running *fn ft*, line *nn*:
Invalid sub-keyword found**

説明

言語処理プログラムが命令の中で特定のサブキーワードを予定している位置に、他のものが検出されました。たとえば、NUMERIC 命令の後にはサブキーワードの DIGITS、FUZZ、または FORM を続けなければなりません。NUMERIC の後に他のものが続いていると、このメッセージが出されます。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX486E Error 28 running *fn ft*, line *nn*:
Invalid LEAVE or ITERATE**

説明

言語処理プログラムは、正しくない LEAVE または ITERATE 命令を検出しました。以下のいずれかに該当したため、命令が正しくありませんでした。

- アクティブなループがない。
- 命令で指定されている名前が、アクティブ・ループの制御変数と一致しない。

内部ルーチン呼び出しおよび INTERPRET 命令を実行すると、DO ループが非アクティブになるため、それらの DO ループが保護されます。したがって、たとえば、サブルーチン内で LEAVE 命令を実行しても、呼び出しルーチン内の DO ループには影響しません。

ループの内部で制御を転送する場合、またはループ内に制御を転送する場合に SIGNAL 命令を使用すると、このメッセージが表示されることがあります。SIGNAL 命令は、すべてのアクティブ・ループを終了させるので、その後 ITERATE 命令または LEAVE 命令を出すと、このメッセージが出されます。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX487E Error 29 running *fn ft*, line *nn*:
Environment name too long**

説明

言語処理プログラムが検出した ADDRESS 命令に指定されている環境名が、8 文字という限度を超えています。

システムの処置

実行は停止します。

ユーザーの処置

環境名を正しく指定します。

**CICREX488E Error 33 running *fn ft*, line *nn*:
Invalid expression result**

説明

言語処理プログラムは、式の結果がその特定の文脈において正しくないことを検出しました。以下のいずれかの結果が誤りである可能性があります。

- ADDRESS VALUE 式
- NUMERIC DIGITS 式
- NUMERIC FORM VALUE 式
- NUMERIC FUZZ 式
- OPTIONS 式
- SIGNAL VALUE 式
- TRACE VALUE 式

(FUZZ は、DIGITS より小さくなければなりません。)

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX489E Error 38 running *fn ft*, line *nn*:
Invalid template or pattern**

説明

言語処理プログラムが構文解析テンプレート内に正しくない特殊文字(%など)を検出したか、あるいは変数トリガーの構文が正しくありませんでした(左括弧の後に記号が見つかりませんでした)。このメッセージは、PARSE VALUE 命令から WITH サブキーワードが省略されている場合にも出ます。

システムの処置

実行は停止します。

ユーザーの処置

必要な訂正を行います。

**CICREX490E Error 48 running *fn ft*, line *nn*:
Failure in system service**

説明

言語処理プログラムは、ユーザー入出力やコンソール・スタックの操作などの何らかのシステム・サービスが正しく動作しなかったために、プログラムの実行を停止します。

システムの処置

実行は停止します。

ユーザーの処置

ユーザーの入力およびプログラムの動作が正しいか確認してください。問題が解決されない場合には、システム・サポート担当員に連絡してください。

**CICREX491E Error 18 running *fn ft*, line *nn*:
THEN expected**

説明

REXX のすべての IF 文節および WHEN 文節の後には、THEN 文節を続けなければなりません。THEN ステートメントの検出前に、別の文節が検出されました。

システムの処置

実行は停止します。

ユーザーの処置

IF または WHEN 文節と後続の文節の間に、THEN 文節を挿入してください。

**CICREX492E Error 32 running *fn ft*, line *nn*:
Invalid use of stem**

説明

REXX プログラムが記号の値を変更しようとしたが、その記号は語幹です。(語幹とは、記号の最初のピリオドまでの部分です。語幹を使用するのは、その語幹で始まるすべての変数进行处理する場合です。) これは、UPPER 命令内で行われた可能性があります。この場合の動作は、不明なのでエラーになります。

システムの処置

実行は停止します。

ユーザーの処置

語幹の値を変更しないように、プログラムを変更します。

**CICREX1106E Error 23 running *fn ft*, line *nn*:
Invalid SBCS/DBCS mixed string.**

説明

有効な OPTIONS EXMODE で処理された文字ストリングで、SO-SI がペアになっていない (つまり、SO だけで SI がいない) か、または SO-SI 文字の間が奇数バイトになっています。

システムの処置

実行は停止します。

ユーザーの処置

正しくない文字ストリングを訂正します。

第 32 章 戻りコード

REXX/CICS 戻りコードを示します。

コマンドは、436 ページの『[特定のコマンドに関連付けられていない戻りコード](#)』にリストされている戻りコードを返す場合もあります。

パネル機能の戻りコード

- 4 警告。パネル機能は処理を続行します
- 8 プログラマー・エラー
- 10 状態情報のあるプログラマー・エラー
- 12 CICS コマンド・エラー
- 14 RFS エラー。理由コードに RFS 戻りコードが含まれています
- 16 内部システム・エラー

追加コード (例えば、状態コードや理由コード) については、323 ページの『[第 29 章 REXX/CICS パネル機能](#)』を参照してください。

SQL 戻りコード

- n* SQL ステートメントの結果としてエラーまたは警告が発生した場合は SQLCODE
- 0 SQL ステートメントは、EXECSQL 環境によって処理されました
- 30 SQLDSECT 変数を作成するための十分なメモリーがありませんでした
- 31 SQL ステートメント域を作成するための十分なメモリーがありませんでした
- 32 SQLDA 変数を作成するための十分なメモリーがありませんでした
- 33 SELECT ステートメント用の結果領域を作成するための十分なメモリーがありませんでした

Db2 戻りコード

- n* Db2 計測機能インターフェース (IFI) の呼び出しの結果を表す正の値。Db2 IFI からの RC がゼロでない場合、REXX 変数 DB2_RC2 には、Db2 IFI 理由コードが含まれます。DB2_RC2 値をこの RC と一緒に使用して、[z/OS 製品資料内の『Db2 の Db2 コード』](#)でエラー判別を行います。
- 0 Db2 コマンドは Db2 IFI によって処理されました。
- 50 指定された Db2 コマンドは、短すぎるか長すぎるために Db2 IFI で処理できません。Db2 コマンドの長さは、6 文字以上かつ 4092 文字以下にする必要があります。

- 51** Db2 IFI 用の出力域を作成するための十分なメモリーがありませんでした。
- 52** Db2 IFI 用の通信域を作成するための十分なメモリーがありませんでした。
- 53** Db2 IFI 用の戻り域を作成するための十分なメモリーがありませんでした。
- 54** DB2_RC2 REXX 変数を作成できません。
- 55** DB2_BNM REXX 変数を作成できません。
- 56** DB2_OUTPUT.*n* REXX 変数を作成できません。
- 57** DB2_OUTPUT.0 REXX 変数を作成できません。

RFS および FLST

- 0** Normal return
- 101** 無効なコマンド
- 102** 無効なオペランド
- 103** ファイルが見つかりません
- 104** 許可されません
- 105** ファイルは既に存在します
- 107** ファイル・プールのスペースが不十分です
- 110** 要求は失敗しました
- 111** ファイル ID が無効です
- 113** ディレクトリーが見つかりません
- 115** ディレクトリーは既に存在しています
- 116** ディレクトリーが指定されていません
- 121** ファイルが破損しています
- 122** stem.0 は無効または範囲外です
- 126** パス・エラー
- 127** CICS 入出力エラー

128
この場所からのコマンドは無効です

130
Directory not empty

131
オペランドが欠落しています

132
ファイル・プール・データ・レコードが欠落しています。ファイル・プールがフォーマットされていない可能性があります。

199
内部エラー

EDITOR および EDIT

0
Normal return

201
無効なコマンド

202
無効なオペランド

203
ファイルが見つかりません

204
許可されません

207
ファイル・プールのスペースが不十分です

210
要求は失敗しました

211
ファイル ID が無効です

223
検索引数が見つかりません

226
ファイルは現在、編集中です

229
範囲外の数値

230
カーソルがファイル域内にありません

231
仮想記憶域不足

232
接頭部コマンドが矛盾しています

236
Not defined

299
内部エラー

1748
Task Input/Output Table (TIOT) に DD 名のエントリーがありません

1749
複合データ・セットにはエクスポートできません

1750

DD 名として複数のデータ・セットが連結されています

DIR

0

Normal return

321

現行 RFS ディレクトリー情報にアクセスできない

322

無効な語幹名

325

RFS ディレクトリーの取得中にエラーが発生した

SET

0

Normal return

421

SET サブコマンドが無効です

422

変数の格納中にエラーが発生しました

423

無効な言語

426

RETRIEVE PFkey オペランドが無効です

427

TERMOUT オペランドが無効です

CD

0

Normal return

521

ファイル・プール定義の取得中にエラーが発生した

522

デフォルトの RFS ディレクトリーの作成中にエラーが発生した

523

現行 RFS ディレクトリー情報の格納中にエラーが発生した

524

RFS ディレクトリーが存在しないか、アクセス権限が許可されていない

525

ディレクトリー情報の取得中にエラーが発生した

526

ファイル・プール/ディレクトリーが無効です

527

過去のルート・ディレクトリーに戻ることはできない

528

結果値の設定中にエラーが発生した

PATH

- 0**
Normal return
- 625**
パス情報の取得中にエラーが発生した
- 626**
RFS ディレクトリー名が無効です
- 627**
PDS 名が無効です
- 628**
結果値の設定中にエラーが発生した
- 629**
データ・セット名が無効です
- 630**
パス情報の格納中にエラーが発生した
- 631**
現在定義されているパスはありません
- 632**
結果の PATH に RFS ディレクトリーや PDS 名が含まれていません

RLS

- 0**
Normal return
- 701**
無効なコマンド
- 702**
無効なオペランド
- 713**
ディレクトリーが見つかりません
- 715**
ディレクトリーは既に存在しています
- 716**
ディレクトリーが指定されていません
- 723**
リストが見つかりません
- 726**
リストが指定されていません
- 728**
リストが更新モードになっています
- 729**
リストが更新モードになっていません
- 730**
ユーザーがサインオンしていません
- 732**
キューが空です
- 733**
指定されたキューが見つかりませんでした
- 736**
語幹または変数が指定されていません

737

語幹名または変数名が長すぎます。

738

語幹または変数のカウントが無効です

743

ブロックが見つかりません

746

CICGETV エラー

747

GETMAIN error

748

FREEMAIN error

749

ENQ エラー

750

DEQ エラー

751

動的領域 GETMAIN のエラー

752

保管済み変数データにエラーがあります

753

保管済み変数は見つかりませんでした

754

ユーザーはリストの所有者ではありません

LISTCMD

0

Normal return

821

環境名が無効です

822

コマンド名が無効です

CLD

0

Normal return

923

現行 RLS ディレクトリー情報の格納中にエラーが発生した

924

RLS ディレクトリーが存在しないか、アクセス権限が許可されていない

925

ディレクトリー情報の取得中にエラーが発生した

926

無効なディレクトリー

927

過去のルート・ディレクトリーに戻ることはできない

928

結果値の設定中にエラーが発生した

DEFCMD

- 0**
Normal return
- 1001**
無効なコマンド
- 1021**
プログラムをロードできない
- 1023**
エントリーが見つからない
- 1048**
使用可能なクライアントがない
- 1099**
内部エラー

DEFSCMD

- 0**
Normal return
- 1101**
無効なコマンド
- 1121**
プログラムをロードできない
- 1123**
エントリーが見つからない
- 1148**
使用可能なクライアントがない
- 1199**
内部エラー

DEFTRNID

- 0**
Normal return
- 1202**
無効なオペランド
- 1222**
無効なオプション
- 1223**
trantable 情報の取得中にエラーが発生した
- 1225**
trantable 情報の取得中にエラーが発生した
- 1226**
EXEC 名の長さエラーです
- 1228**
trantable 値の設定中にエラーが発生した
- 1233**
トランザクションがテーブルに見つかりません

EXECDROP

0

Normal return

1401

無効なコマンド

1402

無効なオペランド

1423

EXECLOAD 情報の格納中にエラーが発生した

1425

EXECLOAD 情報の取得中にエラーが発生した

1448

使用可能なクライアントがない

EXECLOAD

0

Normal return

1501

無効なコマンド

1502

無効なオペランド

1523

EXECLOAD 情報の格納中にエラーが発生した

1525

EXECLOAD 情報の取得中にエラーが発生した

1530

CICPDS ルーチンにリンクできません

1531

CICPDS ルーチンからエラーが戻されました

1532

RFS READ からエラーが戻されました

1533

PDS ビルドからエラーが戻されました

1547

GETMAIN error

1548

使用可能なクライアントがありません

1599

内部エラー

EXECMAP

0

Normal return

1623

EXECLOAD ディレクトリーが見つかりません

ALLOC および FREE

SVC 99 によって与えられた戻りコード

詳しくは、[z/OS MVS Programming: Authorized Assembler Services Guide](#) を参照してください。

1702

無効なオペランド

EXPORT および IMPORT

0

Normal return

1701

無効なコマンド

1702

無効なオペランド

1723

RFS 書き込みエラー

1724

RFS 読み取りエラー

1725

動的割り振りが失敗しました

1726

動的解放が失敗しました

1727

一時データ・キューのオープンが失敗しました

1728

メンバーがエンキューされていません

1729

メンバーは使用中です

1730

レコードが切り捨てられました

1733

エクスポートする入力が見つかりません

1735

一時データ・エラー

1736

予期しない CICS エラー

1737

Invalid request

1738

データ・セット名が無効です

1739

無効な後処理

1741

DSORG がサポート対象外

1742

一時データ・プールのビルド・エラー

1743

REXX 一時データ・キューが使用不可です/見つかりません

1744

ユーザーがサインオンしていません/データ・セットへのアクセスを許可されていません

1745

データ・セットが空

1746

REXX キューが見つかりません

1748

Task Input/Output Table (TIOT) に DD 名のエントリーがありません

1749

複合データ・セットにはエクスポートできません

1750

DD 名として複数のデータ・セットが連結されています

1799

内部エラー

FILEPOOL

0

Normal return

1802

無効なオペランド

1821

無効なファイル・プール・サブコマンド

1822

ファイル・プール・サブコマンドが指定されていません

1823

ファイル・プール情報の格納中にエラーが発生した

1824

ファイル・プール ID が指定されていません

1825

ファイル・プール情報の取得中にエラーが発生した

1826

ファイル・プール ID が無効です

1827

取得されたファイル・プール・データが無効です

1828

ファイル・プールが定義されていません

1829

RFS はライブラリーをファイル・プールに追加できませんでした

1830

RFS はユーザー・ディレクトリーを作成できませんでした

1831

ファイル・プールの DDNAME を指定する必要があります

1832

DDNAME が無効です

1833

ファイル・プール変数が破損しています

1834

プール ID は既に存在します

1835

DDNAME は既に使用されています

1836

ファイル・プールをフォーマットできませんでした

1837

ファイル・プールは最初にフォーマットする必要があります

1838

ファイル・プールの ADD レコードがいっぱいです

1839

ファイル ID が見つかりません

GETVERS

0

Normal return

1910

要求は失敗しました

COPYR2S

0

Normal return

2002

無効なオペランド

2021

無効な構造定義

2022

無効な変数構造定義

2023

フィールド名が見つかりません

2025

GETVAR 要求の処理に失敗しました

2026

無効な数値入力

2027

RFS 読み取りエラー

2028

無効なオフセット

2029

無効な長さ値

COPYS2R

0

Normal return

2102

無効なオペランド

2121

無効な構造定義

2122

無効な変数構造定義

2123

フィールド名が見つかりません

2125

GETVAR 要求の処理に失敗しました

2126

無効な数値入力

2127

RFS 読み取りエラー

2128

無効なオフセット

2129

無効な長さ値

LISTPOOL

0

Normal return

2225

ファイル・プール情報の取得中にエラーが発生した

2226

語幹変数名が無効です

LISTTRNID

0

Normal return

2325

trantable 情報の取得中にエラーが発生した

C2S

0

Normal return

2440

変数名が指定されていません

2441

変数の取得中にエラーが発生しました

2442

変数の格納中にエラーが発生しました

2448

使用可能なクライアントがありません

PSEUDO

0

Normal return

2502

無効なオペランド

2521

オペランドが指定されていません

AUTHUSER

0

Normal return

2602

オペランドが無効であるか、または欠落しています

2621

ユーザー ID の無効な長さを指定します

2642

ユーザー ID の格納中にエラーが発生した

SETSYS

0

Normal return

2721

SETSYS サブコマンドが無効です

2722

変数の格納中にエラーが発生しました

2723

言語が無効です

2726

RETRIEVE PFkey オペランドが無効です

2727

TERMOUT オペランドが無効です

2732

PSEUDO オペランドが無効です

S2C

0

Normal return

2840

変数名が指定されていません

2841

変数の取得中にエラーが発生しました

2842

変数の格納中にエラーが発生しました

2848

使用可能なクライアントがありません

TERMID

0

Normal return

2921

端末 ID の取得中にエラーが発生した

2928

TERMID 値の設定中にエラーが発生した

WAITREAD

0

Normal return

3021

端末が接続されていません

3099

内部エラー

WAITREQ

0

Normal return

3121

WAITREQ が使用不可です

3122

EXEC がサーバーではありません

3123

要求変数の保管中にエラーが発生した

3199

内部エラー

特定のコマンドに関連付けられていない戻りコード

-3

EXEC が見つからないか、認識されていないコマンドです。

-4

ユーザーが許可ユーザーでないか、または EXEC が許可コマンドの使用を許可されていません。

-5

この CICS トランザクション定義に対して、TWA サイズが指定されていないか、指定された TWA サイズの長さが 4 文字未満でした。

-6

REXX/CICS が CICS LINK によって開始され、通信域に 16 文字未満が指定されました。あるいは、REXX/CICS が CICS XCTL によって開始され、通信域に MVS SIB タイプ 1 制御ブロックが含まれていないか、通信域の長さが 16 文字未満でした。

-99

内部エラー

EXEC

n

呼び出された EXEC の出口によって設定された戻りコード。[172 ページの『EXIT』](#)を参照してください。

0

Normal return

-3

EXEC が見つかりません

-10

EXEC 名が指定されていません

-11

EXEC 名が無効です

-12

GETMAIN error

-99

内部エラー

CEDA および CEMT

n

エラーが検出された場合に CICS によって戻される戻りコード。REXX からの CEMT の呼び出しでは、DFHEMTA (CEMT プログラマブル・インターフェース) が使用されるので、戻されるコードは、CEMT コマンドの戻りコードです。CEMT の戻りコードを次に示します。

- 1** NOT FOUND
- 2** CLASS NOT FOUND
- 3** ERROR
- 4** BEGINS WITH DFH
- 5** CHANGE INVALID
- 6** CANNOT NEWCOPY
- 7** NOT AUTHORIZED
- 8** OPTION CONFLICT
- 9** PRIORITY > 255
- 10** NOT FOR CONSOLE
- 11** PROGRAM NOT FOUND
- 12** INVALID AUTHID
- 13** INVALID ATI / TTI
- 14** IS NOT INTRA
- 15** OPEN/SWITCH FAIL
- 16** SDUMP BUSY
- 17** NOT SUCCESSFUL
- 18** 0>=TRIGGER>32767
- 19** NOT FOR INDIRECT
- 20** 0>MAXIMUM>999
- 21** IS NOT EXTRA
- 22** OPEN/CLOSE FAILED

- 23** SDUMP SUPPRESSED
- 24** BEGINS WITH C
- 25** NOT ACTIVE
- 26** NOT CLOSED
- 27** NOT FOR SYS LOG
- 28** ALREADY EXISTS
- 29** CATALOG I/O ERROR
- 30** 1>MAXTASKS>2000
- 31** NOT FOR REMOTE
- 32** NOSTG DSALIMIT
- 33** 0>AGING>65535
- 34** AKP NOT IN SYSTEM
- 35** MAXT. < AMAXT.
- 36** NOT IN SYSTEM
- 37** INVALID COMAUTHID
- 38** 50>AKP>65535
- 39** CATALOG FULL
- 40** 2M>DSALIMIT>16M
- 41** 1>MAXACTIVE>999
- 42** INVALID DUMPCODE
- 43** INVALID DB2ID
- 44** 500>RUN. >2700000
- 45** 100>TIME>3600000
- 46** BAD TRANSAC CLASS
- 47** TIME < SCANDELAY

48 CEILING REACHED
49 1>MROBATCH>255
50 48M>EDSALIM>2047M
51 CLOSE FAILED
52 NOT FOR BDAM
53 CLOCK INOPERATIVE
54 NOT VALID VTAM
55 NOT FOR PATH
56 OPEN FILE
57 BEING CLOSED
58 BEING IMMCLOSED
59 0>SCANDELAY>5000
60 NOT FOR SNASVCMG
61 NOT FOR THIS TASK
62 NOT FOR YOUR TERM
63 INVALID MSGQUEUE
64 NOT FOR YOUR LINE
65 QUEUE IS DISABLED
66 NOSTG EDSALIMIT
67 PARTIAL DUMP
68 DDNAME NOT FOUND
69 BEING ACQUIRED
70 BEING FORCECLOSED
71 NOT FOR HOLD PROG
72 LOAD FAILED

- 73** SDUMP FAILED
- 74** EMPTY OR NOT CLSD
- 75** NOT DISABLED
- 76** CLOSE REQUESTED
- 77** BEING OPENED
- 78** BEING UNENABLED
- 79** BEING DISABLED
- 80** BEING QUIESCED
- 81** SEE MSG DFHIR3793
- 82** START/SWITCH FAIL
- 83** INVALID PLAN
- 84** INVALID INTERVAL
- 85** OUT &-REL INVALID
- 86** SEE MSG DFHIR3768
- 87** SEE MSG DFHIR3786
- 88** NOT FOR MAPSET
- 89** SEE MSG DFHIR3771
- 90** NOT FOR PARTITION
- 91** SEE MSG DFHIR3773
- 92** INV DSRTPROGRAM
- 93** SEE MSG DFHIR3775
- 94** SEE MSG DFHIR3776
- 95** SEE MSG DFHIR3777
- 96** SEE MSG DFHIR3778
- 97** SEE MSG DFHIR3779

98
SEE MSG DFHIR3780

99
SEE MSG DFHIR3781

100
SEE MSG DFHIR3791

101
ONLY FOR VTAM

102
GOING OUT

103
SPECIFY NUMBER

104
NUMBER ERROR

105
NEGPOLL INVALID

106
>20000

107
INVALID ENDOFDAY

108
MAX | SHUTDOWN

109
LINE DCB NOT OPEN

110
INV PLANEXITNAME

111
SET FAILED

112
REMOVE FAILED

113
INVALID SIGNID

114
FILECOUNT > 0

115
INV STATSQUEUE

116
BACKOUT FAILED

117
INV COMTHREADLIM

118
>MAXIMUM

119
INV PURGECYCLE

120
IS INDOUBT

121
4>TCBLIMIT>2000

122
CONNECTION -ACQD

- 123**
DATASET QUIESCING
- 124**
IN PROGRESS
- 125**
DATASET UNAVAIL
- 126**
DATASET QUIESCED
- 127**
BACKUP OCCURRING
- 128**
RESOURCE MISSING
- 129**
STATS MISSING
- 130**
IS SIT PARAMETER
- 131**
CANNOT LOAD PLT
- 132**
INV THREADLIMIT
- 133**
NO DATASET
- 134**
RECOVERY REQUIRED
- 135**
PURGE FAILED
- 136**
FILE IN USE
- 137**
ONLY FOR BDAM
- 138**
STOPPED
- 139**
SEE MSG DFHIR3798
- 140**
PROGRAM IS URM
- 141**
IN USE
- 142**
IN USE BY ICE
- 143**
IN USE BY PCT
- 144**
NOT RELEASED
- 145**
DELETE FAILED
- 146**
INVALID RECOVSTAT
- 147**
NOT OUTSERVICE

148
INV PROTECTNUM

149
INVALID DB2ENTRY

150
IS ENABLED EXIT

151
INV DTRPROGRAM

152
IN USE BY AID

153
NOT FOR SESSION

154
NOT FOR PIPELINE

155
NOT DISCARDABLE

156
NOT LOCAL SYSTEM

157
NOT FOR SYSTEM

158
NOT FOR MODEL

159
NO ACTION

160
CANNOT LOAD XLT

161
ISC NOT DEFINED

162
ALREADY ACTIVE

163
INVALID TRANSID

164
DUPLICATE TRANSID

165
NO BWO SUPPORT

166
DSNB IS INVALID

167
DSNB BDAM OR PATH

168
UPDATE COUNT > 0

169
DSN → SMS MANAGED

170
BWO ERROR

171
DFHTMP ERROR

172
PRESET SIGNON ERR

173
NOT FOR CPSVCMG

174
PRM UNAVAILABLE

175
NOT WHEN XLN DONE

176
INV PROGAUTOINST

177
INV PROGAUTOCTLG

178
INV PROGAUTOEXIT

179
FREE NOT COMPLETE

180
BEING RELEASED

181
REUSE DEFINED

182
UNBLOCKED DEFINED

183
0< MAX. <999999999

184
FORMAT NOT VARBLE

185
NO LSRPOOL

186
CONNECTING

187
INVALID FREQUENCY

188
0>PURGET.>1000000

189
INVALID PSDINT

190
NOT WITH XRF

191
SETLOGON FAILURE

192
BACK LEVEL VTAM

193
ACB CLOSED

194
RECOVERY ERROR

195
DEFERRED

196
NOT FOR LOCAL SYS

197
SEE MSG DFHIR3799

198
ONLY FOR APPC

199
ESM INACTIVE

200
REGISTER ERROR

201
DEREGISTER ERROR

202
AIDS CANCELED

203
WAITING FOR DB2

204
DB2 INACTIVE

205
INVALID INITPARM

206
NOT REQUIRED

207
STILL CLOSING

208
NO AIDS CANCELED

209
DFSMS CATLG ERROR

210
DSNB REMOVED

211
NO RLS SUPPORT

212
SYSTEM LOCKED

213
SDTRAN NOT FOUND

214
SDTRAN DISABLED

215
SDTRAN SHUT DIS

216
DSN → DFSMS VSAM

217
DATASET MIGRATED

218
INVALID IDLE

219
NOT LU61 OR LU62

220
NETID 0 USE PRFRM

221
SEE MSG DFHZC0178

222
NOT GR REGISTERED

223
ENTER NETID

224
NO AFFINITY FOUND

225
SESSIONS IN USE

226
SEE MSG DFHZC0176

227
DELETE INFLIGHT

228
USED BY INDIRECTS

229
IRC IS OPEN

230
IS ERROR CONSOLE

231
SDTRAN IS REMOTE

232
XRF NOT ACTIVE

233
SYSID IN ERROR

234
ERR: SHUNTED UOWS

235
SEE MSG DFHZC0173

236
RLS AND CMT

237
DISCONNECTING

238
KEYLENGTH ERROR

239
RECORDSIZE ERROR

240
MISSING POOL NAME

241
INVALID NAME

242
POOL NOT FOUND

243
CONTEN AND RECOV

244
INVALID ACTION

245
LASTUSED<INTERVAL

246
WAITING

247
NO AUDITLOG

248
PARAM MISMATCH

249
NO CFDT SERVER

250
TCPIP CLOSED

251
PORT IN USE

252
PORT NOT AUTH

253
INVALID STATUS

254
PROFILE NOT FOUND

255
ADDRESS UNKNOWN

256
INVALID Q-TYPE

257
1>MAXOPENTCBS>2000

258
NO JVMCLASS SET

259
NOT A JVM PROGRAM

260
INVALID JVMCLASS

261
INVALID DSNAME

262
1>MAXSOCKETS>65535

263
EXCEEDS HARD LIMIT

264
AT MAXSOCKETS

265
TCIPSERVICE NOT OPENED

266
SESSBEANTIME > 143999

267
RESOURCE NOT INSERVICE

268
MISSING CORBASERVER NAME

269
DJAR IS PENDING RESOLUTION

270
INVALID DB2GROUPID

271
DB2ID & GROUPID ENTERED

272
1>MAXJVMTCBS>999

273
1>MAXXPTCBS>999

274
NOT IIOPLISTENER

275
DSNAPRH NOT FOUND

276
MAXOPENTCBS < DB2CONN TCBLIMIT

277
TCBLIMIT > MAXOPENTCBS

278
DB2 GROUPID NOT FOUND

279
DB2 ID NOT FOUND

280
DJAR CLASH (CORBASERVER SCAN)

281
JVMPROFILE INVALID..SET PROGRAM

282
BEING STARTED

283
BEING RELOADED

284
BEING ENABLED

285
BEING DISCARDED

286
NOT STARTED

287
NOT STOPPED

288
DB2 RESTART-LIGHT

289
CACHESIZE INVALID

290
TCPIP INACTIVE

291
ATTEMPT FORCE PURGE

292
1>MAXSSLTCBS>1024

293
PIPELINE NOT ENABLED

294
MISSING JVMPROFILE

295
NOT VALID FOR DFHRPL

296
RANKING 10 RESERVED FOR DFHRPL

297
RANKING OUT OF RANGE

298
NO ACQ FOR SEND=0

299
MAXJVMTCBS EXCEEDED

300
PSTYPE=NOPS AND PSDI > 0

301
INV FILELIMIT

302
INV PROGRAMLIMIT

303
INV TSQUEUELIMIT

304
MQNAME IN USE

305
INVALID MQNAME

306
MQNAME NOT FOUND

307
WAITING FOR QMGR

308
1>THREADLIMIT>256

309
NO MORE THREADS

310
THREADS LIMITED

311
DRAINING

312
SEE MSG DFHPI2024

313
EXCESSIVE NUMBER OF ELEMENTS

314
1M>TSMMAINLIM>32G

315
ATTEMPT PURGE 1ST

316
USING JVMSERVER

317
INV REUSELIMIT

318
> 25.00% OF MEMLIMIT

319
DEFINED BY BUNDLE

320
SEE MSG DFHSO0123

321
NOT FOR JVM PROG

322
USED BY MGMTPART

注：CICS リリースによって、戻されない戻りコードもあります。

0
Normal return

-101
無効なコマンド

EXECIO

n
エラーが検出された場合に CICS によって戻される戻りコード。

0
Normal return

-202
無効なオペランド

-221
指定したオペランドが多すぎます

-222
recno オペランドが範囲外です

-224
lines オペランドが無効です

CONVTMAP

n
MVS データ・セットの処理試行からの戻りコード

0
Normal return

-302
無効なオペランド

-321
無効な入力レコード

-322
出力ファイルの書き込み中の RFS エラー

SCRNINFO

n
エラーが検出された場合に CICS によって戻される戻りコード。

0
Normal return

-499
内部エラー

CICS

-521
コマンドがサポートされていない

-522
無効なコマンドまたはキーワード

-523
オプションを指定しなければなりません

- 524**
サポートされていないオプションが指定されました
- 525**
矛盾するオプションが指定されました
- 526**
暗黙のオプションが指定されていません
- 527**
オプションの指定が冗長です
- 528**
オプションの値が指定されていません
- 529**
値を持つてはならないオプションに値が指定されています
- 530**
オプションに指定された値は数値ではありません
- 531**
値が無効です
- 532**
指定された値が長すぎます
- 533**
指定された値が短すぎます
- 534**
値が指定されていません
- 535**
変数テーブルがオーバーフローしています
- 536**
変数の数が変数テーブルの制限を超えています
- 537**
引数は変数でなければなりません
- 538**
変数が存在しません
- 539**
変数名が無効です
- 540**
マスター・システム・トレース・フラグがトレースに対してオンになっている必要があります
- 541**
Parsing error
- 542**
総称名が無効です
- 543**
右括弧が欠落しています
- 544**
値/キーワードがあいまいです
- 545**
RIDFLD はフルワードの変数でなければなりません
- 546**
RIDFLd は変数でなければなりません
- 547**
GETVAR の戻りコードが無効です
- 548**
内部 GETVAR エラー

-549

PUTVAR の戻りコードが誤っています

-550

PUTVAR が失敗しました

-551

ストレージを取得できません

-552

EXEC CICS コマンド・テーブルが見つかりません

- EBCDIC shift-out (X'0E'): <
- EBCDIC shift-in (X'0F'): >

注: EBCDIC では、シフトアウト (SO) 文字とシフトイン (SI) 文字によって、DBCS 文字と SBCS 文字を区別します。

DBCS データ操作および記号の使用可能化

OPTIONS 命令によって、REXX で DBCS データをどのように処理するのかを制御します。

- DBCS 操作を使用可能にするには、EXMODE オプションを使用します。
- また、DBCS 記号を使用可能にするには、OPTIONS 命令で ETMODE オプションを使用してください。これは、プログラム内で最初の命令にする必要があります。177 ページの『OPTIONS』を参照してください。

OPTIONS ETMODE が有効な場合、言語処理プログラムは、SO および SI がコメントの中で必ずペアになっていることを確認するための妥当性検査を行います。そうになっていなければ、コメントの内容は検査されません。コメント区切り文字 (/ * および *) は、SBCS 文字でなければなりません。

シンボルとストリング

DBCS には、DBCS 専用のシンボルとストリング、さらに、混合のシンボルとストリングがあります。

DBCS 専用のシンボルと SBCS/DBCS 混合のシンボル

DBCS 専用のシンボルは、以下の表に示す非ブランクの DBCS コードのみで構成されます。

表 7. DBCS の範囲	
バイト	EBCDIC
1 バイト目	X'41' から X'FE' まで
2 バイト目	X'41' から X'FE' まで
DBCS のブランク	X'4040'

混合の DBCS シンボルは、SBCS シンボル、DBCS 専用のシンボル、およびその他の DBCS シンボルの連結により形成されます。EBCDIC では、SO と SI が DBCS シンボルを囲み、SBCS シンボルと区別します。

DBCS シンボルのデフォルト値は、シンボルそのもので、SBCS 文字は大文字に変換されます。

定数シンボルは、SBCS の数字 (0 から 9) または SBCS のピリオドで開始する必要があります。複合シンボルの区切り文字 (ピリオド) は、SBCS 文字でなければなりません。

DBCS 専用ストリングと SBCS/DBCS 混合ストリング

DBCS 専用ストリングは、DBCS 文字のみから構成されます。

SBCS/DBCS 混合ストリングは、SBCS 文字および DBCS 文字の組み合わせで構成されます。EBCDIC では、SO と SI が DBCS データを囲み、SBCS データと区別します。SO および SI は混合ストリングでのみ必要のため、DBCS 専用ストリングとは関連付けられません。

EBCDIC では、以下のようになります。

- DBCS 専用ストリング: .A.B.C
- 混合ストリング: ab<.A.B>
- 混合ストリング: <.A.B>
- 混合ストリング: ab<.C.D>ef

DBCS 記号の妥当性検査

DBCS 記号を使用する際に適用される規則と条件をリストしています。

- シンボルの DBCS 部分は偶数のバイト長でなければならない。

- DBCS 英数字および特殊シンボルは、対応する SBCS 文字とは別なものと見なされる。SBCS 文字だけが、数値、命令キーワード、または演算子において認められます。
- DBCS 文字を、REXX の特殊文字として使用することはできません。
- SO および SI を隣接させることはできません。
- SO または SI をネストにすることはできません。
- SO および SI はペアにしなければなりません。
- DBCS 文字で構成される記号のどの部分にも、DBCS ブランクを含めることはできません。
- DBCS 文字で構成される記号の各部分は、SO および SI で囲まなければなりません。

下記に、誤用の例をいくつか示します。

- <.A.BC> バイト長が奇数であるため誤り。
- <.A.B><.C> SO/SI が連続しているのは誤り。
- <> SO/SI が連続しているのは誤り (ヌル DBCS 記号)。
- <.A<.B>.C> SO/SI が誤ってネストされている。
- <.A.B.C SO/SI がペアになっていないため誤り。
- <.A. .B> ブランクが含まれているため誤り。
- '. A<.B><.C> 記号の誤り。

混合ストリングの妥当性検査

混合ストリングの妥当性検査は、命令、演算子、または関数により異なります。

混合ストリングを、混合ストリングが許されない命令、演算子、または関数で使用すると、構文エラーが生じます。

混合ストリングの妥当性検査では、次の規則が適用されます。

- DBCS ストリングは、SO と SI をもたない限り偶数のバイト長でなければならない。

EBCDIC だけの場合

- SO および SI は、ストリング内でペアになっていなければなりません。
- SO または SI をネストにすることはできません。

下記に、誤用の例をいくつか示します。

- 'ab<cd' 間違い - ペアになっていません。
- '<.A<.B>.C>' 間違い - ネストされています。
- '<.A.BC>' 間違い - バイト長が奇数です。

コメント区切り文字の終わりは、DBCS 文字シーケンス内では見つかりません。例えば、プログラムに /*< */ が含まれている場合、*/ はコメントの終わりとして認識されません。スキャンは、< (SO) と対になる > (SI) を対象としており、*/ を対象としていないためです。

変数が OPTIONS EXMODE 下の REXX プログラム内で作成、修正、または参照されると、正しい混合ストリングが含まれるかどうかの妥当性検査されます。参照した変数に有効でない混合ストリングが含まれていると、命令、関数または演算子によっては構文エラーの原因となる場合があります。

ARG、PARSE、PULL、PUSH、QUEUE、SAY、TRACE、および UPPER 命令はすべて 有効な OPTIONS EXMODE を持つ有効な混合ストリングでなければなりません。

命令と DBCS

命令が DBCS をどのように処理するかを説明し、その例を記載しています。

PARSE

DBCS を使用した PARSE 命令の例。

EBCDIC では、以下のようになります。

```
x1 = '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1
      w1 -> '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 1 w1
      w1 -> '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1 .
      w1 -> '<.A.B>'
```

ワード解析の場合、先行および末尾の SO と SI は不要であるため、これらの制御文字は除去されます。ただし、それでも、有効な混合 DBCS スtring を戻すために、対が 1 つ必要です。

```
PARSE VAR x1 . w2
      w2 -> '<. ><.E><.F><>'
```

この例では、最初の空白でワードを区切り、その String に SO を追加して、DBCS 空白および正しい混合 String を確実なものにしています。

```
PARSE VAR x1 w1 w2
      w1 -> '<.A.B>'
      w2 -> '<. ><.E><.F><>'

PARSE VAR x1 w1 w2 .
      w1 -> '<.A.B>'
      w2 -> '<.E><.F>'
```

このワードの区切り方により、不要な SO と SI をドロップすることができます。

```
x2 = 'abc<>def <.A.B><><.C.D>'

PARSE VAR x2 w1 ' ' w2
      w1 -> 'abc<>def <.A.B><><.C.D>'
      w2 -> ' '

PARSE VAR x2 w1 '<>' w2
      w1 -> 'abc<>def <.A.B><><.C.D>'
      w2 -> ' '

PARSE VAR x2 w1 '<><>' w2
      w1 -> 'abc<>def <.A.B><><.C.D>'
      w2 -> ' '
```

最後の 3 つの例の "、<>、および <><> は、それぞれヌル・String (長さ 0 の String) であることに注意してください。構文解析時に、ヌル・String は String の終わりを一致させます。このため、w1 には String 全体の値が割り当てられ、w2 にはヌル・String が割り当てられます。

PUSH および QUEUE

DBCS を使用した PUSH および QUEUE 命令。

PUSH 命令および QUEUE 命令は、プログラム・スタックに項目を追加するときに使用します。スタック入力は 255 バイトに制限されているため、*expression* は 256 バイト未満に切り捨てなければなりません。切り捨てによって DBCS String が分割されると、REXX は、OPTIONS EXMODE のもとでは SO-SI がペアになるように維持します。

SAY および TRACE

DBCS を使用した SAY および TRACE 命令。

SAY 命令および TRACE 命令は、出力ストリームにデータを書き込むために使用します。PUSH 命令および QUEUE 命令の場合と同様に、REXX は、出力ストリームの要件を満たすために分割されるデータがあっても、SO-SI ペアを必ず維持します。ユーザーの端末にデータを表示するためには、SAY 命令および TRACE 命令を使用します。PUSH 命令および QUEUE 命令の場合と同様に、REXX は、端末行サイズの要件を満た

すために分割されるデータがあっても、SO-SI ペアを必ず維持します。このサイズは、一般には 130 バイトですが、DIAG-24 値が戻す値がそれより小さいときには、130 バイト未満になります。

短い長さにデータが分割されても、OPTIONS EXMODE を使用する場合は DBCS データの整合性が維持されます。EBCDIC では、端末の行サイズが 4 より小さいと、ストリングは SBCS データとして扱われますが、これは 4 が混合ストリング・データの最小であるためです。

UPPER

DBCS を使用した UPPER 命令

OPTIONS EXMODE を使用する場合、UPPER 命令は、1 つ以上の変数の内容の SBCS 文字だけを大文字に変換します。DBCS 文字を変換することはありません。変数の内容が有効な混合ストリング・データでなければ、大文字変換は行われません。

DBCS 関数の処理

組み込み関数のいくつかは、DBCS を処理することができます。

ワードの区切りおよび長さの区切りを扱う関数は、OPTIONS EXMODE では以下の規則に従います。

- 文字のカウント。

ストリングの長さをカウントするときは、論理的な文字の長さ (つまり、1 つの SBCS 論理文字には 1 バイト、1 つの DBCS 論理文字には 2 バイト) が使用されます。EBCDIC では、SO と SI は透過と考えられ、すべてのストリング操作でカウントされません。

- ストリングからの文字の抽出。

論理文字単位でストリングから文字が抽出されます。EBCDIC では、先行する SO と末尾の SI は DBCS の 1 文字の一部とは見なされません。例えば、.A および .B が <.A.B> から抽出され、最終的に完全な DBCS 文字として保存される場合は、SO と SI が各 DBCS 文字に追加されます。複数の文字が連続してストリングから抽出される場合は、文字の間の SO と SI も抽出されます。例えば、.A><.B は <.A><.B> から抽出され、ストリングが、完成したストリングとして最終的に使用されるときに、SO が接頭部となり、SI が接尾部になって <.A><.B> を指定します。

EBCDIC の場合のいくつかの例を、以下に示します。

```
S1 = 'abc<>def'

SUBSTR(S1,3,1)    ->  'c'
SUBSTR(S1,4,1)    ->  'd'
SUBSTR(S1,3,2)    ->  'c<>d'

S2 = '<><.A.B><>'

SUBSTR(S2,1,1)    ->  '<.A>'
SUBSTR(S2,2,1)    ->  '<.B>'
SUBSTR(S2,1,2)    ->  '<.A.B>'
SUBSTR(S2,1,3,'x') ->  '<.A.B><>x'

S3 = 'abc<><.A.B>'

SUBSTR(S3,3,1)    ->  'c'
SUBSTR(S3,4,1)    ->  '<.A>'
SUBSTR(S3,3,2)    ->  'c<><.A>'
SUBSTR(S3,3,1)    ->  'ab<><.A.B>'
DELSTR(S3,4,1)    ->  'abc<><.B>'
DELSTR(S3,3,2)    ->  'ab<.B>'
```

- 文字の連結。

ストリング連結を行うことができるのは、有効な混合ストリングだけです。EBCDIC では、ストリングの連結の結果、SI と SO (または SO と SI) が隣接すると、不要なものが除去されます。DELSTR 関数などの暗黙的な連結であっても、不要な SO と SI が除去されます。

- 文字の比較。

文字単位でストリングを比較する場合は、有効な混合ストリングが使用されます。DBCS 文字は、比較の際 SBCS 文字より常に大きいと見なされます。厳密比較を除くすべての比較で、SBCS ブランク、DBCS

ブランク、および EBCDIC 内の 先行と末尾の連続する SO と SI (または SI と SO) は除去されます。長さが同じでない場合は、SBCS ブランクが追加される場合があります。

EBCDIC では、非ブランク文字間の連続する SO と SI (または SI と SO) も 比較の場合除去されます。

注：厳密比較の演算子は、有効でない混合ストリングを指定した場合でも 構文エラーとはなりません。

EBCDIC では、以下のようになります。

```
'<.A>' = '<.A. >'      -> 1      /* true */
'<><><.A>' = '<.A><><>'    -> 1      /* true */
'<> <.A>' = '<.A>'      -> 1      /* true */
'<.A><><.B>' = '<.A.B>'    -> 1      /* true */
'abc' < 'ab<. >'      -> 0      /* false */
```

- ストリングからのワードの抽出。

ワード は、ストリング内の文字が SBCS または DBCS のブランクで区切られていることを意味します。

EBCDIC では、ワードがストリング内で分割されるが、ワードの操作で、ワード内の連続する SO と SI (または SI と SO) が除去されないか、または分割されなければ、先行と後書きの連続する SO と SI (または SI と SO) も除去されます。ワードの先行と後書きの連続する SO と SI (または SI と SO) は、一度に抽出されるワードに属する場合は除去されません。

EBCDIC では、以下のようになります。

```
W1 = '<><. .A. . .B><.C. .D><>'

SUBWORD(W1,1,1)      -> '<.A>'
SUBWORD(W1,1,2)      -> '<.A. . .B><.C>'
SUBWORD(W1,3,1)      -> '<.D>'
SUBWORD(W1,3)        -> '<.D>'

W2 = '<.A. .B><.C><> <.D>'

SUBWORD(W2,2,1)      -> '<.B><.C>'
SUBWORD(W2,2,2)      -> '<.B><.C><> <.D>'
```

組み込み関数の例

DBCS をサポートし、定義された規則に従う組み込み関数の例を示します。

関数および構文図の詳細については、[193 ページの『第 19 章 関数』](#)を参照してください。

ABBREV

文字の比較規則とストリングからの文字の抽出規則が適用されます。

EBCDIC では、以下のようになります。

```
ABBREV('<.A.B.C>','<.A.B>')      -> 1
ABBREV('<.A.B.C>','<.A.C>')      -> 0
ABBREV('<.A><.B.C>','<.A.B>')      -> 1
ABBREV('aa<>bbccdd','aabbcc')     -> 1
```

COMPARE

埋め込みのための文字の連結、ストリングからの文字の抽出、および文字の比較の規則が適用されます。

EBCDIC では、以下のようになります。

```
COMPARE('<.A.B.C>','<.A.B><.C>')  -> 0
COMPARE('<.A.B.C>','<.A.B.D>')    -> 3
COMPARE('ab<>cde','abcdx')        -> 5
COMPARE('<.A><>','<.A>','<. >')    -> 0
```

COPIES

文字の連結規則が適用されます。

EBCDIC では、以下のようになります。


```
COPIES('<.A.B>',2)      -> '<.A.B.A.B>'
COPIES('<.A><.B>',2)     -> '<.A><.B.A><.B>'
COPIES('<.A.B><>',2)     -> '<.A.B><.A.B><>'
```

DATATYPE

DATATYPE 関数の例。

```
DATATYPE('<.A.B>')      -> 'CHAR'
DATATYPE('<.A.B>', 'D') -> 1
DATATYPE('<.A.B>', 'C') -> 1
DATATYPE('a<.A.B>b', 'D') -> 0
DATATYPE('a<.A.B>b', 'C') -> 1
DATATYPE('abcde', 'C')   -> 0
DATATYPE('<.A.B>', 'C')   -> 0
DATATYPE('<.A.B>', 'S')   -> 1      /* if ETMODE is on */
```

string が有効な混合ストリングではなく、C または D が *type* として指定されていた場合には、0 が戻されます。

FIND

ストリングからのワードの抽出、 および文字の比較の規則を適用します。

```
FIND('<.A. .B.C> abc', '<.B.C> abc') -> 2
FIND('<.A. .B><.C> abc', '<.B.C> abc') -> 2
FIND('<.A. . .B> abc', '<.A> <.B>')   -> 1
```

INDEX、POS、および LASTPOS

ストリングからの文字の抽出規則と文字の比較規則が適用されます。

```
INDEX('<.A><.B><><.C.D.E>', '<.D.E>') -> 4
POS('<.A>', '<.A><.B><><.A.D.E>')   -> 1
LASTPOS('<.A>', '<.A><.B><><.A.D.E>') -> 3
```

INSERT および OVERLAY

ストリングからの文字の抽出規則と文字の比較規則が適用されます。

EBCDIC では、以下ようになります。

```
INSERT('a', 'b<><.A.B>',1)      -> 'ba<><.A.B>'
INSERT('<.A.B>', '<.C.D><>',2)     -> '<.C.D.A.B><>'
INSERT('<.A.B>', '<.C.D><><.E>',2) -> '<.C.D.A.B><><.E>'
INSERT('<.A.B>', '<.C.D><>',3, '<.E>') -> '<.C.D><.E.A.B>'

OVERLAY('<.A.B>', '<.C.D><>',2)      -> '<.C.A.B>'
OVERLAY('<.A.B>', '<.C.D><><.E>',2) -> '<.C.A.B>'
OVERLAY('<.A.B>', '<.C.D><><.E>',3) -> '<.C.D><><.A.B>'
OVERLAY('<.A.B>', '<.C.D><>',4, '<.E>') -> '<.C.D><.E.A.B>'
OVERLAY('<.A>', '<.C.D><.E>',2)      -> '<.C.A><.E>'
```

JUSTIFY

埋め込みの文字の連結、 およびストリングからの文字の抽出の規則を適用します。

```
JUSTIFY('<><.A. . .B><.C. .D>',10, 'p')
-> '<.A>ppp<.B><.C>ppp<.D>'
JUSTIFY('<><.A. . .B><.C. .D>',11, 'p')
-> '<.A>pppp<.B><.C>ppp<.D>'
JUSTIFY('<><.A. . .B><.C. .D>',10, '<.P>')
-> '<.A.P.P.P.B><.C.P.P.P.D>'
JUSTIFY('<><.X. .A. . .B><.C. .D>',11, '<.P>')
-> '<.X.P.P.A.P.P.B><.C.P.P.D>'
```

LEFT、RIGHT、および CENTER

埋め込みの文字の連結、 およびストリングからの文字の抽出の規則を適用します。

EBCDIC では、以下ようになります。

```

LEFT('<.A.B.C.D.E>',4)      -> '<.A.B.C.D>'
LEFT('a<>',2)                -> 'a<>'
LEFT('<.A>',2,'*')           -> '<.A>*'
RIGHT('<.A.B.C.D.E>',4)      -> '<.B.C.D.E>'
RIGHT('a<>',2)               -> 'a'
CENTER('<.A.B>',10,'<.E>')    -> '<.E.E.E.E.A.B.E.E.E.E>'
CENTER('<.A.B>',11,'<.E>')    -> '<.E.E.E.E.A.B.E.E.E.E.E>'
CENTER('<.A.B>',10,'e')       -> 'eeee<.A.B>eeee'

```

LENGTH

文字のカウント規則が適用されます。

EBCDIC では、以下のようになります。

```

LENGTH('<.A.B><.C.D><>')      -> 4

```

REVERSE

ストリングからの文字の抽出規則と文字の連結規則が適用されます。

EBCDIC では、以下のようになります。

```

REVERSE('<.A.B><.C.D><>')      -> '<<<.D.C><.B.A>'

```

SPACE

ストリングからのワードの抽出規則と文字の連結規則が適用されます。

EBCDIC では、以下のようになります。

```

SPACE('a<.A.B. .C.D>',1)      -> 'a<.A.B> <.C.D>'
SPACE('a<.A><><. .C.D>',1,'x') -> 'a<.A>x<.C.D>'
SPACE('a<.A><. .C.D>',1,'<.E>') -> 'a<.A.E.C.D>'

```

STRIP

ストリングからの文字の抽出規則と文字の連結規則が適用されます。

EBCDIC では、以下のようになります。

```

STRIP('<<<.A><.B><.A><>', '<.A>') -> '<.B>'

```

SUBSTR および DELSTR

ストリングからの文字の抽出規則と文字の連結規則が適用されます。

EBCDIC では、以下のようになります。

```

SUBSTR('<<<.A><><.B><.C.D>',1,2) -> '<.A><><.B>'
DELSTR('<<<.A><><.B><.C.D>',1,2) -> '<<<.C.D>'
SUBSTR('<.A><><.B><.C.D>',2,2)   -> '<.B><.C>'
DELSTR('<.A><><.B><.C.D>',2,2)   -> '<.A><><.D>'
SUBSTR('<.A.B><>',1,2)           -> '<.A.B>'
SUBSTR('<.A.B><>',1)             -> '<.A.B><>'

```

SUBWORD および DELWORD

ストリングからのワードの抽出規則と文字の連結規則が適用されます。

EBCDIC では、以下のようになります。

```

SUBWORD('<<<.A. .B><.C. .D>',1,2) -> '<.A. .B><.C>'
DELWORD('<<<.A. .B><.C. .D>',1,2) -> '<<<.D>'
SUBWORD('<<<.A. .B><.C. .D>',1,2) -> '<.A. .B><.C>'
DELWORD('<<<.A. .B><.C. .D>',1,2) -> '<<<.D>'
SUBWORD('<.A. .B><.C><<.D>',1,2) -> '<.A. .B><.C>'
DELWORD('<.A. .B><.C><<.D>',1,2) -> '<.D>'

```

SYMBOL

SYMBOL 関数の例。

EBCDIC では、以下のようになります。

```
Drop A.3 ; <.A.B>=3      /* if ETMODE is on */

SYMBOL('<.A.B>')    ->    'VAR'
SYMBOL(<.A.B>)      ->    'LIT' /* has tested "3" */
SYMBOL('a.<.A.B>') ->    'LIT' /* has tested A.3 */
```

TRANSLATE

ストリングからの文字の抽出、文字の比較、および文字の連結の規則が適用されます。

EBCDIC では、以下のようになります。

```
TRANSLATE('abcd','<.A.B.C>','abc')    -> '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','abc')    -> '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','ab<>c')  -> '<.A.B.C>d'
TRANSLATE('a<>bcd','<><.A.B.C>','ab<>c') -> '<.A.B.C>d'
TRANSLATE('a<>xcd','<><.A.B.C>','ab<>c') -> '<.A>x<.C>d'
```

VALUE

VALUE 関数の例。

EBCDIC では、以下のようになります。

```
Drop A3 ; <.A.B>=3 ; fred='<.A.B>'

VALUE('fred')      ->    '<.A.B>' /* looks up FRED */
VALUE(fred)         ->    '3'      /* looks up <.A.B> */
VALUE('a'<.A.B>)    ->    'A3'     /* if ETMODE is on */
```

VERIFY

ストリングからの文字の抽出規則と文字の比較規則が適用されます。

EBCDIC では、以下のようになります。

```
VERIFY('<><><.A.B><><.X>','<.B.A.C.D.E>') -> 3
```

WORD、WORDINDEX、および WORDLENGTH

ストリングからのワード抽出規則と、WORDINDEX および WORDLENGTH の場合の文字のカウンtr規則の適用

EBCDIC では、以下のようになります。

```
W = '<><.A. .B><.C. .D>'

WORD(W,1)          ->    '<.A>'
WORDINDEX(W,1)      ->    2
WORDLENGTH(W,1)     ->    1

Y = '<><.A. .B><.C. .D>'

WORD(Y,1)           ->    '<.A>'
WORDINDEX(Y,1)       ->    1
WORDLENGTH(Y,1)      ->    1

Z = '<.A .B><.C> <.D>'

WORD(Z,2)           ->    '<.B><.C>'
WORDINDEX(Z,2)       ->    3
WORDLENGTH(Z,2)      ->    2
```

WORDS

ストリングからのワード抽出規則の適用

EBCDIC では、以下のようになります。

```
W = '<>.A. .B>.C. .D>'
WORDS(W)      ->    3
```

WORDPOS

ストリングからのワードの抽出規則と文字の比較規則が適用されます。

EBCDIC では、以下のようになります。

```
WORDPOS('<.B.C> abc','<.A. .B.C> abc')      ->    2
WORDPOS('<.A.B>','<.A.B. .A.B>.B.C. .A.B>',3) ->    4
```

DBCS 処理関数

DBCS 混合ストリングをサポートする関数について説明します。これらの関数は、OPTIONS モードには関係なく混合ストリングを取り扱います。

注: DBCS 関数を使用する場合には、*length* が必ずバイト単位で計測されます (文字単位で計測される *LENGTH(string)* とは異なります)。

カウント・オプション

EBCDIC では、カウント・オプションが関数内に指定される場合に、長さを判別するときに、SO と SI が存在するものと考えるかどうかを制御できます。Y を指定すると、混合ストリング内の SO と SI がカウントされます。N (デフォルト) を指定すると、SO と SI はカウントされません。

DBADJUST

EBCDIC、DBADJUST では、指定した *operation* に基づいて、*string* 内のすべての隣接する SI と SO (または SO と SI) 文字を調整します。

```
➡ DBADJUST — ( — string — ) ➡
                     |
                     | — operation —
```

有効な操作を以下に示します。大文字で強調表示された文字だけが必要であり、後続の文字はすべて無視されます。

ブランク

連続する文字をブランク (X'4040') に変更します。

除去

隣接する文字を除去します。これがデフォルトです。

EBCDIC の例

```
DBADJUST('<.A>.B>a<b','B')      ->    '<.A. .B>a  b'
DBADJUST('<.A>.B>a<b','R')      ->    '<.A.B>ab'
DBADJUST('<>.A.B>','B')          ->    '<. .A.B>'
```

DBBRACKET

EBCDIC、DBBRACKET では、DBCS 専用ストリングに SO および SI の大括弧を追加します。

```
➡ DBBRACKET — ( — string — ) ➡
```

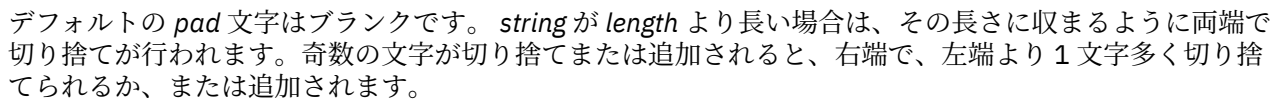
string が DBCS 専用ストリングでない場合は、SYNTAX エラーが発生します。つまり、入力ストリングは偶数のバイト長でなければならない、各バイトは有効な DBCS 値でなければならない。

```
DBBRACKET(' .A.B')      ->    '<.A.B>'
```

```
DBBRACKET('abc')        ->    SYNTAX error
```

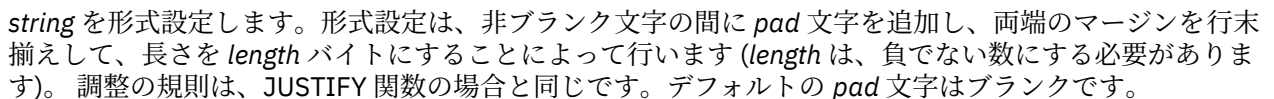
```
DBBRACKET('<.A.B>')      ->    SYNTAX error
```

DBCENTER は、*string* が中央に置かれた長さ *length* の文字列を返します。必要に応じて、長さを補うために *pad* 文字が追加されます。



DBCENTER(' <.A.B.C>',4)	->	' <.B>'
DBCENTER(' <.A.B.C>',3)	->	' <.B>'
DBCENTER(' <.A.B.C>',10,'x')	->	'xx<.A.B.C>xx'
DBCENTER(' <.A.B.C>',10,'x','Y')	->	'x<.A.B.C>x'
DBCENTER(' <.A.B.C>',4,'x','Y')	->	'<.B>'
DBCENTER(' <.A.B.C>',5,'x','Y')	->	'x<.B>'
DBCENTER(' <.A.B.C>',8,'<.P>')	->	' <.A.B.C>'
DBCENTER(' <.A.B.C>',9,'<.P>')	->	' <.A.B.C.P>'
DBCENTER(' <.A.B.C>',10,'<.P>')	->	'<.P.A.B.C.P>'
DBCENTER(' <.A.B.C>',12,'<.P>','Y')	->	'<.P.A.B.C.P>'

DBCJUSTIFY は *string* を形式設定します。形式設定は、非空白文字の間に *pad* 文字を追加し、両端のマージンを行末揃えして、長さを *length* バイトにすることによって行います (*length* は、負でない数にする必要があります)。



```
DBCJUSTIFY(' <><AA  BB><CC>',20,, 'Y')
-> ' <AA>      <BB>      <CC>'

DBCJUSTIFY(' <><  AA  BB><  CC>',20, ' <XX>', 'Y')
-> ' <AAXXXXXXBBXXXXXXCC>'

DBCJUSTIFY(' <><  AA  BB><  CC>',21, ' <XX>', 'Y')
-> ' <AAXXXXXXBBXXXXXXCC>'
```

```

DBCJUSTIFY('<>< AA BB>< CC>',11,'<XX>','Y')
-> '<AAXXXXBB>'

DBCJUSTIFY('<>< AA BB>< CC>',11,'<XX>','N')
-> '<AAXBBXXCC>'

```

DBLEFT

DBLEFT は、*string* の左端の *length* 個の文字を含む長さ *length* のストリングを戻します。

➡ DBLEFT — (— *string* — , — *length* —) ➡

戻されるストリングは、必要に応じて、右側で *pad* 文字が埋め込まれます (または切り捨てられます)。デフォルトの *pad* 文字はブランクです。

option で、カウント規則を制御できます。Y を指定すると、混合ストリング内の SO と SI がそれぞれ 1 つとしてカウントされます。N を指定すると、SO も SI もカウントされません。これがデフォルトです。

EBCDIC の例

```

DBLEFT('ab<.A.B>',4)      -> 'ab<.A>'
DBLEFT('ab<.A.B>',3)      -> 'ab'
DBLEFT('ab<.A.B>',4,'x','Y') -> 'abxx'
DBLEFT('ab<.A.B>',3,'x','Y') -> 'abx'
DBLEFT('ab<.A.B>',8,'<.P>') -> 'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>') -> 'ab<.A.B.P>'
DBLEFT('ab<.A.B>',8,'<.P>','Y') -> 'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>','Y') -> 'ab<.A.B>'

```

DBRIGHT

DBRIGHT は、*string* の右端の *length* 個の文字を含む長さ *length* のストリングを戻します。

➡ DBRIGHT — (— *string* — , — *length* —) ➡

戻されるストリングの左側には、必要に応じて、*pad* 文字が埋め込まれます (あるいは切り捨てられます)。デフォルトの *pad* 文字はブランクです。

option で、カウント規則を制御できます。Y を指定すると、混合ストリング内の SO と SI がそれぞれ 1 つとしてカウントされます。N を指定すると、SO も SI もカウントされません。これがデフォルトです。

EBCDIC の例

```

DBRIGHT('ab<.A.B>',4)      -> '<.A.B>'
DBRIGHT('ab<.A.B>',3)      -> '<.B>'
DBRIGHT('ab<.A.B>',5,'x','Y') -> 'x<.B>'
DBRIGHT('ab<.A.B>',10,'x','Y') -> 'xxab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>') -> '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',9,'<.P>') -> '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>','Y') -> 'ab<.A.B>'
DBRIGHT('ab<.A.B>',11,'<.P>','Y') -> 'ab<.A.B>'
DBRIGHT('ab<.A.B>',12,'<.P>','Y') -> '<.P>ab<.A.B>'

```


DBTOSBCS

DBTOSBCS は、ストリング *string* 内の、渡された有効な DBCS 文字 (DBCS ブランクを含む) のすべてを対応する SBCS の同等のものに変換します。

➡ DBTOSBCS — (— *string* —) ➡

その他の DBCS 文字およびすべての SBCS 文字は、変更されません。EBCDIC では、SO および SI の大括弧が、必要に応じて除去されます。

EBCDIC の例

```
DBTOSBCS('<.S.d>/<.2.-.1>')    -> 'Sd/2-1'
DBTOSBCS('<.X. .Y>')            -> '<.X> <.Y>'
```

注: 上記の例で .d は、SBCS の d に対応する DBCS 文字です。しかし、.X と .Y は、対応する SBCS 文字がないため、変換されません。

DBUNBRACKET

EBCDIC では、DBUNBRACKET は SO と SI の対で囲まれた DBCS 専用ストリング *string* から、SO と SI の対を除去します。*string* が大括弧で囲まれていない場合には、構文エラーになります。

➡ DBUNBRACKET — (— *string* —) ➡

EBCDIC の例

```
DBUNBRACKET('<.A.B>')            -> '.A.B'
DBUNBRACKET('ab<.A>')           -> SYNTAX error
```

DBVALIDATE

DBVALIDATE は、*string* が有効な混合ストリングまたは SBCS ストリングであれば、1 を戻します。それ以外の場合は、0 を戻します。

➡ DBVALIDATE — (— *string* — , — 'C' —) ➡

混合ストリングの妥当性検査の規則は、以下のとおりです。

- 有効な DBCS 文字コードだけである。
- DBCS ストリングの長さが、偶数バイトである。
- EBCDIC のみ: SO と SI が正しくペアになっている。

EBCDIC では、C が省略されると、実行されている実装環境の有効範囲内に収まっているかどうかを確認するために、各 DBCS 文字の左端のバイトだけが検査されます (つまり、EBCDIC では、左端のバイトの範囲は X'41' から X'FE' までです)。

EBCDIC の例

```
z='abc<de'
DBVALIDATE('ab<.A.B>')          -> 1
DBVALIDATE(z)                   -> 0

y='C1C20E111213140F'X
DBVALIDATE(y)                   -> 1
DBVALIDATE(y,'C')               -> 0
```


DBWIDTH

DBWIDTH は、*string* の長さをバイト数で戻します。

➤ DBWIDTH — (— *string* — , — *option* —) ➤

option で、カウント規則を制御できます。Y を指定すると、混合ストリング内の SO と SI がそれぞれ 1 つとしてカウントされます。N を指定すると、SO も SI もカウントされません。これがデフォルトです。

EBCDIC の例

```
DBWIDTH('ab<.A.B>', 'Y')    ->    8
DBWIDTH('ab<.A.B>', 'N')    ->    6
```


第 34 章 予約キーワードおよび特殊変数

キーワードを通常の記号として使用しても、それによって混乱することのない状況であればかまいません。

ここでは、正確な規則を説明します。

特殊変数には、RC、RESULT、および SIGL の 3 つがあります。

予約キーワード

REXX の自由構文では暗黙に、ある特定の文脈において言語処理プログラムが使用するために、一部の記号が予約されています。特定の命令の中で、命令を各部分に分けるためにシンボルが予約されている場合もあります。このようなシンボルは、キーワードと呼ばれます。

REXX キーワードの例としては、DO 命令内の WHILE、および IF または WHEN 文節の後に続く THEN (この場合、文節の終止符としての役割があります) があります。

上記の場合とは別に、文節内の最初のトークンで、しかもその後に = または : が続いていない単純記号についてのみ、命令のキーワードであるか否かが検査されます。文節内の他の場所であれば、キーワードと見なされることなく自由に記号を使用することができます。

ただし、ユーザーが REXX キーワードと同じ名前 (例えば、QUEUE) のホスト・コマンドまたはコマンドを実行することはお勧めできません。これは、プログラマーの REXX プログラムが、ある期間にわたって、制御できない環境で使用される可能性がある場合や、プログラマーがプログラムを完全にすきのないものにする必要がある場合に問題になることがあります。

そのような場合は、REXX プログラムを作成するときに、コマンド行の (少なくとも) 最初のワードを引用符で囲んでください。以下に例を示します。

```
'SCRNINFO'
```

引用符を用いると、効率が上がるという利点もあります。さらに、この形式では、SIGNAL ON NOVALUE 条件を使用して EXEC の保全性を検査することができます。

また、このようなコマンド・ストリングの前に引用符を 2 つ重ねて指定する方法もあります。この結果、ヌル・ストリングがコマンド・ストリングの前に連結されます。以下に例を示します。

```
''SCRNINFO
```

3 つ目の方法は、式全体 (または最初の記号) を括弧で囲むものです。以下に例を示します。

```
(SCRNINFO)
```

どの方式にするかは、プログラマー個人の選択です。REXX 言語を使用する上で、これらの手段の選択が強制されることはありません。

特殊変数

RC、RESULT、SIGL は特殊変数です。

言語処理プログラムは、下記の 3 つの特殊変数を自動的にセットすることができます。

RC

実行されたホスト・コマンド (またはサブコマンド) からの戻りコードがセットされます。SIGNAL イベント SYNTAX、ERROR、FAILURE が発生した後に、そのイベントに対応するコード、つまり、構文エラー番号 (411 ページの『第 31 章 エラー番号とメッセージ』を参照) またはコマンド戻りコードが RC に設定されます。RC は、イベント NOVALUE または HALT の後では変更されません。

注: デバッグ・モードから手動でホスト・コマンドを実行しても、RC の値は変わりません。

RESULT

サブルーチン内の RETURN 命令で式を指定すると、そのサブルーチンが呼び出されたときに、RETURN 命令によってセットされます。RETURN 命令に式が指定されていなければ、RESULT はドロップされません (初期設定されません)。

SIGL

制御が最後にラベルに移動したときに実行中だった文節の行番号が入ります。(SIGNAL、CALL、内部関数呼び出し、またはエラー条件がトラップされることによって、この番号がセットされます。)

上記の変数には、いずれも初期値がありません。上記の変数は、その他のあらゆる変数と同様、変更することが可能であり、PROCEDURE 命令および DROP 命令は通常どおり、それらに作用します。

REXX プログラムでは、この他にも一定の情報を常に使用することができます。これには、プログラムが呼び出された名前と、プログラムのソース (PARSE SOURCE 命令を使用して使用可能。[179 ページの『PARSE』](#)を参照。)が含まれます。PARSE SOURCE 出力は、ストリング CICS の後に呼び出しタイプ、大文字での EXEC の名前、ファイルの名前、および実行中の PDS または DDNAME/メンバーで構成されます。これらの後には、プログラムが呼び出されたときに使用された名前および初期 (デフォルト) コマンド環境が続きます。

また、PARSE VERSION ([179 ページの『PARSE』](#)を参照) は、実行中の言語処理プログラム・コードのバージョンと日付が表示されるようにします。組み込み関数の TRACE および ADDRESS は、それぞれ、現行のトレース設定値および環境名を戻します。

組み込み関数の DIGITS、FORM、FUZZ を使用すると、NUMERIC 関数の現行設定値を取得できます。

第 35 章 デバッグ補助機能

このセクションでは、問題の対話式デバッグ、実行の中断、およびトレースの制御について説明します。

プログラムの対話式デバッグ

デバッグ機能を使用すると、プログラムの実行を対話式に制御することができます。

TRACE 動作を接頭部 ? を指定したもの (例えば、TRACE ?A や TRACE 組み込み関数など) に変更すると、対話式デバッグがオンになり、対話式デバッグがアクティブであることがユーザーに示されます。プログラム内のそれ以降の TRACE 命令は無視され、コンソールからトレースされるほぼすべての命令の後で、言語処理プログラムは停止します (例外については、以下の説明を参照してください)。言語処理プログラムが停止すると、画面の下隅に READ で示され、3 つのデバッグ・アクションが使用可能になります。

1. ヌル行を入力する (文字もブランクも入れない) と、言語処理プログラムは、デバッグ入力のために次に停止するまで、実行を続けます。したがって、ヌル行を繰り返し入力すると、停止点から停止点までのステップが実行されます。例えば、TRACE ?A の場合、ヌル行を繰り返し入力することは、プログラム全体をとおして 1 ステップずつ実行することになります。
2. 等号 (=) を入力する (ブランクは入れない) と、言語処理プログラムは、最後にトレースした文節を再実行します。例えば、IF 文節が間違っただけのブランチを行おうとしている場合、IF 文節に影響を与える変数 (複数可) の値を変更してから、IF 文節を再実行します。

文節が実行されると、言語処理プログラムは再び停止します。

3. その他のものを入力すると、1 つまたは複数の文節の行として扱われ、即座に処理されます。 (つまり、プログラム内に DO; 行; END; が入力された場合と同様に処理されます)。INTERPRET 命令と同じ規則が適用されます (例えば、DO-END 構成は完全でなければならない)。命令の内部に構文エラーがあると、標準のメッセージが表示され、再入力するように求められます。同様に、ストリングの処理中は、不注意から制御が移動することがないように、他の SIGNAL 条件はすべて使用不可にされます。

ストリングの実行中は、ホスト・コマンドからの非ゼロが戻りコードが表示されることを除き、トレースは実行されません。ホスト・コマンドは常に実行されます (すなわち、TRACE 命令に接頭部 ! があっても影響されませんが、RC 変数はセットされません)。

ストリングが処理された後は、TRACE 命令が入力されていなければ、その後にデバッグ入力があるまで、言語処理プログラムは再び停止します。この前者の (TRACE 命令が入力されている) 場合、言語処理プログラムは、 (必要であれば) トレース動作を即座に変更してから、次の停止点 (それが存在する場合) まで実行を続けます。トレース・アクションを変更 (例えば、「すべて (All)」から「結果 (Results)」) にしてから、その命令を再実行するには、組み込み関数 TRACE (218 ページの『TRACE』を参照) を使用する必要があります。例えば、CALL TRACE I を指定すると、トレース動作が I に変更され、停止後のステートメントの再実行が可能になります。TRACE 命令で接頭部を使用した場合、あるいは TRACE 0 を入力するかオプションを指定せずに TRACE を入力した場合には、対話式デバッグが動作中であればオフになります。

数字形式の TRACE 命令を使用すると、デバッグ入力のための停止をせずに、プログラムのセクションを実行することができます。TRACE n (つまり、正の結果) は、実行を続行し、 (対話式デバッグがアクティブであるか、またはアクティブになった場合に) 次の n 回の停止をスキップできるようにします。TRACE -n (つまり、負の結果) を入力すると、停止せずに実行を続けて、トレースされる予定だった n 個の文節のトレースを禁止することができます。

TRACE 命令によって選択されたトレース動作は、サブルーチン呼び出しのたびに、保管および復元されます。この意味は、プログラムの実行中に (例えば、Results をトレースするための TRACE ?R を使用した後)、トレースの不要なサブルーチンに入った場合は、TRACE 0 を入力してトレースをオフにできるということです。それ以降はサブルーチン内の命令はトレースされませんが、呼び出し元へ戻ると、トレースが復元されます。

同様に、サブルーチンだけをトレースしたい場合には、そのサブルーチンの先頭に TRACE ?R 命令を入力します。そのルーチンのトレースが終わると、トレースの元の状況が復元され、 (サブルーチンに入るとき

にトレースがオフであった場合には、次にそのサブルーチンに入るまでは、トレース (および対話式デバッグ) がオフになります。

どのような命令も対話式デバッグで実行できるので、実行をかなり制御できます。

注: 対話式デバッグ中に、PULL ステートメントだけでなく、対話式デバッグによっても停止が起こる可能性があります。PULL ステートメントを使用するプログラムの場合は、個々の停止についてその理由を知る必要があります。プログラム内では、PULL ステートメントはしばしば SAY ステートメントとペアになっています。PULL のトレース行の後の停止では、ユーザーは PULL に対してデータを 入力する必要があります (この停止は、PULL に対してデータを 入力するための特別なものです)。これと対応する SAY ステートメントの後の停止では、ユーザーがデータを 入力する必要はありません (これは、対話式デバッグの停止です)。

注: 文節の中には安全に再実行できないものもあるので、それらの文節の場合は、トレースしていたとしても、言語処理プログラムはその後で停止することはしません。該当する HTTP ヘッダーは、以下のとおりです。

- 2 回目以降のループのすべての反復 DO 文節。
- すべての END 文節 (いずれにしても有効な停止場所とはなりません)。
- すべての THEN、ELSE、OTHERWISE、またはヌル文節。
- すべての RETURN および EXIT 文節。
- すべての SIGNAL および CALL 文節 (言語処理プログラムは、ターゲット・ラベルをトレースした後で停止します)。
- CALL ON または SIGNAL ON がトラップする条件を提示するすべての文節 (CALL または SIGNAL のターゲット・ラベルをトレースした後で停止します)。
- 構文エラーの原因となるすべての文節。 (このような文節は、SIGNAL ON SYNTAX によってトラップすることはできますが、再実行はできません。)

例

```
Say expr      /* displays the result of evaluating the      */
              /* expression.                             */

name=expr     /* alters the value of a variable.                  */

Trace 0       /* (or Trace with no options) turns off                  */
              /* interactive debug and all tracing.                 */

Trace ?A      /* turns off interactive debug but continues             */
              /* tracing all clauses.                                  */

Trace L       /* makes the language processor pause at labels          */
              /* only. This is similar to the traditional            */
              /* "breakpoint" function, except that you            */
              /* do not have to know the exact name and            */
              /* spelling of the labels in the program.          */

exit          /* stops execution of the program.                      */

Do i=1 to 10; say stem.i; end
              /* displays ten elements of the array stem.         */
```

実行への割り込みおよびトレースの制御

標準の CICS 機能を使用して、REXX EXEC (REXX トランザクション) に割り込むことができます。

適切な権限があれば、CEMT SET TASK PURGE コマンドを発行して EXEC を停止できます。詳しくは、[CEMT SET TASK](#) を参照してください。

第 36 章 基本マッピング・サポートの例

この例は、REXX/CICS 環境で CICS 基本マッピング・サポート (BMS) を使用するための手順を示しています。

1. BMS マップは、アセンブルし、CICS ライブラリーにリンクする必要があります。このライブラリーは、CICS 領域始動 JCL の LIBDEF 内になければなりません。
2. REXX/CICS CONVTPMAP コマンドを使用してファイル構造を生成する予定の場合は、BMS マップをアセンブルしてマップ DSECT を作成する必要があります。
3. BMS マップは、リソース定義オンライン (RDO) を使用して CICS に対して定義する必要があります。
4. 画面との間でデータの読み取りやデータの送信を行う場合には、マップ入出力域の長さに合う GETMAIN CICS ストレージが必要です。このストレージは、ヌルに初期設定する必要があります。
5. ファイル構造内のフィールドが複数のラベルで表される場合、そのフィールドを参照する際には、構造内の該当フィールドを参照する最後のラベルを使用する必要があります。

この例では、BMS マップ PANELG を使用します。マップ定義は以下のとおりです。

```
TITLE 'PANEL GROUP FOR REXX/CICS' 00000010
PRINT ON,NOGEN 00000020
PANELG DFHMSD TYPE=MAP,LANG=ASM,MODE=INOUT,STORAGE=AUTO,SUFFIX= 00000030
        TITLE 'TEST PANEL FOR REXX/CICS' 00000040
DPANEL1 DFHMDI SIZE=(24,80),CTRL=(FREEKB),MAPATTS=(COLOR,HILIGHT), *00000050
        DSATTS=(COLOR,HILIGHT),COLUMN=1,LINE=1,DATA=FIELD, *00000060
        TIOAPFX=YES,OBFT=NO 00000070
        DFHMDI POS=(1,1),LENGTH=1,ATTRB=(PROT,BRT) 00000080
        DFHMDI POS=(5,27),LENGTH=22,INITIAL='REXXCICS HEADER PANEL1', *00000090
        ATTRB=(PROT,NORM) 00000100
        DFHMDI POS=(5,73),LENGTH=6,INITIAL='PANEL1',ATTRB=(PROT,NORM) 00000110
        DFHMDI POS=(9,6),LENGTH=25, *00000120
        INITIAL='PLEASE ENTER YOUR USERID:',ATTRB=(PROT,NORM) 00000130
* DUSERID 00000140
DUSERID DFHMDI POS=(9,32),LENGTH=8,ATTRB=(UNPROT,BRT,IC,FSET) 00000150
        DFHMDI POS=(9,41),LENGTH=1,ATTRB=(PROT,NORM) 00000160
        DFHMDI POS=(14,6),LENGTH=4,INITIAL='MSG:',ATTRB=(PROT,NORM) 00000170
* DMSG 00000180
DMSG DFHMDI POS=(14,11),LENGTH=29,ATTRB=(UNPROT,NORM,FSET) 00000190
        DFHMDI POS=(14,41),LENGTH=1,ATTRB=(PROT,NORM) 00000200
        DFHMSD TYPE=FINAL 00000210
END 00000220
```

マップ DSECT は以下のとおりです。DSECT は、USER.REXXCICS.MAPS(PANELG) という名前の MVS PDS にあります。

```
DS      0H      ENSURE ALIGNMENT
DPANEL1S EQU *      START OF MAP DEFINITION
DS      12C      TIOA PREFIX
SPACE
DUSERIDL DS      CL2 . INPUT DATA FIELD LEN
DUSERIDF DS      0C . DATA FIELD FLAG
DUSERIDA DS      C . DATA FIELD ATTRIBUTE
DUSERIDC DS      C . COLOUR ATTRIBUTE
DUSERIDH DS      C . HIGHLIGHTING ATTRIBUTE
DUSERIDI DS      0CL8 . INPUT DATA FIELD
DUSERIDO DS      CL8 . OUTPUT DATA FIELD
SPACE
DMSGGL DS      CL2 . INPUT DATA FIELD LEN
DMSGGF DS      0C . DATA FIELD FLAG
DMSGGA DS      C . DATA FIELD ATTRIBUTE
DMSGGC DS      C . COLOUR ATTRIBUTE
DMSGGH DS      C . HIGHLIGHTING ATTRIBUTE
DMSGGI DS      0CL29 . INPUT DATA FIELD
DMSGGO DS      CL29 . OUTPUT DATA FIELD
SPACE
DPANEL1E EQU *      END OF MAP DEFINITION
ORG DPANEL1S      ADDRESS START OF MAP
* CALCULATE MAPLENGTH, ASSIGNING A VALUE OF ONE WHERE LENGTH=ZERO
DPANEL1L EQU DPANEL1E-DPANEL1S
DPANEL1I DS      0CL(DPANEL1L+1-(DPANEL1L/DPANEL1L))
```

```

DPANEL10 DS 0CL(DPANEL1L+1-(DPANEL1L/DPANEL1L))
          ORG
* * * END OF MAP DEFINITION * * *
          SPACE 3
          ORG

```

CONVTMAP コマンドは、DSECT を受け取り、RFS に格納されたファイル構造を作成するために使用されます。このコマンドは、以下のように入力します。

```
'CONVTMAP USER.TEST(PANELG) POOL1:¥USERS¥USER1¥PANELG.DATA'
```

以下に、CONVTMAP によって作成されるファイル構造を示します。

```

00000 ***** TOP OF DATA *****
00001 DUSERIDL 13 2 C
00002 DUSERIDF 15 1 C
00003 DUSERIDA 15 1 C
00004 DUSERIDC 16 1 C
00005 DUSERIDH 17 1 C
00006 DUSERIDI 18 8 C
00007 DUSERIDO 18 8 C
00008 DMSGSL 26 2 C
00009 DMSGF 28 1 C
00010 DMSGA 28 1 C
00011 DMSGC 29 1 C
00012 DMSGH 30 1 C
00013 DMSGI 31 29 C
00014 DMSGO 31 29 C
00015 ***** BOTTOM OF DATA*****

```

以下の例は、EXEC BMSMAP1 です。これにより、ユーザー ID を求める単純なパネルが作成されます。

```

/* This EXEC uses CICS SEND and RECEIVE commands */
/* The panel has two fields USERID and a message */
/* field. The panel is initially displayed with */
/* a message - "USERID must be 8 characters" */

/* GETMAIN storage to be used for data mapping */
/* and initialize */
'PSEUDO OFF'
ZEROES = '00'x
'CICS GETMAIN SET(WORKPTR) LENGTH(90) INITIMG(ZEROES)'

VAR1 = 'USERID must be 8 characters'

/* Copy the REXX variable VAR1 to the GETMAINED storage */
'COPYR2S VAR1 WORKPTR 30'

/* Copy the storage area to REXX variable */
'COPYS2R WORKPTR X 0 90'

'CICS SEND MAP(PANELG) FREEKB ERASE FROM(X)'
'CICS RECEIVE MAP(PANELG) INTO(Y)'

/* Copy Y into the GETMAINED storage area and then copy the data */
/* to REXX variables using the file structure generated */
/* previously by the CONVTMAP command */
'COPYR2S Y WORKPTR 0 90'
'COPYS2R WORKPTR * POOL1:¥USERS¥USER1¥PANELG.DATA'

/* loop until the user enters a USERID exactly 8 characters in */
/* length */
do forever
  MUSERID = STRIP(DUSERIDO)
  if LENGTH(MUSERID) < 8 then
    do
      DMSGO = 'Please enter 8 char USERID'
      'COPYR2S * WORKPTR POOL1:¥USERS¥USER1¥PANELG.DATA'
      'COPYS2R WORKPTR X 0 90'
      'CICS SEND MAP(PANELG) FREEKB ERASE FROM(X)'
      'CICS RECEIVE MAP(PANELG) INTO(Z)'
      'COPYR2S Z WORKPTR 0 90'
      'COPYS2R WORKPTR * POOL1:¥USERS¥USER1¥PANELG.DATA'
    end
  else LEAVE
end
'SENDE'
say ' '

```



```
say 'Hello' DUSERID0', Welcome to REXX/CICS !!'  
exit
```

BMSMAP1 EXEC により、以下のパネルが作成されました。

```
                REXX/CICS HEADER PANEL1                PANEL1  
PLEASE ENTER YOUR USERID:  
MSG: USERID must be 8 characters
```

```
                REXX/CICS HEADER PANEL1                PANEL1  
PLEASE ENTER YOUR USERID: TEST  
MSG: Please enter 8 character USERID
```

第 37 章 参考文献

さらに情報を見つけるには

CICS TS および REXX に関する詳細については、以下の資料を参照してください。

- [アプリケーション開発のリファレンス](#)には、アプリケーション・プログラミング・コマンドに関する情報を記載しています。
- [リファレンス: システム・プログラミング](#)には、システム・プログラミング・コマンドに関する情報を記載しています。
- 「*REXX Language, A Practical Approach to Programming*」(M. F. Cowlshaw 著) には、REXX 言語に関する一般情報が記載されています。
- [CICS supplied transactions descriptions](#) には、CEMT および CEDA に関する情報を記載しています。
- [z/OS MVS プログラミング: アセンブラー・サービスガイド](#).
- [z/OS MVS プログラミング: アセンブラー・サービス解説書 ABE-HSP](#).
- [z/OS MVS プログラミング: アセンブラー・サービス解説書 IAR-XCT](#).

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す。この資料の他の言語版を IBM から入手できる場合があります。ただし、これを入手するには、本製品または当該言語版製品を所有している必要がある場合があります。

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。IBM 製品、プログラムまたはサービスに代えて、IBM の知的所有権を侵害することのない機能的に同等のプログラムまたは製品を使用することができません。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。

〒 103-8510

東京都中央区日本橋箱崎町 19 番 21 号

日本アイ・ビー・エム株式会社

法務・知的財産

知的財産権ライセンス渉外

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態を提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様自身の責任でご使用ください。

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。

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 Director of Licensing

IBM Corporation

North Castle Drive, MD-NC119 Armonk,

NY 10504-1785

US

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。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関す

る実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者をお願いします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名前はすべて架空のものであり、類似する個人や企業が実在しているとしても、それは偶然にすぎません。

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めめかしたり、保証することはできません。これらのサンプル・プログラムは特定物として現存するままの状態を提供されるものであり、いかなる保証も提供されません。IBM は、お客様の当該サンプル・プログラムの使用から生ずるいかなる損害に対しても一切の責任を負いません。

プログラミング・インターフェース情報

本書は、REXX/CICS インタープリター用のプログラムの作成を支援することを目的としています。本書では、REXX for Customer Information Control System (REXX for CICS) が提供する汎用プログラミング・インターフェースとそれに関連するガイダンス情報について主に解説しています。汎用プログラミング・インターフェースにより、お客様は REXX for CICS のサービスを受けるプログラムを記述することができます。ただし、本書には、REXX for CICS が提供するプロダクト・センシティブ・プログラミング・インターフェースとそれに関連するガイダンス情報についても解説しています。プロダクト・センシティブ・プログラミング・インターフェースは、お客様がインストールを行う際に、REXX for CICS の診断、変更、モニター、修復、調整、チューニングなどのタスクを実行できるようにするものです。これらのインターフェースを使用すると、IBM のソフトウェア製品の詳細な設計や実装に対する依存関係が生じます。

プロダクト・センシティブ・プログラミング・インターフェースは、上記のような特殊な目的のためにのみ使用してください。詳細な設計や実装に依存するため、新しい製品リリースまたはバージョンで実行する場合、あるいは保守サービスの結果、そのようなインターフェースに対して記述されたプログラムの変更が必要になることが予想されます。

プロダクト・センシティブ・プログラミング・インターフェースとそれに関連するガイダンス情報は、章またはセクションの始まりの文によって、その出現箇所を識別できます。

商標

IBM、IBM ロゴおよび ibm.com® は、世界の多くの国で登録された International Business Machines Corporation の商標または登録商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、<http://www.ibm.com/legal/copytrade.shtml> をご覧ください。

Adobe、Adobe ロゴ、PostScript、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

インテル、Intel、Intel ロゴ、Intel Inside、Intel Inside ロゴ、Intel Centrino、Intel Centrino ロゴ、Celeron、Intel Xeon、Intel SpeedStep、Itanium、および Pentium は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。

Java™ およびすべての Java 関連の商標およびロゴは Oracle やその関連会社の米国およびその他の国における商標または登録商標です。

Linux® は、Linus Torvalds の米国およびその他の国における登録商標です。

Microsoft、Windows、Windows NT および Windows ロゴは、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。

製品資料に関するご使用条件

これらの資料は、以下のご使用条件に同意いただける場合に限りご使用いただけます。

適用範囲

IBM Web サイトの「ご利用条件」に加えて、以下のご使用条件が適用されます。

個人使用

これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布（頒布、送信を含む）または表示（上映を含む）することはできません。

商用使用

これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

権利

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。

IBM これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態でご提供され、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なしで提供されます。

IBM オンラインでのプライバシー・ステートメント

サービス・ソリューションとしてのソフトウェアも含めた IBM ソフトウェア製品 ("ソフトウェア・オファリング") では、製品の使用に関する情報の収集、エンド・ユーザーの使用感の向上、エンド・ユーザーとの対話またはその他の目的のために、Cookie はじめさまざまなテクノロジーを使用することがあります。多くの場合、ソフトウェア・オファリングにより個人情報が収集されることはありません。IBM の「ソフトウェア・オファリング」の一部には、個人情報を収集できる機能を持つものがあります。ご使用の「ソフトウェア・オファリング」が、これらの Cookie およびそれに類するテクノロジーを通じてお客様による個人情報の収集を可能にする場合、以下の具体的事項をご確認ください。

この「ソフトウェア・オファリング」が Cookie およびさまざまなテクノロジーを使用してエンド・ユーザーから個人を特定できる情報を収集する機能を提供する場合、お客様は、このような情報を収集するにあたって適用される法律、ガイドライン等を遵守する必要があります。これには、エンドユーザーへの通知や同意の要求も含まれますがそれらには限られません。

このような目的での Cookie を含む様々なテクノロジーの使用の詳細については、『[IBM プライバシー・ポリシー](#)』および『[IBM オンラインでのプライバシー・ステートメント](#)』の『"クッキー、Web ビーコン、その他のテクノロジー"』および『[IBM ソフトウェア製品および Software as a Service のプライバシー・ステートメント](#)』を参照してください。

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]

アクティブ・ループ [175](#)
アセンブラー・プログラム
 渡されるコマンド引数 [308](#)
アセンブラー・プログラムに渡されるコマンド引数 [308](#)
アドレス設定値 [163](#), [165](#)
アンダーフロー、算術演算の [248](#)
暗黙のセミコロン [146](#)
一時記憶域キュー (TSQ) [383](#)
一時的なコマンドの宛先変更 [163](#)
位置調整、JUSTIFY 関数によるテキストの [210](#)
一般的な概念 [141](#)
インストールの検証 [119](#)
埋め込み文字、定義 [196](#)
英数字検査、DATATYPE による [202](#)
永続的なコマンドの宛先変更 [163](#)
エラー・コード [411](#)
エラー・メッセージ [12](#)
オーバーフロー、算術演算の [248](#)
オーバーレイ、あるストリングを別のストリングへ [212](#)
オンライン・リソース定義 (RDO) [473](#)
オンライン・リソース定義トランザクション (CEDA) [3](#)

[カ行]

解釈実行、データの [174](#)
解析でのストリング・パターン [86](#)
解析での相対数字パターン [86](#)
解析でのパターン [86](#)
解析での変数ストリング・パターン [86](#)
解析における高度な手法 [236](#)
概要
 CICS の [133](#)
 仮定、XEDIT [11](#)
 間接計算、データの [174](#)
 完全修飾ファイル ID [255](#)
 記号およびストリング、DBCS における [454](#)
 疑似乱数関数、RANDOM の [213](#)
 基本マッピング・サポート (BMS) [161](#), [368](#)
 行の長さおよび幅、端末の [211](#)
 切り捨て、数の [219](#)
 クライアント EXEC の例 [322](#)
 クライアント/サーバー (client/server) [127](#)
 クライアント/サーバー・サポート
 REXX/CICS [321](#)
 繰り返し、COPIES を使用してのストリングの [201](#)
 グループ化、繰り返し実行する命令の [168](#)
 計測機能インターフェース (IFI) [313](#)
 現行ディレクトリー [284](#), [300](#)
 現行の端末行の長さ [211](#)
 厳密な等号演算子 [149](#)
 厳密な非等号演算子 [149](#)
 厳密なより大演算子 [149](#)

 厳密により小演算子 [149](#)
 厳密により小でない演算子 [149](#)
 厳密により小または等しい演算子 [149](#)
 厳密により大でない演算子 [149](#)
 厳密により大または等しい演算子 [149](#)
 厳密比較 [149](#)
 項およびデータ [147](#)
 公開された変数 [181](#)
 工学表記法 [246](#)
 構成
 REXX [115](#)
 構造および構文 [141](#)
 構文解析でのブレースホルダー [15](#), [85](#)
 構文解析の概説 [238](#)
 構文解析命令 [235](#)
 構文図
 REXX [139](#)
 コーディング・スタイル [109](#)
 コード・ページ [142](#)
 コピー、COPIES を使用してのストリングの [201](#)
 コマンド環境のリセット [349](#)
 コマンド間のマッピング [349](#)
 コマンド実行セキュリティ [285](#)
 コマンド・レベル・インタープリター・トランザクション (CECI) [129](#)
 混合 DBCS ストリング [202](#)
 コンマ [86](#)

[サ行]

サーバー EXEC の例 [322](#)
最下層、実行中に達したプログラムの [172](#)
再帰呼び出し [165](#)
サブキーワード [153](#)
サブストリング [215](#)
サブディレクトリー [283](#), [299](#)
サブルーチン内での複数のストリングの構文解析 [236](#)
サブルーチンの呼び出し時に保管される例外条件 [165](#)
算術演算 [241](#)
シーケンス、XRANGE を使用した照合 [222](#)
時間、真夜中から計算された [216](#)
式
 算術演算 [21](#)
 定義 [21](#)
 トレース [95](#), [97](#)
 比較 [24](#)
 連結 [28](#)
識別、ユーザーの [219](#)
字下げ、トレース中の [187](#)
システム管理者 [130](#)
事前定義変数 [313](#), [315](#), [318](#)
自動サーバー始動 (ASI) [321](#)
シフトアウト (SO) 文字 [453](#), [457](#)
シフトイン (SI) 文字 [453](#), [457](#)
重要、STORAGE 関数 [224](#)
障害の定義 [156](#)
条件がトラップされたときに取られるアクション [250](#)

条件がトラップされないときに取られる処置 [250](#)
照合シーケンス、XRANGE を使用した [222](#)
初期設定されていない変数 [153](#)
深夜 12 時から計算された分数 [216](#)
数値の中の 10 の累乗 [143](#)
スタイル、コーディング [109](#)
ストリング
 DBCS [453](#)
ストリングからのワード [221](#)
ストリングの解析 [236](#)
ストリング・パターンと定位置パターン [236](#)
ストリング・パターンと定位置パターンの組み合わせ [236](#)
スペーシング、フォーマット、SPACE 関数 [214](#)
制御変数 [41](#)
精度、算術演算の [242](#)
製品の概説 [133](#)
セキュア
 ファイル・アクセス [285](#)
セキュリティ [129](#), [285](#), [300](#)
セキュリティ出口 [124](#)
セット・バッファ・アドレス (SBA) (set buffer address (SBA)) [224](#)
選択、ABBREV 関数によるデフォルトの [196](#)
相対定位置パターン [230](#)
想定したコマンドの宛先 [163](#)
挿入、あるストリングを別のストリングへ [209](#)
そろえ、テキストの右、RIGHT 関数 [214](#)

[タ行]

代数優先順位 [151](#)
タイプ、DATATYPE による検査、データの [202](#)
対話式デバッグ [187](#), [471](#)
探索、ストリングから句を [207](#)
中止、使用プログラムの [172](#)
定位置パターンが入っているテンプレート [230](#)
定義
 サブディレクトリー [283](#)
 ディレクトリー ID [283](#)
 パネル [324](#)
 ファイル・プール [283](#)
 ルート・ディレクトリー [283](#)
停止 (HI) 即時コマンド、解釈 [471](#)
停止のトラップ [249](#)
定数 [20](#)
定数シンボル [154](#)
ディレクトリー ID [283](#), [299](#)
テキスト・エディター [255](#)
テキスト間隔の設定 [214](#)
デバッグ補助機能 [471](#)
テンプレート
 解析 [86](#)
特殊な場合 [236](#)
特殊な変更、制御の流れの [249](#)
特殊変数 [469](#)
トランザクション ID [129](#), [255](#)
トレース開始 (TS) 即時コマンド [471](#)
トレース終了 (TE) 即時コマンド [471](#)
トレースバック、構文エラーの [187](#)

[ナ行]

内容アドレス可能ストレージ [154](#)

並べ替え、TRANSLATE 関数によるデータの [218](#)
ニブル [143](#)
ネスト、制御構造の [165](#)

[ハ行]

場合の解析 [236](#)
排他 OR 演算子 [27](#), [150](#)
排他 OR、文字ストリングの [199](#)
パターンでの等号 [87](#)
バック、X2C によるストリングの [223](#)
バックスラッシュの使用 [143](#), [149](#)
パネル
 オブジェクト生成プログラム [324](#)
 生成 [330](#)
 定義 [324](#)
 入出力 [330](#)
パネル機能の状態コード [338](#)
パネル機能の戻りコード [335](#)
反復ループ [41](#)
日付およびバージョン、言語処理プログラムの [179](#)
ビット、DATATYPE を使用して検査される [202](#)
非等号演算子 [149](#)
等しいこと、テスト [149](#)
等しくないこと、テスト [149](#)
秒、真夜中から計算された [216](#)
評価、式の [147](#)
ファイル・アクセス・セキュリティ [285](#)
ファイル・システム
 コマンド [285](#)
ファイル・タイプ拡張子 [157](#)
ファイル・プール
 ルート・ディレクトリー [283](#)
ファイル名、タイプ、モード、プログラムの [179](#)
ファイル・リスト・ユーティリティ (FLST) [291](#)
フィーチャー
 REXX [3](#)
復元、変数の [172](#)
副次式 [147](#)
複数のストリング [88](#), [236](#)
不正確な数値比較 [245](#)
浮動小数 [246](#)
ブランク、処理 [85](#)
ブランク行 [10](#)
ブレースホルダー [85](#)
ブレースホルダーとしてのピリオドの使用 [85](#)
フローチャート [38](#)
プログラム ID [5](#), [141](#)
プログラムの修正 [108](#)
プログラムの入力 [11](#)
プログラム・ロード・テーブル (PLT) [124](#)
ページ、コード [142](#)
ヘルプ・ファイルの作成 [101](#), [120](#)
変数ストリング・パターン [233](#)
変数の値、VALUE による取り出し [219](#)
変数の派生名 [154](#)
包含 OR 演算子 [27](#), [150](#)
保護、変数の [181](#)
保護桁 [243](#)
補助リスト [172](#), [181](#)
ホスト・コマンド環境 [92](#)

[マ行]

マスター端末トランザクション (CEMT) [3](#)
末尾 [154](#)
マルチウェイ呼び出し [165](#), [187](#)
未定ループ [168](#)
無限ループ [168](#)
無条件の中止、使用プログラムの [172](#)
戻りコード [423](#)

[ヤ行]

有効数字、算術演算の [242](#)
ユーザー、識別 [219](#)
優先順位、演算子の [151](#)
予約、キーワードの [469](#)
より大きくない演算子 [149](#)
より小演算子 (<) [149](#)
より小か等しい演算子 (<=) [149](#)
より小またはより大演算子 (<>) [149](#)
より大演算子 [149](#)
より大か等しい演算子 (>=) [149](#)
より大またはより小演算子 (><) [149](#)
より小さくない演算子 [149](#)

[ラ行]

乱数関数、RANDOM の [213](#)
リソース定義のインストール [115](#)
リテラル・ストリング [6](#), [143](#)
隣接 [29](#), [148](#)
ルート・ディレクトリー [283](#), [299](#)
連想ストレージ [154](#)

[ワ行]

割り当て解除、変数の [172](#)
割り込み命令 [50](#)

[数字]

10 の累乗 [246](#)

A

ALLOC コマンド [350](#)
AND 論理演算子 [150](#)
AND、文字ストリングの [198](#)
ARBCHAR コマンド [258](#)
ARG オプション、PARSE 命令の [179](#)
ARG 関数による引数のチェック [197](#)
ARG 命令 [15](#), [75](#)
ARGS コマンド [258](#)
AUTH コマンド [285](#), [286](#)
AUTHUSER コマンド [123](#), [124](#), [129](#), [350](#)

B

BACKWARD コマンド [259](#)
BASE オプション、DATE 関数の [203](#)
BASSM アセンブラー命令 [308](#)
BMS の例 [473](#)

BMSMAP1 EXEC [473](#)
BOTTOM コマンド [259](#)
BSM アセンブラー命令 [308](#)

C

C2S コマンド [308](#), [321](#), [373](#)
CALL 命令 [51](#), [68](#)
CANCEL コマンド [260](#), [275](#)
CASE コマンド [260](#)
CATMOUSE EXEC [104](#)
CD コマンド [157](#), [255](#), [284](#), [351](#), [379](#)
CEDA コマンド [352](#)
CEMT コマンド [353](#)
CENTURY オプション、DATE 関数の [203](#)
CHANGE コマンド [261](#)
CICEPROF マクロ [264](#)
CICGETV ルーチン [310](#)
CICPARMS 制御ブロック [309](#)
CICREX プログラム [124](#)
CICS
戻りコード [450](#)
CICS コマンド [91](#)
CICS 初期設定 JCL の更新 [117](#)
CICS の概要 [133](#)
CICS メッセージ
REXX エラー・コード [411](#)
CICSECX1 セキュリティー出口 [124](#)
CICSECX2 セキュリティー出口 [125](#)
CICSPROF exec [130](#)
CICSTART exec [123](#), [124](#)
CICSTART.PROC の更新 [116](#)
CICXPROF マクロ [264](#)
CKDIR コマンド [286](#), [300](#)
CKFILE コマンド [287](#)
CLD コマンド [300](#), [367](#)
CMDLINE コマンド [262](#)
COMPARE 関数 [200](#)
CON EXEC (練習問題) [104](#)
CONVTMAP コマンド [368](#)
COPY コマンド [287](#)
COPYR2S コマンド [369](#)
COPYS2R コマンド [371](#)
CTLCHAR コマンド [262](#)
CURLINE コマンド [263](#)

D

Db2 インターフェース [313](#)
Db2 の埋め込み [317](#)
DBADJUST 関数 [462](#)
DBBRACKET 関数 [462](#)
DBCENTER 関数 [463](#)
DBCJUSTIFY 関数 [463](#)
DBCS 記号の妥当性検査 [454](#)
DBLEFT 関数 [464](#)
DBRIGHT 関数 [464](#)
DBRLEFT 関数 [465](#)
DBRRIGHT 関数 [465](#)
DBTODBCS 関数 [465](#)
DBTOSBCS 関数 [466](#)
DBUNBRACKET 関数 [466](#)
DBVALIDATE 関数 [466](#)

DBWIDTH 関数 [467](#)
DEFCMD コマンド [129](#), [307](#), [321](#), [374](#)
DEFCMD コマンドでの CICS LINK オプション [308](#)
DEFCMD コマンドでの CICS LOAD オプション [308](#)
DEFSCMD コマンド [307](#), [376](#)
DEFTRNID コマンド [378](#)
DELETE コマンド [288](#), [301](#)
DIGITS オプション、NUMERIC 命令の [176](#), [242](#)
DIR コマンド [10](#), [379](#)
DISKR コマンド [288](#)
DISKW コマンド [283](#), [289](#)
DISPLAY コマンド [263](#)
DO 命令の BY 句 [168](#)
DO 命令の FOR 句 [168](#)
DO 命令の TO 句 [168](#)
DO...END 命令 [41](#)
DOWN コマンド [264](#)
DPATH コマンド [284](#)

E

EDIT コマンド [255](#), [264](#), [380](#)
EDITSVR [92](#)
ERROR 条件、SIGNAL 命令および CALL 命令の [252](#)
ETMODE [177](#), [454](#)
EUROPEAN オプション、DATE 関数の [203](#)
EXEC ID [8](#)
EXEC コマンド [265](#), [381](#)
EXECDB2 コマンド環境 [313](#)
EXECDROP コマンド [382](#)
EXECIO コマンド [383](#)
EXECLOAD コマンド [384](#)
EXECMAP コマンド [385](#)
EXECSQL コマンド環境 [313](#)
EXIT 命令 [50](#), [68](#)
EXMODE [454](#)
EXPORT コマンド [386](#)
EXPOSE オプション、PROCEDURE 命令の [181](#)
EXTERNAL オプション、PARSE 命令の [179](#)

F

FAILURE 条件、SIGNAL 命令および CALL 命令の [249](#), [252](#)
FIFO (先入れ / 先出し法) スタック化 [184](#)
FILE コマンド [266](#)
FILEPOOL コマンド [124](#), [387](#)
FIND 関数 [207](#)
FIND コマンド [267](#)
FLST
 トランザクション [298](#)
 EXEC [298](#)
FLST コマンド
 CANCEL [292](#)
 COPY [292](#)
 DELETE [293](#)
 DOWN [293](#)
 END [293](#), [298](#)
 EXEC [293](#)
 FLST [294](#)
 MACRO [295](#)
 PFKEY [296](#)
 REFRESH [296](#)
 RENAME [296](#)

FLST コマンド (続き)
 SORT [297](#)
 SYNONYM [298](#)
 UP [298](#)
FLSTSVR [92](#)
FOREVER repetitor、DO 命令の [168](#)
FORM 関数 [207](#)
FORWARD コマンド [268](#)
FREE コマンド [390](#)
FUZZ 関数 [209](#)

G

GET コマンド [268](#)
GETDIR コマンド [289](#)
GETPDS コマンド [269](#)
GETVERS コマンド [390](#)

H

HALT 条件、SIGNAL 命令および CALL 命令の [249](#), [252](#)
HELLO EXEC [11](#)
HELP コマンド [391](#)

I

IMPORT コマンド [391](#)
INDEX [209](#)
INPUT コマンド [269](#)

J

JOIN コマンド [270](#)
JULIAN オプション、DATE 関数の [203](#)

L

LEAVE 命令 [49](#)
LEFT コマンド [270](#)
LIFO (後入れ / 先出し法) スタック化 [183](#)
LINEADD コマンド [271](#)
LINEIN オプション、PARSE 命令の [179](#)
LISTCMD コマンド [393](#)
LISTPOOL コマンド [393](#)
LISTTRNID コマンド [394](#)
LPREFIX コマンド [271](#)
LPULL コマンド [301](#)
LPUSH コマンド [302](#)
LQUEUE コマンド [302](#)
LSRPOOL 定義の更新 [116](#)

M

MACRO コマンド [271](#)
MKDIR コマンド [283](#), [290](#), [299](#), [303](#)
MONTH オプション、DATE 関数の [203](#)
MSGLINE コマンド [272](#)

N

NOETMODE [177](#)
NOEXMODE [177](#)

Normal オプション、DATE 関数の [203](#)
NOT 演算子 [143](#), [150](#)
NOTYPING フラグ、エラー・メッセージの前にクリアされる
[411](#)
NULLS コマンド [273](#)
NUMBERS コマンド [273](#)
NUMERIC 命令の FORM オプション [176](#), [246](#)

O

OPTIONS 命令 [454](#)
OR、文字ストリングの [198](#)
ORDERED オプション、DATE 関数の [203](#)

P

PANEL コマンド [330](#), [331](#)
PARSE ARG 命令 [84](#)
PARSE PULL 命令 [84](#)
PARSE UPPER ARG 命令 [84](#)
PARSE UPPER PULL 命令 [84](#)
PARSE UPPER VALUE [84](#)
PARSE UPPER VAR [85](#)
PATH コマンド [157](#), [284](#), [394](#)
PFKEY コマンド [274](#)
PFKLINE コマンド [274](#)
PROCEDURE 命令 [73](#), [74](#)
PSEUDO コマンド [396](#)
PULL オプション、PARSE 命令の [179](#)
PULL 命令 [11](#), [14](#), [83](#)

Q

QQUIT コマンド [275](#), [277](#)
QUERY コマンド [276](#)
QUIT コマンド [277](#)

R

RC [469](#)
RDIR コマンド [290](#)
READ コマンド [303](#)
RENAME コマンド [290](#)
RESERVED コマンド [277](#), [291](#)
RESET コマンド [278](#)
RESULT [469](#)
RETURN 命令 [51](#), [68](#)
REXX Db2 インターフェースの構成 [121](#)
REXX 言語
 フィーチャー [3](#)
REXX プログラム ID [5](#), [8](#), [141](#)
REXX プログラムに渡されるコマンド引数 [308](#)
REXX/CICS [423](#)
REXX/CICS List System (RLS) [299](#)
REXX/CICS コマンド [349](#)
REXX/CICS コマンド定義機能 [307](#)
REXX/CICS テキスト・エディター [255](#)
REXX/CICS パネル機能 [323](#)
REXX/CICS ファイル・システム (RFS) [283](#)
REXXCICS [92](#)
RFS [92](#)
RFS コマンド [283](#), [397](#)
RFS ファイル・プールの形式設定 [118](#)

RFS ファイル・プールの作成 [115](#)
RIGHT コマンド [278](#)
RLS [92](#)
RLS コマンド
 CKDIR [300](#)
 DELETE [301](#)
 LPULL [301](#)
 LPUSH [302](#)
 LQUEUE [302](#)
 MKDIR [303](#)
 READ [303](#)
 VARDROP [304](#)
 VARGET [304](#)
 VARPUT [305](#)
 WRITE [305](#)

S

S2C コマンド [308](#), [321](#), [407](#)
SAVE コマンド [279](#)
SAY 命令 [5](#), [11](#), [12](#), [160](#)
SBCS ストリング [453](#)
SCRNINFO コマンド [402](#)
SELECT WHEN...OTHERWISE...END [38](#)
SET コマンド [403](#)
SETSYS コマンド [124](#), [405](#)
SIGL [469](#)
SIGNAL 命令 [54](#)
SIGNAL 命令の SYNTAX 条件 [249](#), [252](#)
SORT コマンド [279](#)
SOURCE オプション、PARSE 命令の [179](#)
SPLIT コマンド [280](#)
SQL ステートメント [91](#)
SQL の組み込み [313](#)
SQL 連絡域 (SQLCA) [316](#)
STANDARD オプション、DATE 関数の [203](#)
STRIP コマンド [280](#)
SUBSTR [215](#)
SYNONYM コマンド [281](#)
SYSSBA 関数 [224](#)

T

TERMID コマンド [407](#)
TOP コマンド [281](#)
TRACE 設定値の照会 [218](#)
TRACE 命令 [160](#)
TRUNC コマンド [282](#)

U

UNTIL 句、DO 命令の [168](#)
UP コマンド [282](#)
USA オプション、DATE 関数の [203](#)
USERS ディレクトリー [283](#), [299](#)

V

VALUE オプション、PARSE 命令の [179](#)
VAR オプション、PARSE 命令の [179](#)
VARDROP コマンド [304](#)
VARGET コマンド [304](#)
VARPUT コマンド [305](#)

verb
 [DEFINE 324](#)
 [PANEL 328](#)
VERSION オプション、PARSE 命令の [179](#)

W

WAITREAD コマンド [408](#)
WAITREQ コマンド [308](#), [321](#), [408](#)
WEEKDAY オプション、DATE 関数の [203](#)
WHILE 句、DO 命令の [168](#)
WRITE コマンド [299](#), [305](#)

X

XOR、文字ストリングの [199](#)
XOR、論理 [150](#)

[特殊文字]

DBCS

 関数の処理 [457](#)
 サポート [453](#)
 処理 [453](#)
 処理関数 [462](#)
 ストリング [453](#)
 説明 [453](#)
 名前、使用 [10](#)
 文字 [453](#)

REXX 命令

 構文規則 [5](#), [6](#)
 フォーマット [5](#)
 PROCEDURE [73](#)
 PULL [14](#)

受け渡し

 情報 [71](#)
 引数 [17](#)

エラー

 デバッグ [95](#), [97](#)

規則

 構文規則 [5](#), [6](#)
 フリー・フォーマット [5](#), [6](#)

構文

 REXX の規則 [5](#), [6](#)

トレース

 演算のトレース [95](#), [97](#)

フリー・フォーマット

 REXX 命令 [6](#)

プログラム

 エラー・メッセージ [12](#)
 構成 [113](#)
 情報の受け渡し [14](#)
 入力の受け取り [15](#)

メッセージ

 解釈 [12](#)

ADDRESS 命令

 例 [8](#)

CICREX1106E [421](#)

CICREX218E [412](#)

CICREX219E [412](#)

CICREX255T [412](#)

CICREX449E [412](#)

CICREX450E [412](#)

CICREX451E [413](#)

CICREX452E [413](#)

CICREX453E [413](#)

CICREX454E [413](#)

CICREX455E [413](#)

CICREX456E [414](#)

CICREX457E [414](#)

CICREX458E [414](#)

CICREX459E [414](#)

CICREX460E [415](#)

CICREX461E [415](#)

CICREX462E [415](#)

CICREX463E [415](#)

CICREX464E [416](#)

CICREX465E [416](#)

CICREX466E [416](#)

CICREX467E [416](#)

CICREX468E [416](#)

CICREX469E [417](#)

CICREX470E [417](#)

CICREX471E [417](#)

CICREX472E [417](#)

CICREX473E [417](#)

CICREX474E [418](#)

CICREX475E [418](#)

CICREX476E [418](#)

CICREX477E [418](#)

CICREX478E [418](#)

CICREX479E [419](#)

CICREX480E [419](#)

CICREX481E [419](#)

CICREX482E [419](#)

CICREX483E [419](#)

CICREX484E [419](#)

CICREX485E [420](#)

CICREX486E [420](#)

CICREX487E [420](#)

CICREX488E [420](#)

CICREX489E [420](#)

CICREX490E [421](#)

CICREX491E [421](#)

CICREX492E [421](#)

instruction

 PROCEDURE [73](#)

 SAY [5](#)

REXX

 文節 [8](#)

REXX プログラム

 演算子 [19](#)

 式 [19](#)

 変数 [19](#)

演算子

 算術演算 [21](#)

 連結 [28](#)

演習

 演算式の計算 [24](#)

 関数の作成 [80](#)

解析

 ワードへの [85](#)

 PARSE ARG [84](#)

 PARSE UPPER ARG [84](#)

開発

 REXX プログラム [1](#)

機能

機能 (続き)
 サブルーチンとの比較 [63](#)
 変数の保護 [73](#)
 コマンド
 引用符 [91](#)
 EDIT [91](#)
 RFS [91](#)
 RLS [91](#)
 作成
 REXX プログラム [5](#)
 サブルーチン
 関数との比較 [63](#)
 作成 [64](#)
 変数の保護 [73](#)
 算術演算子
 タイプ [21](#)
 実行
 REXX プログラム [5](#)
 ステートメント
 SQL [91](#)
 説明
 変数 [19](#)
 入力
 大文字への変換の防止 [16](#)
 文節
 REXX のタイプ [8](#)
 変数
 説明 [19](#)
 例
 完全修飾ファイル ID [283](#)
 現行ディレクトリー [284](#)
 サンプル・パネル [341](#)
 単純な REXX プログラム [5](#)
 デバッグ補助機能 [471](#)
 ADDRESS 命令 [8](#)
 DBCS の名前の使用 [10](#)
 ! 接頭部、TRACE オプションの [187](#)
 ? 接頭部、TRACE オプションの [187](#)
 .DEFINE verb [324](#), [325](#)
 .PANEL verb [328](#), [329](#)
 * (乗算演算子) [22](#), [149](#), [243](#)
 - トレース・フラグ [187](#)
 ** (累乗演算子) [22](#), [149](#)
 / (除算演算子) [22](#), [149](#), [243](#)
 // (剰余演算子) [22](#), [149](#)
 /= (非等号演算子) [149](#)
 /== (厳密に等しくない演算子) [149](#)
 & (AND 論理演算子) [27](#), [150](#)
 && (排他的 OR 演算子) [27](#), [150](#)
 % (整数除算演算子) [22](#), [149](#)
 + (加算演算子) [22](#), [149](#), [243](#)
 +++ トレース・フラグ [187](#)
 < (より小演算子) [25](#), [149](#)
 << (厳密なより小演算子) [149](#)
 <=< (厳密により小または等しい演算子) [149](#)
 <= (より小か等しい演算子) [25](#), [149](#)
 <> (より小またはより大演算子) [149](#)
 = (等号) [25](#)
 == (厳密な等号演算子) [25](#), [149](#), [243](#)
 > (より大演算子) [25](#), [149](#)
 >.> トレース・フラグ [187](#)
 >< (より大またはより小演算子) [25](#), [149](#)
 >= (より大か等しい演算子) [25](#), [149](#)
 >> (厳密なより大演算子) [149](#)
 >>= (厳密により大または等しい演算子) [149](#)
 >>> トレース・フラグ [187](#)
 >C> トレース・フラグ [187](#)
 >F> トレース・フラグ [187](#)
 >L> トレース・フラグ [187](#)
 >O> トレース・フラグ [187](#)
 >P> トレース・フラグ [187](#)
 >V> トレース・フラグ [187](#)
 ¬ (NOT 演算子) [25](#), [150](#)
 ¬< (より小でない演算子) [149](#)
 ¬<< (厳密により小でない演算子) [149](#)
 ¬= (非等号演算子) [149](#)
 ¬== (厳密な非等号演算子) [149](#)
 ¬> (より大でない演算子) [149](#)
 ¬>> (厳密により大でない演算子) [149](#)
 | (包含 OR 演算子) [27](#), [150](#)
 || (連結演算子) [29](#), [148](#)
 ¥ (NOT 演算子) [25](#), [27](#), [149](#)
 ¥< (より小さくない演算子) [25](#), [149](#)
 ¥<< (厳密により小でない演算子) [149](#)
 ¥= (非等号演算子) [25](#), [149](#)
 ¥== (厳密に等しくない演算子) [25](#), [149](#)
 ¥> (より大きくない演算子) [25](#), [149](#)
 ¥>> (厳密により大でない演算子) [149](#)

