

CICS Transaction Server용 REXX  
버전 1 릴리스 1

사용자 안내서 및 참조서



#### 참고

이 정보와 이 정보가 지원하는 제품을 사용하기 전에, [455 페이지의 『주의사항』](#)에 있는 정보를 확인하십시오.

이 개정판은 새 개정판에서 별도로 명시하지 않는 한 CICS®용 REXX Development System, 버전 1 릴리스 1, 프로그램 번호 5655-086과 CICS용 REXX Runtime Facility, 버전 1 릴리스 1, 프로그램 번호 5655-087과 모든 후속 릴리스와 수정에 적용됩니다.

© Copyright International Business Machines Corporation 1974, 2020.

# 목차

이 PDF 정보.....	xiii
제 1 부 REXX 애플리케이션 개발.....	1
제 1 장 REXX의 기능 및 컴포넌트.....	3
제 2 장 REXX 애플리케이션 작성 및 실행.....	5
REXX 명령어의 구문.....	5
REXX 명령어의 형식.....	6
REXX 명령어의 문자 케이스.....	6
REXX 절의 유형.....	8
2바이트 문자 세트 이름을 사용하는 프로그램.....	10
프로그램에서 입력.....	10
프로그램 실행.....	11
오류 메시지 해석.....	12
대문자로의 변환 방지.....	12
프로그램에 정보 전달.....	13
프로그램 스택 또는 터미널 입력 디바이스에서 정보 가져오기.....	14
프로그램 호출 시 값 지정.....	14
대문자로 입력 변환 방지.....	15
연습: ARG 명령어 사용.....	16
인수 전달.....	16
제 3 장 변수 및 표현식을 사용하여 변수 데이터 처리.....	17
변수.....	17
표현식.....	19
산술 연산자.....	19
비교 연산자.....	22
논리(부울) 연산자.....	24
연결 연산자.....	26
연산자의 우선순위.....	26
제 4 장 프로그램 내 플로우 제어.....	29
조건부 명령어.....	29
IF...THEN...ELSE 명령어.....	29
중첩된 IF...THEN...ELSE 명령어.....	31
SELECT WHEN...OTHERWISE...END 명령어.....	34
루프 명령어.....	37
반복 루프.....	37
조건부 루프.....	40
복합 루프.....	44
인터럽트 명령어.....	46
EXIT 명령어.....	46
CALL 및 RETURN 명령어.....	47
SIGNAL 명령어.....	50
제 5 장 함수.....	53
내장 함수.....	54
서브루틴 및 함수.....	58
서브루틴 및 함수 작성.....	59

내부 또는 외부 서브루틴 또는 함수 사용 선택.....	66
정보 전달.....	66
서브루틴 또는 함수에서 정보 수신.....	72
<b>제 6 장 데이터 조작.....</b>	<b>75</b>
복합 변수 및 스템 사용.....	75
복합 변수의 개념.....	75
스템 사용.....	76
연습 - 복합 변수 및 스템 사용.....	76
데이터 구문 분석.....	77
구문 분석 명령어.....	77
단어로 구문 분석에 대한 추가 정보 .....	79
패턴으로 구문 분석.....	79
인수로서 여러 문자열 구문 분석.....	81
<b>제 7 장 프로그램에서 명령 사용.....</b>	<b>85</b>
명령에서 따옴표 사용.....	85
명령에서 변수 사용.....	85
다른 REXX 프로그램을 명령으로 호출.....	85
프로그램에서 명령 실행.....	86
명령이 호스트 환경으로 어떻게 전달됩니까?.....	86
호스트 명령 환경 변경.....	87
<b>제 8 장 프로그램에서 문제점 진단.....</b>	<b>89</b>
TRACE 명령어로 표현식 추적.....	89
TRACE 명령어로 명령 추적.....	90
REXX 특수 변수 RC 및 SIGL 사용.....	91
대화식 디버그 기능을 사용하여 추적.....	91
대화식 TRACE 출력 저장.....	93
<b>제 9 장 REXX/CICS 도움말 유틸리티 사용.....</b>	<b>95</b>
<b>제 10 장 프로그래밍 스타일 및 기술.....</b>	<b>97</b>
직접 테스트.....	97
휴식 시간.....	98
프로그램 디자인.....	100
루프를 디자인하기 위한 메소드.....	100
결론.....	100
지금까지 수행한 작업.....	100
단계별 세분화: 예제 .....	101
데이터 재고려.....	102
프로그램 수정.....	102
프로그램 수정.....	102
프로그램 추적.....	102
코딩 스타일.....	103
<b>제 2 부 REXX 구성.....</b>	<b>107</b>
<b>제 11 장 REXX 지원 구성.....</b>	<b>109</b>
RFS 파일 풀 작성.....	109
자원 정의 작성.....	109
LSRPOOL 정의 검토.....	110
CICSTART 멤버 업데이트.....	110
CICS 초기화 JCL 수정.....	111
RHS 파일 풀 형식화.....	112
설치 확인.....	112
도움말 파일 작성.....	113

REXX Db2 인터페이스 구성.....	114
제 12 장 REXX/CICS 시스템 정의 및 관리.....	115
권한 부여된 REXX/CICS 명령 및 권한 부여된 명령 옵션.....	115
시스템 프로파일 exec.....	115
권한 부여된 MVS PDS REXX 라이브러리.....	115
권한 부여된 사용자 정의.....	115
시스템 옵션 설정.....	116
REXX 파일 시스템(RFS) 파일 풀 정의.....	116
CICSTART의 PLT 항목 작성.....	116
보안 엑시트.....	116
CICSECX1.....	116
CICSECX2.....	117
제 13 장 성능 고려사항.....	119
제 14 장 보안.....	121
REXX/CICS는 다중 트랜잭션 ID를 지원합니다.....	121
REXX/CICS 파일 보안.....	121
REXX/CICS 명령 레벨 보안.....	121
REXX/CICS 권한 부여된 명령 지원.....	121
보안 정의.....	122
<b>제 3 부 CICS Transaction Server용 REXX: 참조.....</b>	<b>125</b>
제 15 장 제품 기능의 개요.....	127
제 16 장 구문 다이어그램을 읽는 방법.....	129
제 17 장 REXX 일반 개념.....	131
구조 및 일반 구문.....	131
문자.....	132
설명.....	132
토큰.....	132
내재적 세미콜론.....	136
연속.....	136
표현식 및 연산자.....	136
표현식.....	136
연산자.....	137
소괄호와 연산자 우선순위.....	140
절 및 명령어.....	141
지정 및 기호.....	142
상수 기호.....	143
단순 기호.....	143
복합 기호.....	143
스텝.....	144
외부 환경에 대한 명령.....	145
CICS에서 실행되는 REXX의 기본 구조.....	146
표준 REXX 기능의 지원.....	148
REXX 명령 환경 지원.....	149
표준 CICS 기능에 대한 지원.....	149
다른 프로그래밍 언어에 대한 인터페이스.....	150
제 18 장 키워드 명령어.....	151
ADDRESS.....	151
ARG.....	152
CALL.....	153

DO .....	155
DROP.....	159
EXIT.....	160
IF.....	161
INTERPRET.....	161
ITERATE .....	162
LEAVE .....	163
NOP.....	163
NUMERIC.....	164
OPTIONS.....	165
PARSE.....	166
PROCEDURE.....	168
PULL.....	169
PUSH.....	170
QUEUE.....	170
RETURN.....	171
SAY.....	171
SELECT.....	171
SIGNAL.....	172
TRACE.....	174
UPPER.....	177
제 19 장 함수.....	179
구문.....	179
함수와 서브루틴.....	179
내장 함수.....	181
ABBREV(약어).....	182
ABS(절대값).....	182
ADDRESS.....	182
ARG(인수).....	183
BITAND (Bit by Bit AND).....	184
BITOR (Bit by Bit OR).....	184
BITXOR(Bit by Bit Exclusive OR).....	184
B2X(2진을 16진으로).....	185
CENTER/CENTRE.....	185
COMPARE.....	185
CONDITION .....	186
COPIES .....	186
C2D(문자를 10진수로).....	187
C2X(문자를 16진수로).....	187
DATATYPE .....	188
DATE .....	189
DBCS(Double-Byte Character Set Functions).....	190
DELSTR(문자열 삭제).....	190
DELWORD(단어 삭제).....	190
DIGITS .....	191
D2C(10진수를 문자로).....	191
D2X(10진수를 16진으로).....	191
ERRORTEXT .....	192
EXTERNALS.....	192
FIND.....	192
FORM .....	193
FORMAT .....	193
FUZZ .....	194
INDEX.....	194
INSERT.....	194
JUSTIFY.....	195
LASTPOS(마지막 위치).....	195

LEFT .....	195
LENGTH.....	196
LINESIZE.....	196
MAX(최대).....	196
MIN(최소).....	196
OVERLAY .....	197
POS(위치).....	197
QUEUED .....	197
RANDOM .....	198
REVERSE .....	198
RIGHT .....	198
SIGN.....	199
SOURCELINE .....	199
SPACE .....	199
STORAGE.....	200
STRIP .....	200
SUBSTR(하위 문자열).....	200
SUBWORD .....	200
SYMBOL .....	201
TIME .....	201
TRACE.....	203
TRANSLATE .....	203
TRUNC(자르기).....	203
USERID.....	204
VALUE.....	204
VERIFY.....	205
WORD .....	205
WORDINDEX .....	206
WORDLENGTH .....	206
WORDPOS(단어 위치).....	206
WORDS .....	206
XRANGE(16진 범위).....	207
X2B(16진을 2진으로).....	207
X2C(16진을 문자로).....	207
X2D(16진을 10진으로).....	208
REXX/CICS에서 제공되는 외부 기능.....	208
STORAGE.....	208
SYSSBA.....	209
 제 20 장 구문 분석.....	 211
단어로 구문 분석하기 위한 단순 템플리트.....	211
문자열 패턴을 포함하는 템플리트.....	213
위치(숫자) 패턴을 포함하는 템플리트.....	213
변수 패턴을 사용한 구문 분석.....	216
UPPER 사용.....	217
구문 분석 명령어 요약.....	218
구문 분석 명령어 예제.....	218
고급 구문 분석 정보.....	219
여러 문자열 구문 분석.....	219
문자열과 위치 패턴 결합: 특수한 경우.....	220
DBCS 문자 구문 분석.....	220
구문 분석 단계 세부사항.....	220
 제 21 장 숫자와 산술 오퍼레이션 .....	 225
소개: 수.....	225
산술 기능의 정의.....	226
숫자.....	226
정밀도.....	226

산술 연산자.....	226
산술 오퍼레이션 규칙: 기본 연산자.....	227
산술 오퍼레이션 규칙: 추가 연산자.....	228
숫자 비교.....	229
지수 표기법.....	230
숫자 정보.....	231
정수.....	231
REXX에 의해 직접 사용되는 숫자.....	231
오류.....	232
 제 22 장 조건 및 조건 트랩.....	233
조건이 트래핑되지 않은 경우 조치.....	234
조건이 트랩되는 경우 조치.....	234
조건 정보.....	235
특수 변수.....	236
 제 23 장 REXX/CICS 텍스트 편집기.....	239
명령행 명령.....	242
ARBCHAR.....	242
ARGS.....	242
BACKWARD.....	243
BOTTOM .....	243
CANCEL.....	243
CASE.....	244
CHANGE.....	245
CMDLINE.....	245
CTLCHAR.....	246
CURLINE.....	246
DISPLAY.....	247
DOWN.....	247
EDIT.....	248
EXEC.....	249
FILE .....	250
FIND.....	250
FORWARD.....	251
GET.....	252
GETPDS.....	252
INPUT.....	253
JOIN.....	253
LEFT .....	254
LINEADD.....	254
LPREFIX.....	255
MACRO .....	255
MSGLINE.....	256
NULLS.....	256
NUMBERS.....	257
PFKEY.....	257
PFKLINE.....	258
QQUIT.....	258
QUERY.....	259
QUIT.....	260
RESERVED.....	260
RESET.....	261
RIGHT .....	261
SAVE.....	262
SORT.....	263
SPLIT.....	263
STRIP .....	264



SYNONYM.....	264
TOP .....	264
TRUNC .....	265
UP.....	265
제 24 장 REXX/CICS 파일 시스템.....	267
파일 풀, 디렉토리 및 파일.....	267
현재 디렉토리 및 경로.....	268
보안.....	269
RFS 명령 .....	269
AUTH.....	269
CKDIR.....	270
CKFILE.....	270
COPY.....	270
DELETE.....	271
DISKR.....	271
DISKW.....	272
GETDIR.....	272
MKDIR.....	273
RDIR.....	273
RENAME.....	273
REXX/CICS 파일 목록 유틸리티.....	274
호출.....	274
REXX/CICS 파일 목록 유틸리티 아래 매크로.....	275
FLST 명령.....	275
FLST 리턴 코드.....	281
FLST로부터 exec 및 트랜잭션 실행.....	281
제 25 장 REXX/CICS 목록 시스템.....	283
디렉토리 및 목록.....	283
현재 디렉토리 및 경로.....	284
보안.....	284
RLS 명령 .....	284
CKDIR.....	284
DELETE.....	285
LPULL.....	285
LPUSH.....	285
LQUEUE.....	286
MKDIR.....	286
READ.....	287
VARDROP.....	287
VARGET.....	288
VARPUT.....	288
WRITE.....	289
제 26 장 REXX/CICS 명령 정의.....	291
백그라운드.....	291
명령 정의.....	291
REXX 프로그램에 전달되는 명령 인수.....	292
어셈블러 프로그램에 전달되는 명령 인수.....	292
CICPARMS 제어 블록.....	292
비REXX 언어 인터페이스.....	293
CICGETV: REXX 변수 가져오기, 설정하기 또는 삭제를 위한 호출.....	294
제 27 장 REXX/CICS Db2 인터페이스.....	295
프로그래밍 고려사항.....	295
SQL문 임베드.....	295
결과 수신.....	297

SQL 통신 영역 사용.....	298
SQL문 사용 예제.....	298
Db2 명령 임베드.....	299
결과 수신.....	300
Db2 명령을 사용하는 예.....	301
 제 28 장 REXX/CICS 상위 레벨 클라이언트/서버 지원.....	303
상위 레벨의 고유하고 투명한 REXX 클라이언트 인터페이스.....	303
REXX 기반 애플리케이션 클라이언트 및 서버에 대한 지원.....	303
클라이언트/서버 계산에서 REXX의 값.....	303
REXX/CICS 클라이언트 exec 예제.....	304
REXX/CICS 서버 exec 예제.....	304
 제 29 장 REXX/CICS 패널 기능.....	305
패널 정의.....	306
.DEFINE verb를 사용하여 필드 제어 문자 정의.....	306
.DEFINE.....	307
.PANEL verb를 사용하여 실제 PANEL 레이아웃 정의.....	309
.PANEL.....	310
패널 생성 및 패널 입출력(I/O).....	311
PANEL RUNTIME.....	312
PANEL 변수.....	316
패널 기능 리턴 코드 정보.....	317
상태 코드 및 입력 코드.....	319
위치 코드.....	323
샘플 패널의 예제.....	323
REXX 패널 프로그램의 예제.....	325
 제 30 장 REXX/CICS 명령.....	331
ALLOC.....	331
AUTHUSER.....	332
CD.....	333
CEDA.....	334
CEMT .....	335
CLD.....	349
CONVTMAP.....	350
COPYR2S.....	351
COPYS2R.....	352
C2S.....	354
DEFCMD.....	355
DEFSCMD.....	357
DEFTRNID.....	359
DIR.....	360
EDIT.....	361
EXEC.....	361
EXECDROP.....	362
EXECIO.....	363
EXECLOAD.....	364
EXECMAP.....	366
EXPORT.....	366
FILEPOOL.....	368
FLST.....	370
FREE.....	370
GETVERS.....	371
HELP.....	371
IMPORT.....	371
LISTCMD.....	373
LISTPOOL.....	374

LISTTRNID.....	374
PATH.....	375
PSEUDO.....	376
RFS.....	377
RLS.....	379
SCRNINFO.....	382
SET.....	382
SETSYS.....	385
S2C.....	386
TERMID.....	387
WAITREAD.....	388
WAITREQ.....	388
제 31 장 오류 번호 및 메시지 .....	391
CICREXnnn 오류 메시지.....	392
제 32 장 리턴 코드.....	401
패널 기능 리턴 코드.....	401
SQL 리턴 코드.....	401
Db2 리턴 코드.....	401
RFS 및 FLST.....	402
EDITOR 및 EDIT.....	403
DIR.....	404
SET.....	404
CD.....	404
PATH.....	404
RLS.....	405
LISTCMD.....	406
CLD.....	406
DEFCMD.....	407
DEFSCMD.....	407
DEFTRNID.....	407
EXECDROP.....	408
EXECLOAD.....	408
EXECMAP.....	408
ALLOC 및 FREE.....	409
EXPORT 및 IMPORT.....	409
FILEPOOL.....	410
GETVERS.....	411
COPYR2S.....	411
COPYS2R.....	411
LISTPOOL.....	412
LISTTRNID.....	412
C2S.....	412
PSEUDO.....	412
AUTHUSER.....	412
SETSYS.....	413
S2C.....	413
TERMID.....	413
WAITREAD.....	413
WAITREQ.....	414
명령에 특정하지 않은 리턴 코드.....	414
EXEC.....	414
CEDA 및 CEMT.....	415
EXECIO.....	428
CONVTMAP.....	428
SCRNINFO.....	428
CICS.....	428

제 33 장 2바이트 문자 세트(DBCS) 지원 .....	431
DBCS: 일반 설명.....	431
DBCS 데이터 조작 및 기호 사용 설정.....	432
기호와 문자열.....	432
명령어 및 DBCS.....	433
DBCS 함수 처리.....	435
기본 제공 함수 예.....	436
DBCS 처리 함수.....	439
DBADJUST.....	440
DBBRACKET.....	440
DBCENTER.....	440
DBCJUSTIFY.....	441
DBLEFT.....	441
DBRIGHT.....	442
DBRLEFT.....	442
DBRRIGHT.....	442
DBTODBCS.....	443
DBTOSBCS.....	443
DBUNBRACKET.....	443
DBVALIDATE.....	443
DBWIDTH.....	444
제 34 장 예약된 키워드 및 특수 변수.....	445
예약 키워드.....	445
특수 변수.....	445
제 35 장 디버그 지원.....	447
프로그램 대화식 디버깅.....	447
실행 인터럽트 및 추적 제어.....	448
제 36 장 BMS(Basic Mapping Support) 예.....	449
제 37 장 참고 목록.....	453
<b>주의사항 .....</b>	<b>455</b>
<b>색인.....</b>	<b>459</b>

## 이 PDF 정보

---

이 PDF는 REXX/CICS 또는 CICS Transaction Server용 REXX를 설명합니다. 이 IBM® 프로그램 제품은 연관된 런타임 기능과 함께 REXX/CICS용 고유 REXX 기반 애플리케이션 개발, 사용자 정의, 프로토타입 작성과 프로시저 언어 환경을 제공합니다.

CICS Transaction Server용 REXX(CICS용 REXX)는 CICS/ESA용 REXX를 위한 새 이름입니다.

- CICS용 REXX Development System, 버전 1 릴리스 1은 CICS/ESA용 REXX Development System, 버전 1 릴리스 1의 새 이름입니다.
- CICS용 REXX Runtime Facility, 버전 1 릴리스 1은 CICS/ESA용 REXX Runtime Facility, 버전 1 릴리스 1의 새 이름입니다.

버전, 릴리스와 제품 번호는 변경되지 않습니다. CICS용 REXX에 대한 이 문서에서 모든 참조는 CICS와 사용하기 위해 지원되는 이 제품의 이전 버전을 포함하며 CICS/ESA용 REXX를 호출합니다.

### 이 PDF를 읽어야 하는 사용자

이 PDF는 CICS Transaction Server용 REXX 명령어와 함수를 참조해야 하는 사용자 및 구문 분석과 같이 REXX 언어 항목에 관한 세부사항을 알아야 하는 사용자를 위한 것입니다. REXX 프로그램을 쓰는 방법을 배우려는 누군가를 위한 것입니다. 사용자의 유형은 애플리케이션 프로그래머, 시스템 프로그래머, 일반 사용자, 관리자, 개발자, 테스터 및 지원 인원을 포함합니다.

### 이 PDF 이해

이 PDF는 사용자 안내서와 참고 자료 모두를 포함합니다. REXX 애플리케이션 개발 및 REXX 지원 구성 섹션은 CICS Transaction Server용 REXX에 익숙하는데 도움이 됩니다. 참조 섹션은 REXX 명령어, 함수 및 주석을 포함합니다. 명령어, 함수 및 주석은 자체 섹션에서 알파벳순으로 나열됩니다. REXX에서 프로그래밍하기 위해 알아야 하는 일반 개념에 관한 세부사항도 포함됩니다.

이 책에서 설명된 프로그래밍 언어는 REstructured eXtended eXecutor 언어(일반적으로 REXX라는)라고 합니다. 이 책은 CICS Transaction Server REXX 언어 프로세서(이후 언어 프로세서로 단축화된)가 처리하거나 REstructured eXtended eXecutor 언어를 해석하는 방법도 설명합니다.

### 이 PDF의 날짜

이 PDF가 2019년 1월 31일에 작성되었습니다.



---

## 제 1 부 REXX 애플리케이션 개발

이 절에서는 REXX 프로그램 및 해당 구문을 소개하고, REXX 프로그램 작성 및 실행과 관련된 단계를 설명하고, 일반 문제점을 방지하기 위해 이해해야 하는 개념에 대해 설명합니다.

REXX 프로그램은 REXX 해석기가 직접 해석하는 REXX 언어 명령어로 구성됩니다. 프로그램은 CICS 명령과 같은 호스트 환경이 실행하는 명령을 포함할 수 있습니다(85 페이지의 [『제 7 장 프로그램에서 명령 사용』](#) 참조).





# 제 1 장 REXX의 기능 및 컴포넌트

REXX는 다용도 프로그래밍 언어입니다. 환경을 호스팅하는 명령과 함께 사용할 수 있으며 강력한 기능을 제공하며 광범위한 수학적 기능이 있습니다.

REXX 프로그램은 CICS의 많은 태스크를 수행할 수 있습니다. 여기에는 발행하는 EXEC CICS 명령, SQL문, CEDA(자원 정의 온라인 트랜잭션) 및 CEMT(마스터 터미널 트랜잭션) 유틸리티에 대한 명령이 포함됩니다.

## REXX의 기능

REXX의 기능 중 일부는 다음과 같습니다.

### 사용 용이성

REXX 언어는 많은 명령어가 의미 있는 영어 단어이기 때문에 읽고 작성하기가 쉽습니다. REXX 명령어는 SAY, PULL, IF...THEN...ELSE..., DO...END 및 EXIT와 같은 일반 단어입니다.

### 자유 형식

REXX 형식에는 몇 가지 규칙만 있습니다. 특정 열에서 명령어를 시작할 필요가 없습니다. 행에서 공백을 건너뛰거나, 전체 행을 건너뛸 수 있습니다. 많은 행의 명령어 스패를 가지거나 한 행에 복수의 명령어가 있을 수 있습니다. 변수를 미리 정의할 필요가 없습니다. 명령어를 대문자, 소문자 또는 대소문자로 입력할 수 있습니다. [REXX 명령어의 구문을 참조하십시오.](#)

### 기본 제공 함수

REXX는 텍스트와 숫자 모두에 대해 다양한 처리, 검색 및 비교 오퍼레이션을 수행하는 기본 제공 함수를 제공합니다. 기타 기본 제공 함수는 형식화 기능과 산술 계산을 제공합니다.

### 디버깅 기능

REXX/CICS에서 실행 중인 REXX 프로그램에서 오류가 발생하면 REXX는 오류를 설명하는 메시지를 작성합니다. REXX TRACE 명령어와 대화식 디버그 기능을 사용하여 프로그램 오류를 찾을 수도 있습니다.

### 해석형 언어

REXX/CICS 제품에는 REXX/CICS 해석기가 포함되어 있습니다. REXX 프로그램이 실행되면 해석기가 직접 각 행을 처리합니다. 실행하기 전에 프로그램을 컴파일하거나 링크 편집할 필요가 없습니다.

### 광범위한 구문 분석 기능

REXX에는 문자 조작을 위한 광범위한 구문 분석 기능이 포함되어 있으므로 문자, 숫자 및 혼합 입력을 구분하는 패턴을 설정할 수 있습니다.

## REXX의 컴포넌트

REXX는 다음 컴포넌트로 구성되어 있고 프로그래머에게 강력한 도구입니다.

### 절

절은 명령어, 널 절 또는 레이블일 수 있습니다. 명령어는 다음과 같을 수 있습니다.

- 키워드 명령어
- 지정사항
- 명령(REXX/CICS 및 CICS 명령과 SQL).

언어 프로세서는 키워드 명령어와 지정사항을 처리합니다.

### 기본 제공 함수

이러한 함수는 언어 프로세서에 빌드인되어 있으며 편리한 처리 옵션을 제공합니다.

### 외부 함수

REXX/CICS는 시스템과 상호작용하여 REXX에 대한 특정 태스크를 수행하는 이러한 함수를 제공합니다.

### 데이터 스택 기능

데이터 스택은 I/O와 다른 유형의 처리에 대한 데이터를 저장할 수 있습니다

## 선행 조건

REXX/CICS는 모든 지원되는 CICS Transaction Server 릴리스에서 실행됩니다. CICS TS에 필요한 사항 외에 다른 전제조건이 없습니다.

## 런타임 기능

REXX/CICS MVS™ 런타임 기능은 REXX/CICS MVS exec에 대한 런타임 환경을 제공하므로 REXX/CICS MVS 개발 시스템이 설치되지 않은 CICS 리전에서 REXX/CICS exec을 실행할 수 있습니다.

이는 회사에 REXX 기반 개발에 특정 리전만 사용 가능하고 해당 exec을 실행하려는 CICS 리전이 여러 개인 경우에 특히 유용합니다. 런타임 기능을 사용하여 전체 REXX/CICS MVS 개발 시스템의 라이선스를 구매하지 않고도 다른 리전에서 REXX 전제조건이 있는 제품을 사용할 수 있습니다.

## 제 2 장 REXX 애플리케이션 작성 및 실행

REXX 언어의 한 가지 장점은 일반 영어와의 유사성입니다. 이러한 유사성은 다음 예와 같이 REXX 프로그램을 쉽게 읽고 작성할 수 있게 합니다.

### 간단한 프로그램의 예

출력 행을 작성하려면 REXX 명령어 SAY 다음에 원하는 텍스트를 사용하십시오.

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

그림 1. 예제: *Hello World* 프로그램

이 프로그램은 주석 행으로 시작하여 이를 REXX 프로그램으로 식별합니다. 주석은 /\*로 시작하고 \*/로 끝납니다.

프로그램을 실행할 때 SAY 명령어는 다음 출력을 터미널 디바이스로 보냅니다.

```
Hello world!
```

### 더 긴 프로그램의 예

더 긴 프로그램이어도 명령은 일반 영어와 유사하며 이해하기 쉽습니다. 이 예에서는 두 개의 숫자를 추가하는 ADDTWO 프로그램을 호출합니다.

1. CICS 터미널에서 화면을 지우고 다음 명령을 입력하십시오.

```
REXX addtwo
```

2. 두 개의 수를 입력하십시오.

ADDTWO 프로그램이 여기 있습니다. 프로그램 코드의 주석은 입력된 첫 번째 숫자는 42이고 두 번째 숫자는 21이라고 가정합니다.

```
/****** 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'.'
```

그림 2. 예: *ADDTWO* 프로그램

프로그램 예를 실행하면 첫 번째 PULL 명령어는 변수 *number1*에 값 42를 지정합니다. 두 번째 PULL 명령어는 변수 *number2*에 값 21을 지정합니다. 다음 행에는 지정이 포함됩니다. 언어 프로세서는 *number1*과 *number2*의 값을 더하고 결과인 63을 *sum*에 지정합니다. 마지막으로 SAY 명령어는 출력 행을 표시합니다.

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

## REXX 명령어의 구문

일부 프로그래밍 언어에는 각 행에 문자를 입력하는 방법 및 위치에 대한 엄격한 규칙이 있습니다. 예를 들어, 아셈블러 명령문은 특정 열에서 시작해야 합니다. 반면에 REXX에는 간단한 구문 규칙이 있습니다. 대문자, 소문자 또는 대소문자를 혼합하여 사용할 수 있습니다. REXX에는 입력할 수 있는 열에 대한 제한사항이 없습니다.

명령어는 모든 행의 모든 열에서 시작할 수 있습니다. 다음은 모두 유효한 명령어입니다.

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

다음 명령어가 터미널 출력 디바이스로 전송됩니다.

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

## REXX 명령어의 형식

REXX 언어의 형식은 자유롭습니다. 단어 사이에 추가 공백을 삽입할 수 있다는 의미입니다.

예를 들어, 다음은 모두 같은 의미입니다.

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

오류를 발생시키지 않고 프로그램 전체에 빈 행을 삽입할 수도 있습니다.

## REXX 명령어의 문자 케이스

REXX 명령어를 소문자, 대문자 또는 대소문자로 입력할 수 있습니다. 언어 프로세서는 영문자를 작은따옴표 또는 큰따옴표로 묶지 않는 한 대문자로 변환합니다.

예를 들어, SAY, Say 및 say는 모두 같은 뜻입니다.

### 명령어에서 따옴표 사용

일치하는 따옴표 안에 있는 일련의 문자는 리터럴 문자열입니다. 다음 예제는 리터럴 문자열을 포함합니다.

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

리터럴 문자열을 서로 다른 두 가지 유형의 따옴표로 묶지 마십시오. 예를 들어, 다음은 올바르지 않습니다.

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

SAY 명령어에서 리터럴 문자열 주위에 따옴표를 생략하면 언어 프로세서는 일반적으로 명령문을 대문자로 변환합니다. 예를 들어, 다음과 같습니다.

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

결과는 다음과 같습니다.

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

(이는 단어가 이미 값을 지정했었던 변수의 이름이 아니라고 가정합니다. REXX에서 변수의 기본값은 대문자로 된 고유 이름입니다.)

문자열에 아포스트로피가 포함된 경우 리터럴 문자열을 큰 따옴표로 묶을 수 있습니다.

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

작은따옴표 한 쌍이 하나로 처리되므로 아포스트로피 대신 작은따옴표 두 개를 사용할 수도 있습니다.

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

어느 쪽이든, 결과는 동일합니다.

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

## 명령어 종료

행은 일반적으로 세미콜론(;)을 포함하거나 쉼표(,)로 끝나는 경우를 제외하고 하나의 명령어를 포함합니다.

줄의 끝 또는 세미콜론은 명령의 끝을 표시합니다. 한 명령어를 한 행에 넣으면 행의 끝이 명령어의 끝을 나타냅니다. 한 행에 여러 개의 명령어를 넣을 경우 인접한 명령어를 세미콜론으로 구분해야 합니다.

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

이 예는 세 개의 행이 될 수 있습니다.

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

## 명령어 계속

쉼표는 연속 문자입니다. 명령어가 다음 행으로 계속 진행됨을 나타냅니다. 이러한 방식으로 사용될 때 쉼표는 행이 연결될 때 공백을 추가합니다. 다음 행에서 리터럴 문자열이 계속될 때 쉼표 연속 문자가 작동하는 방식은 다음과 같습니다.

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

첫 번째 행의 끝에 있는 쉼표는 두 행이 출력을 위해 연결될 때 공백(extended 및 REXX)을 추가합니다. 단일 행 결과:

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

다음 두 명령어는 동일하며 결과는 동일합니다.

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

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

두 개의 개별 문자열 사이의 공백은 유지됩니다.

```
문자열입니다.
```

## 공백을 추가하지 않고 리터럴 문자열 계속

명령을 두 번째 이상의 행으로 계속 진행해야 하지만 REXX에서 행에 공백을 추가하지 않으려면 연결 피연산자(두 개의 단일 OR 막대, ||)를 사용하십시오.

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

이 예제는 결과적으로 한 행이고 단어 "broken"에 공백이 없이 표시됩니다.

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

또한 다음 두 명령어는 동일하며 결과가 동일합니다.

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

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

이 예의 결과는 다음과 같습니다.

```
This isa string.
```

두 예제에서 연결 연산자는 두 문자열 사이의 공백을 삭제합니다.

다음 예제는 REXX의 자유로워 형식을 보여줍니다.

```
/****** 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.'
```

그림 3. 자유 형식의 예

이 예제를 실행하면 6행의 동일한 출력이 생성되며 그 다음에 들여쓰기된 한 행이 표시됩니다.

```
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.
```

한 행의 어느 위치에서나 명령어를 시작할 수 있고, 빈 행을 삽입할 수 있으며, 명령어에서 단어 사이에 추가 공백을 삽입할 수 있습니다. 언어 프로세서는 빈 행을 무시하고 1보다 큰 공백은 무시합니다. 이러한 형식의 유연성을 통해 빈 행과 공백을 삽입하여 프로그램을 보다 쉽게 읽을 수 있습니다.

공백과 간격은 구문 분석 중에만 중요합니다. [77 페이지의 『데이터 구문 분석』](#)의 내용을 참조하십시오.

## REXX 절의 유형

REXX 절은 명령어, 널 절 및 레이블일 수 있습니다. 명령어는 키워드 명령어, 지정 또는 명령일 수 있습니다.

다음 예제는 이런 유형의 절이 있는 프로그램을 보여줍니다. 각 유형의 절에 대한 설명이 예제를 따릅니다.

```
/* 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
```

### 키워드 명령어

키워드 명령어는 언어 프로세서가 수행할 작업을 지시합니다. 언어 프로세서가 수행할 작업을 식별하는 REXX 키워드로 시작합니다. 예를 들어, DO는 명령어를 그룹화하여 반복적으로 실행할 수 있으며 IF는 조건이 충족되는지 테스트합니다. SAY는 현재 터미널 출력 디바이스에 기록합니다.

IF, THEN 및 ELSE는 하나의 명령어로 함께 작동하는 3개의 키워드입니다. 각 키워드는 명령의 서브세트인 절을 형성합니다. IF 키워드 뒤에 오는 표현식이 true이면, THEN 키워드 뒤에 오는 명령어가 처리됩니다. 그렇지 않으면, ELSE 키워드 뒤에 오는 명령어가 처리됩니다. (THEN과 동일한 행에 ELSE 절을 넣는 경우 ELSE 앞에 세미콜론이 필요합니다.) THEN 또는 ELSE 다음에 둘 이상의 명령어를 넣으려면 명령어 그룹 앞에 DO를 사용하고 그 뒤에 END를 사용하십시오. IF 명령어에 대한 추가 정보는 [29 페이지의 『조건부 명령어』](#)에 표시되어 있습니다.

EXIT 키워드는 언어 프로세서가 프로그램을 종료하도록 지시합니다. 앞의 예제에서 EXIT의 사용이 필요합니다. 그렇지 않으면 언어 프로세서가 레이블 sub: 다음의 서브루틴에서 코드를 실행하기 때문입니다. 일부 프로그램

(예: 서브루틴이 없는 프로그램)에서는 EXIT가 필요하지 않지만 포함하는 것이 좋은 프로그래밍 관행입니다. EXIT에 대한 추가 정보는 [46 페이지의 『EXIT 명령어』](#)에 표시되어 있습니다.

## 지정

지정은 값을 변수에 제공하거나 변수의 현재 값을 변경합니다. 간단한 지정 명령어는 다음과 같습니다.

```
number = 4
```

위의 프로그램에서 간단한 지정 명령어는 store\_a=0입니다. 지정의 왼쪽(등호 앞)에는 오른쪽(등호 뒤)에서 값을 수신할 변수의 이름이 포함됩니다. 오른쪽은 실제 값(예: 4) 또는 표현식일 수 있습니다. 표현식은 산술 연산식과 같이 평가되어야 하는 항목입니다. 표현식은 숫자, 변수 또는 둘 다를 포함할 수 있습니다.

```
number = 4 + 4  
number = number + 4
```

첫 번째 예에서 number의 값은 8입니다. 두 번째 예가 프로그램의 첫 번째 예를 직접 따르는 경우 number의 값은 12가 될 수 있습니다. 표현식에 대한 추가 정보는 [19 페이지의 『표현식』](#)의 내용을 참조하십시오.

## 레이블

sub:와 같은 레이블은 기호 이름 뒤에 콜론이 옵니다. 레이블에는 1바이트 문자 또는 2바이트 문자 또는 1바이트 문자 및 2바이트 문자 조합이 포함될 수 있습니다. (2바이트 문자는 OPTIONS ETMODE가 프로그램의 첫 번째 명령어인 경우에만 유효합니다.) 레이블은 프로그램의 일부를 식별하며 일반적으로 서브루틴 및 함수에서 SIGNAL 명령어와 함께 사용됩니다. (제어를 주 프로그램으로 다시 전송하려면 서브루틴 끝에 RETURN 명령어를 포함해야 합니다.) 레이블 사용에 대한 추가 정보는 [58 페이지의 『서브루틴 및 함수』](#) 및 [50 페이지의 『SIGNAL 명령어』](#)의 내용을 참조하십시오.

## 널 절

널 절은 공백, 주석 또는 둘 다로 구성됩니다. 언어 프로세서는 널 절을 무시하지만 프로그램을 쉽게 읽을 수 있게 합니다.

## 설명

주석은 /\*로 시작하고 \*/로 끝납니다. 주석은 하나 이상의 행 또는 행의 일부에 있을 수 있습니다. REXX 명령어를 읽는 사용자에게 명확하지 않을 수 있는 정보를 주석으로 넣을 수 있습니다. 프로그램 시작 부분의 주석은 프로그램의 전반적인 목적을 설명하고 특별한 고려사항을 나열할 수 있습니다. 개별 명령어 옆에 있는 주석은 그 용도를 명확하게 할 수 있습니다.

**참고:** REXX/CICS에서는 REXX 프로그램이 주석으로 시작되지 않아도 됩니다. 그러나 이식성의 이유로 REXX라는 단어가 포함된 주석으로 각 REXX 프로그램을 시작할 수 있습니다. 모든 언어 프로세서에이 프로그램 ID가 필요한 것은 아닙니다. 그러나 MVS TSO 및 CICS에서 동일한 exec를 실행하려면, /\* REXX \*/ 프로그램 ID를 포함하여 TSO 요구사항을 충족해야 합니다.

## 빈 행

빈 행은 명령어 그룹을 분리하고 가독성을 향상시킵니다. 프로그램이 읽기 쉬울수록 이해하고 유지보수하기가 더 용이합니다.

## 명령

명령은 표현식으로만 구성되는 절입니다. 명령은 처리를 위해 이전에 정의된 환경으로 전송됩니다. (평가할 수 없는 표현식 파트는 따옴표로 묶어야 합니다.) 이전 프로그램 예제에는 명령이 포함되지 않았습니다. 다음 예제는 명령을 ADDRESS 명령어에 포함시킵니다.

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

ADDRESS는 키워드 명령어입니다. ADDRESS 명령어에서 환경과 명령을 지정하면 지정한 환경으로 단일 명령이 전송됩니다. 이 경우 환경은 REXXCICS입니다. 명령은 환경을 따르는 표현식입니다.

'DIR'

DIR 명령은 현재 파일 시스템 디렉토리의 파일을 나열합니다. 호스트 명령 환경 변경에 대한 세부사항은 87 페이지의 『호스트 명령 환경 변경』의 내용을 참조하십시오. 명령 발행에 대한 자세한 정보는 85 페이지의 『제 7 장 프로그램에서 명령 사용』의 내용을 참조하십시오.

## 2바이트 문자 세트 이름을 사용하는 프로그램

리터럴 문자열, 기호 및 주석에 대해 REXX 프로그램에서 2바이트 문자 세트(DBCS) 이름을 사용할 수 있습니다. 이러한 문자열은 1바이트, 2바이트 또는 둘 다의 조합일 수 있습니다. DBCS 이름을 사용하려면 OPTIONS ETMODE는 프로그램에서 첫 번째 명령어이어야 합니다. 이는 언어 프로세서가 DBCS 문자가 포함된 문자열의 유효성을 검사하도록 지정합니다.

DBCS 문자를 SO(shift-out) 및 SI(shift-in) 구분 기호 안에 묶어야 합니다. (SO 문자는 X'0E'이고 SI 문자는 X'0F'입니다). SO 및 SI 문자는 인쇄할 수 없습니다. 다음 예에서 미만(<) 및 초과(>) 기호는 각각 SO(shift-out) 및 SI(shift-in)를 표시합니다. 예를 들어, <S.Y.M.D> 및 <D.B.C.S.R.T.N>은 다음 예에서 DBCS 기호를 표시합니다.

### 예

다음은 DBCS 변수 이름과 DBCS 서브루틴 레이블을 사용하는 프로그램의 예입니다.

```
/* 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
```

## 프로그램에서 입력

### 이 태스크 정보

다음 프로그램을 입력하면, REXX 파일 시스템(RFS)에 상주하는 파일을 위한 REXX/CICS 편집기를 사용하십시오. 파일을 MVS 파티션된 데이터 세트(PDS 멤버)에 저장할 수 있는 편집기를 사용할 수도 있습니다. 이 정보는 REXX/CICS 편집기를 사용한다고 가정합니다.

프로그램 이름은 HELLO EXEC입니다(우선, 파일 유형이 exec이라고 가정함).

1. CESN을 입력하고 요청 시 사용자 ID 및 비밀번호를 제공하여 REXX for CICS 터미널에 사인온하십시오.
2. 화면을 지우십시오.
3. 입력:

```
edit hello.exec
```

4. /\* REXX HELLO EXEC \*/으로 시작하여 11 페이지의 그림 4에 표시된 대로 정확하게 프로그램에 입력하십시오. 그런 다음 EDIT 명령을 사용하여 저장하십시오.

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

이제 프로그램은 실행될 준비가 됩니다.



```

/* REXX HELLO EXEC */

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

```

그림 4. HELLO EXEC

## 프로그램 실행

### 이 태스크 정보

exec을 실행하기 전에 화면을 지우십시오. 파일 유형이 EXEC인 프로그램을 실행하려면 REXX를 입력한 다음 파일 이름을 입력하십시오. 이 경우 명령행에 rexx hello를 입력하고 Enter를 누르십시오. 시작해 보십시오!

이름을 Sam으로 가정합니다. sam을 입력하고 Enter를 누르십시오. Hello SAM이 표시됩니다.

```

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

```

다음과 같은 상황이 발생합니다.

1. SAY 명령어는 다음을 표시합니다. Hello! What is your name?
2. PULL 명령어는 프로그램을 일시정지하고 응답을 대기합니다.
3. 명령행에 sam을 입력한 다음 Enter를 누릅니다.
4. PULL 명령어는 SAM이라는 단어를 who라는 변수(컴퓨터의 저장영역 위치)에 넣습니다.
5. IF 명령문은 who가 없음과 동일한지 묻습니다.

```
who = ""
```

이는 다음을 의미합니다. "who에 저장된 값이 없음과 동일합니까?" 찾기 위해 REXX는 저장된 이름을 변수 이름으로 대체합니다. 따라서 이제 질문은 다음과 같습니다. SAM이 없음과 동일합니까?

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

6. True가 아닙니다. then 뒤에 명령어가 처리되지 않습니다. 대신 REXX는 else 뒤 명령어를 처리합니다.
7. SAY 명령어는 "Hello" who를 표시합니다. 다음으로 평가됨

```
Hello SAM
```

이제 응답을 먼저 입력하지 않고 Enter를 누르면 다음과 같은 결과가 발생합니다.

```

hello
Hello! What is your name?

Hello stranger!

```

그런 다음 다시, 사용자 이름을 입력해야 함을 이해하지 못했을 수 있습니다. (프로그램이 사용자 파트를 보다 명확하게 작성해야 합니다.) 어쨌든 이름을 입력하는 대신 Enter 키를 누르는 경우:

1. PULL 명령어는 who라는 컴퓨터 저장영역의 위치에 "(없음)"을 넣습니다.
2. 다시, IF 명령어는 변수

```
who = ""
```

의미를 테스트합니다. who의 값은 없음과 동일합니까? who의 값이 대체될 때 다름으로 스캔합니다.

```
"" = ""
```

그리고 이번에는 true입니다.

3. 따라서 then 뒤의 명령어가 처리되고 else 뒤의 명령어는 처리되지 않습니다.

## 오류 메시지 해석

오류가 포함된 프로그램을 실행하면 오류 메시지에 종종 오류가 발생한 행이 포함되며 오류에 대한 설명이 제공 됩니다. 오류 메시지는 구문 오류와 계산 오류로 인해 발생할 수 있습니다.

### 예

다음 프로그램에는 구문 오류가 있습니다.

```
/****** 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
```

그림 5. 구문 오류가 있는 프로그램의 예

프로그램이 실행되면 언어 프로세서는 다음과 같은 출력 행을 보냅니다.

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

이 프로그램은 언어 프로세서가 주석의 끝에 \*/가 누락된 오류를 감지할 때까지 실행됩니다. 이 행에는 구문 오류가 있으므로 PULL 명령어는 데이터 스택 또는 터미널의 데이터를 사용하지 않습니다. 프로그램이 종료되고 언어 프로세서가 오류 메시지를 보냅니다.

첫 번째 오류 메시지는 언어 프로세서가 오류를 발견한 명령문의 행 번호로 시작합니다. 세 개의 더하기(+++)와 명령문의 콘텐츠는 다음과 같습니다.

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

두 번째 오류 메시지는 메시지 번호로 시작합니다. 프로그램 이름, 언어 프로세서가 오류를 발견한 행 및 오류에 대한 설명이 포함된 메시지가 다음에 옵니다.

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

이 프로그램에서 구문 오류를 수정하려면 7행의 주석 끝에 \*/를 추가하십시오.

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

## 대문자로의 변환 방지

언어 프로세서는 일반적으로 영문자를 처리하기 전에 대문자로 변환합니다. 대문자로의 변환을 방지할 수 있습니다.

## 프로그램의 문자

프로그램에서 영문자가 대문자로 변환되는 것을 방지하려면 문자를 작은따옴표 또는 큰따옴표로 묶으십시오. 언어 프로세서는 따옴표 안에 있는지 여부에 관계없이 숫자와 특수 문자를 변경하지 않습니다. 영문자, 숫자 및 특수 문자가 혼합된 문구와 함께 SAY 명령어를 사용하는 경우 언어 프로세서는 영문자만 변경합니다.

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

결과는 다음과 같습니다.

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

이 예에서는 다른 값이 지정된 변수의 이름이 단어가 아니라고 가정합니다.

따옴표는 프로그램의 정보가 입력된 대로 정확하게 처리되도록 합니다. 이는 다음 상황에 중요합니다.

- 출력은 소문자이거나 대문자와 소문자를 혼합해야 합니다.
- 명령이 올바르게 처리되는지 확인하기 위해. 예를 들어, 프로그램의 변수 이름이 명령 이름과 동일하면 명령이 발행될 때 프로그램이 오류로 종료될 수 있습니다. 명령과 동일한 변수 이름을 사용하지 않고 모든 명령을 따옴표로 묶는 것이 좋은 프로그래밍 방법입니다.

## 프로그램의 문자 입력

언어 프로세서는 입력을 읽거나 다른 프로그램의 입력을 전달할 때 처리하기 전에 영문자를 대문자로 변경합니다. 대문자로의 변환을 방지하려면 PARSE 명령어를 사용하십시오.

예를 들어, 다음 프로그램은 터미널에서 입력을 읽고 이 정보를 터미널 출력 디바이스로 보냅니다.

```
/****** 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
```

입력이 tyrannosaurus인 경우 언어 프로세서는 다음 출력을 생성합니다.

```
TYRANNOSAURUS
```

언어 프로세서가 표시된 대로 정확하게 입력을 읽게 하려면 PULL 명령어 대신 PARSE PULL 명령어를 사용하십시오.

```
PARSE PULL animal
```

입력이 TyRann0sauRus인 경우 출력은 다음과 같습니다.

```
TyRann0sauRus
```

## 연습 - 예제 프로그램 실행 및 수정

이전 예제를 작성하고 실행할 수 있습니다. 이제 PULL 명령어를 PARSE PULL 명령어로 변경하고 차이점을 확인하십시오.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## 프로그램에 정보 전달

프로그램이 실행될 때 여러 가지 방법으로 정보를 프로그램에 전달할 수 있습니다.

## 이 태스크 정보

- PULL을 사용하여 프로그램 스택 또는 터미널 입력 디바이스의 정보를 가져와서 프로그램에 전달할 수 있습니다.
- 프로그램 호출 시 입력을 지정할 수 있습니다.

## 프로그램 스택 또는 터미널 입력 디바이스에서 정보 가져오기

PULL 명령어는 프로그램이 입력을 수신하는 한 가지 방법입니다.

## 이 태스크 정보

PULL 명령어는 입력 행을 분리하여 터미널에서 한 번에 두 개 이상의 값을 추출할 수 있습니다.

## 예

다음 ADDTWO 프로그램은 PULL을 사용하여 두 개의 입력 번호를 수신합니다.

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

그림 6. PULL을 사용하는 프로그램의 예

예제의 다음 변형은 입력 행을 분리하여 PULL 명령어를 사용하여 터미널에서 한 번에 두 개 이상의 값을 추출하는 방법을 보여줍니다.

```
/****** 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'.'
```

그림 7. PULL을 사용하는 예의 변형

프로그램을 호출하려면 CICS 터미널에서 다음 명령을 실행하십시오.

```
REXX addtwo 42 21
```

PULL 명령어는 터미널에서 숫자 42 및 21을 추출합니다.

## 프로그램 호출 시 값 지정

프로그램이 입력을 수신하는 또 다른 방법은 프로그램을 호출할 때 지정한 값을 통하는 것입니다.

예를 들어, 두 숫자 42 및 21을 ADD라는 프로그램에 전달하려면, CICS 명령을 사용할 수 있습니다.

```
REXX add 42 21
```

ADD 프로그램은 다음 예제와 같이 ARG 명령어를 사용하여 입력을 변수에 지정합니다.

```
/****** 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'.'
```

그림 8. ARG를 사용하는 프로그램의 예

ARG는 첫 번째 숫자 42를 number1에 지정하고 두 번째 숫자 21을 number2에 , 지정합니다.

값의 숫자가 ARG 또는 PULL 이후 변수 이름의 수보다 적거나 많으면 다음 절에서 설명된 대로 오류가 발생할 수 있습니다.

### 너무 적은 값 지정

PULL 또는 ARG 뒤에 변수 수보다 적은 수의 값을 지정하면 추가 변수가 널 문자열로 설정됩니다. 다음은 프로그램에 하나의 숫자만 전달하는 예입니다.

```
REXX add 42
```

언어 프로세서는 값 42를 ARG 다음에 오는 첫 번째 변수인 number1에 지정합니다. 널 문자열을 두 번째 변수인 number2에 지정합니다. 이 상황에서 두 개의 변수를 추가하려고 시도하면 프로그램은 오류로 종료됩니다. 다른 상황에서는 프로그램이 오류로 종료되지 않습니다.

### 너무 많은 값 지정

PULL 또는 ARG 다음에 나오는 변수 수보다 많은 값을 지정하면 마지막 변수는 나머지 값을 가져옵니다. 예를 들면, 3개의 숫자를 프로그램 ADD에 전달합니다.

```
REXX add 42 21 10
```

언어 프로세서는 값 42를 ARG 다음에 오는 첫 번째 변수인 number1에 지정합니다. 값 21 10을 number2 두 번째 변수에 지정합니다. 이 상황에서 두 개의 변수를 추가하려고 시도하면 프로그램은 오류로 종료됩니다. 다른 상황에서는 프로그램이 오류로 종료되지 않습니다.

마지막 변수가 나머지 값을 가져오지 못하게 하려면 PULL 또는 ARG 명령어 끝에 마침표(.)를 사용하십시오.

```
ARG number1 number2 .
```

마침표는 원하지 않는 추가 정보를 수집하기 위한 더미 변수 역할을 합니다. 이 경우 number1은 42를 수신하고, number2는 21을 수신하며, 마침표는 10이 버려지도록 보장합니다. 추가 정보가 없으면 마침표는 무시됩니다. 다음과 같이 PULL 또는 ARG 명령어 내 플레이스홀더로서 마침표를 사용할 수도 있습니다.

```
ARG . number1 number2
```

이 경우 첫 번째 값, 42는 버려지고 number1 및 number2는 다음 두 개의 값, 21 및 10을 가져옵니다.

## 대문자로 입력 변환 방지

PULL 명령어와 같이 ARG 명령어는 영문자를 대문자로 변경합니다. 대문자로의 변환을 방지하려면 PARSE ARG를 사용하십시오.

### 예

이 예는 PARSE ARG를 사용하여 대문자로의 변환을 방지하는 방법을 보여줍니다.

```
/****** 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.'
```

그림 9. PARSE ARG를 사용하는 프로그램의 예

## 연습: ARG 명령어 사용

### 이 태스크 정보

왼쪽 열은 프로그램에 전송된 입력 값을 표시합니다. 오른쪽 열은 입력을 수신하는 프로그램 내 ARG 명령어입니다. 각 변수가 수신하는 값은 무엇입니까?

	입력	입력을 수신하는 변수
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 .

### ANSWERS

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

## 인수 전달

프로그램에 전달된 값은 일반적으로 호출된 인수입니다. 인수는 하나의 단어 또는 단어의 문자열로 구성될 수 있습니다. 공백은 인수 내의 단어를 서로 분리합니다. 전달된 인수의 개수는 프로그램 호출 방법에 따라 다릅니다.

### 이 태스크 정보

CALL 명령어 또는 REXX 함수 호출을 사용하여 REXX 프로그램을 호출할 때 해당 프로그램에 최대 20개의 인수를 전달할 수 있습니다. 각 인수를 쉼표로 다음 인수와 구분하십시오.

함수 및 서브루틴에 대한 자세한 정보는 58 페이지의 『서브루틴 및 함수』의 내용을 참조하십시오. 인수에 대한 자세한 정보는 81 페이지의 『인수로서 여러 문자열 구문 분석』의 내용을 참조하십시오.

## 제 3 장 변수 및 표현식을 사용하여 변수 데이터 처리

컴퓨터 프로그래밍의 가장 강력한 측면 중 하나는 변수 데이터를 처리하여 결과를 얻을 수 있는 기능입니다.

### 이 태스크 정보

프로세스의 복잡도와 관계없이 데이터를 알 수 없거나 다양할 경우 데이터의 기호를 대체합니다. 해당 값이 다를 수 있는 기호를 변수라고 합니다. 해결되도록 계산되어야 하는 기호 또는 숫자 그룹을 표현식이라고 합니다.

### 변수

변수는 값을 표시하는 문자 또는 문자 그룹입니다. 변수에는 1바이트 문자 또는 2바이트 문자, 또는 둘 다 포함될 수 있습니다.

2바이트 문자는 OPTIONS ETMODE가 프로그램의 첫 번째 명령어인 경우에만 유효합니다. 다음 변수 *big*은 값 100만 또는 1,000,000을 표시합니다.

```
big = 1000000
```

변수는 다른 시간에 다른 값을 참조할 수 있습니다. *big*에 다른 값을 지정하면, 다시 변경될 때까지 재지정 값을 가져옵니다.

```
big = 999999999
```

변수는 프로그램을 작성할 때 알 수 없는 값을 나타낼 수 있습니다. 다음 예제에서는 사용자 이름을 알 수 없으므로 변수 *who*로 표시됩니다.

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

### 변수 이름

값을 표시하는 파트인 변수 이름은 항상 지정 명령문의 왼쪽에 있으며 값 자체는 오른쪽에 있습니다.

다음 예제에서 변수 이름은 *variable1*입니다.

```
variable1 = 5  
SAY variable1
```

이전 지정 명령문의 결과로 언어 프로세서는 *variable1*에 값 5를 지정하고 SAY는 다음을 생성합니다.

```
5
```

변수 이름은 다음 문자로 구성될 수 있습니다.

#### A - Z

알파벳 대문자

#### a - z

알파벳 소문자

#### 0 - 9

숫자

#### ? ! . \_

특수 문자

#### X'41' - X'FE'

2바이트 문자 세트(DBCS) 특성

**참고:** 2바이트 문자는 OPTIONS ETMODE가 프로그램의 첫 번째 명령어인 경우에만 유효합니다.

다음 제한사항이 변수 이름에 적용됩니다.

- 첫 번째 문자는 0 - 9 또는 마침표(.)일 수 없음
  - 변수 이름은 250바이트를 초과할 수 없습니다. DBCS 문자를 포함하는 이름의 경우 각 DBCS 문자를 2바이트로 계산하고 SO(shift-out) 및 SI(shift-in)를 각각 1 바이트로 계산합니다.
  - SO(X'0E ') 및 SI(X'0F')는 DBCS 이름 내에서 DBCS 문자를 구분해야 합니다. 또한,
    - SO와 SI는 연속될 수 없습니다.
    - SO/SI의 중첩은 허용되지 않습니다.
    - DBCS 이름에는 DBCS 공백(X'4040')이 포함될 수 없습니다.
  - 변수 이름은 RC, SIGL 또는 RESULT(REXX 특수 변수)가 아니어야 합니다. [특수 변수](#)의 내용을 참조하십시오.
- 다음 이름은 허용 가능한 변수 이름의 예입니다.

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

또한 OPTIONS ETMODE이 프로그램의 첫 번째 명령어인 경우 다음은 유효한 DBCS 변수 이름입니다. 여기서, <는 shift-out을 표시하고, >는 shift-in을 표시하며, X, Y 및 Z는 DBCS 문자를 표시하고 소문자 문자 및 숫자는 자체를 나타냅니다.

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

## 변수값

변수 이름이 표시하는 값인 변수의 값은 다음과 같이 분류될 수 있습니다.

- 상수는 다음과 같이 표시되는 숫자입니다.

```
정수(12)
10진수(12.5)
부동 소수점 숫자(1.25E2)
부호있는 숫자(-12)
문자열 상수(' 12')
```

- 문자열은 다음과 같이 따옴표 안에 있거나 없을 수 있는 하나 이상의 단어입니다.

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

- 다른 변수의 값은 다음과 같습니다.

```
variable1 = variable2
```

앞의 예제에서 *variable1*은 *variable2*의 값으로 변경되지만 *variable2*는 동일하게 유지됩니다.

- 표현식은 다음과 같이 계산되어야 합니다.

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

변수에 값이 지정되기 전에 해당 값은 대문자로 변환된 자체 이름 값입니다. 예를 들어, 변수 *new*에 값이 지정되지 않으면

```
SAY new
```

는 다음과 같이 됩니다.

```
NEW
```

## 연습: 유효한 변수 이름 식별

다음 중 유효한 REXX 변수 이름은 무엇입니까?



1. 8eight
2. \$25.00
3. MixedCase
4. nine\_to\_five
5. 결과

#### ANSWERS

1. 첫 번째 문자가 숫자이므로 올바르지 않습니다.
2. 첫 번째 문자가 통화 기호(\$)이므로 올바르지 않습니다.
3. 유효함
4. 유효함
5. 유효하지만 서브루틴에서 결과를 수신할 경우에만 사용해야 하는 특수 변수 이름입니다.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## 표현식

표현식은 평가해야 하는 것으로 숫자, 변수 또는 문자열과 0개 이상의 연산자로 구성됩니다. 연산자는 숫자, 변수 및 문자열에 대해 수행할 평가 유형을 판별합니다. 연산자에는 산술, 비교, 논리 및 연결의 네 가지 유형이 있습니다.

## 산술 연산자

산술 연산자는 유효한 숫자 상수 또는 유효한 숫자 상수를 표시하는 변수에서 작동합니다.

### 숫자 상수의 유형

#### 12

정수는 십진 소수점 또는 쉼표가 없습니다. NUMERIC DIGITS 명령어를 사용하여 이 기본값을 대체하지 않는 경우 정수를 사용한 산술 연산의 결과에는 최대 9자리가 포함될 수 있습니다. NUMERIC DIGITS 명령어에 대한 정보는 [NUMERIC](#)의 내용을 참조하십시오. 정수의 예는 다음과 같습니다.

```
123456789 0 91221 999
```

#### 12.5

**10진수**는 십진 소수점을 포함합니다. 소수점 이하의 산술 연산 결과는 소수점 전과 후의 총 최대 9자리수 (NUMERIC DIGITS 기본값)로 제한됩니다. 10진수의 예는 다음과 같습니다.

```
123456.789 0.888888888
```

#### 1.25E2

지수 표시법의 **부동 소수점 숫자**는 과학적 표기법이라고 합니다. E 다음의 숫자는 소수점이 이동하는 위치 수를 나타냅니다. 따라서 1.25E2(1.25E+2라고도 함)는 소수점을 오른쪽 두 자리로 이동하고 결과는 125입니다. E 다음에 빼기(-)가 있으면 소수점이 왼쪽으로 이동합니다. 예를 들어, 1.25E-2는 .0125입니다.

매우 크거나 매우 작은 수를 나타내려면 부동 소수점 숫자를 사용할 수 있습니다. 지수 표기법(부동 소수점 숫자)에 대한 자세한 정보는 [지수 표기법](#)의 내용을 참조하십시오.

#### -12

숫자 옆에 빼기(-)가 있는 **부호 있는** 숫자는 음수 값을 표시합니다. 숫자 옆에 더하기(+)가 있는 부호 있는 숫자는 양수 값을 표시합니다. 숫자에 부호가 없으면 양수 값인 것처럼 처리됩니다.

## 산술 연산자

### 연산자 의미

**+**  
더하기

**-**  
빼기

**\***  
곱하기

**/**  
나누기

**%**  
나머지 없이 정수를 나누고 리턴

**//**  
나눈 후 나머지만 리턴

**\*\***  
숫자를 정수 거듭제곱으로 올림

**- number**  
(접두부 -) 뺄셈 0 - number와 동일

**+number**  
(접두부 +) 덧셈 0 + number와 동일

숫자 상수 및 산술 연산자를 사용하여 다음과 같은 산술 표현식을 작성할 수 있습니다.

```
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 */
```

## 나누기

세 개의 연산자는 나눗셈을 나타냅니다. 각 연산자는 나누기 표현식의 결과를 다른 방식으로 계산합니다.

**/**  
응답을 10진수로 나누고 표현합니다. 예를 들어, 다음과 같습니다.

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

**%**  
응답을 정수로 나누고 표현합니다. 나머지는 무시됩니다. 예를 들어, 다음과 같습니다.

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

**//**  
응답을 나머지로만 나누고 표현합니다. 예를 들어, 다음과 같습니다.

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

## 평가 순서

산술 표현식에 둘 이상의 연산자가 있는 경우 숫자와 연산자의 순서가 중요할 수 있습니다. 예를 들어 다음 표현식에서 언어 프로세서는 어떤 작업을 처음 수행합니까?

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

왼쪽에서 오른쪽으로 진행하면서 언어 프로세서는 다음과 같이 표현식을 평가합니다.

- 먼저 소괄호 내의 표현식을 평가합니다.
  - 그런 다음 우선순위가 낮은 연산자가 있는 표현식보다 우선순위가 높은 연산자가 있는 표현식을 평가합니다.
- 산술 연산자 우선순위는 다음과 같습니다(첫 번째가 가장 높음).

표 1. 산술 연산자 우선순위	
연산자 기호	연산자 설명
- +	접두부 연산자
**	Power®(지수)
* / % //	곱하기와 나누기
+ -	더하기와 빼기

따라서 위의 예는 다음 순서로 평가됩니다.

#### 1. 소괄호 안의 표현식

$$7 + 2 * \left( \frac{9}{3} \right) - 1$$

#### 2. 곱하기

$$7 + \frac{2 * 3}{6} - 1$$

#### 3. 왼쪽에서 오른쪽으로 더하기와 빼기

$$7 + 6 - 1 = 12$$

### 산술 연산식 사용

프로그램에서 여러 가지 방식으로 산술 연산식을 사용할 수 있습니다. 다음 예제는 몇 개의 산술 연산자를 사용하여 달러와 센트 값에서 소수점 이하 자릿수를 반올림하고 제거합니다.

```

/***** 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'.
```

그림 10. 산술 연산식 사용 예

### 연습: 산술 연산식 계산

다음 프로그램은 어떤 출력 행을 생성합니까?

```

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

다음의 값은 무엇입니까?

1.  $6 - 4 + 1$
2.  $6 - (4 + 1)$
3.  $6 * 4 + 2$

4.  $6 * (4 + 2)$

5.  $24 \% 5 / 2$

#### ANSWERS

1. 이 가족에는 5명이 있습니다.

2. 값은 다음과 같습니다.

a. 3

b. 1

c. 26

d. 36

e. 2

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## 비교 연산자

비교 연산자는 숫자 또는 문자열을 비교하고 평가를 수행할 수 있습니다. 비교 연산자를 사용하는 표현식은 산술 표현식과 같이 숫자 값을 리턴하지 않습니다. 비교 표현식은 **true**를 표시하는 **1**을 리턴하거나 **false**를 표시하는 **0**을 리턴합니다.

다음은 일반적인 비교입니다.

항이 같습니까?( $A = Z$ )

첫 번째 항이 두 번째 항보다 큼습니까?( $A > Z$ )

첫 번째 항이 두 번째 항보다 작습니까?( $A < Z$ )

예를 들어,  $A = 4$  및  $Z = 3$ 인 경우, 이전 비교 질문의 결과는 다음과 같습니다.

• ( $A = Z$ )는  $4 = 3$ 입니까? 0(False)

• ( $A > Z$ )는  $4 > 3$ 입니까? 1(True)

• ( $A < Z$ )는  $4 < 3$ 입니까? 0(False)

### 일반적으로 사용되는 비교 연산자

#### 연산자

##### 의미

=

같음

==

엄격하게 같음

\=

같지 않음

\==

엄격하게 같지 않음

>

초과

<

미만

><

크거나 작음(같지 않음과 동일)

>=

크거나 같음

\<  
보다 작지 않음

<=  
보다 작거나 같음

\>  
보다 크지 않음

**참고:** NOT 문자(~)는 백슬래시(\)와 동의어입니다. 사용 가능 여부 및 개인 취향에 따라 두 문자를 서로 바꿔 사용할 수 있습니다. 이 정보는 백슬래시(\) 문자를 사용합니다.

### 엄격하게 같음 및 같음 연산자

두 표현식이 **엄격하게 같음**이면, 공백과 대소문자(표현식이 문자인 경우)를 포함한 모든 항목이 정확히 동일합니다.

두 표현식이 **같음**이면 다음은 동일한 것으로 해석됩니다. 다음 표현식은 모두 true입니다.

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

### 비교식 사용

IF...THEN...ELSE 명령문에서 종종 비교 표현식을 사용합니다. 다음 예제에서는 두 개의 값을 비교하기 위해 IF...THEN...ELSE 명령어를 사용합니다. 이 명령어에 대한 자세한 정보는 [29 페이지의 『IF...THEN...ELSE 명령어』](#)의 내용을 참조하십시오.

```
/****** 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."
```

그림 11. 비교식을 사용하는 예

### 연습: 비교식 사용

비교식을 사용하는 앞의 예를 기반으로 다음 점심 비용으로부터 언어 프로세서가 생성하는 결과는 무엇입니까?

**Yesterday's Lunch**  
**Today's Lunch**

**4.42**  
3.75

**3.50**  
3.50

**3.75**  
4.42

다음 표현식의 결과(0 또는 1)는 무엇입니까?

1. "Apples" = "Oranges"
2. "Apples" = "Apples"
3. " Apples" == "Apples"
4. 100 = 1E2

5. `100 \= 1E2`
6. `100 \== 1E2`

## ANSWERS

1. 언어프로세서는 다음 문장을 생성합니다.
  - 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. 표현식은 결과적으로 다음이 됩니다. 0은 false이고 1은 true입니다.
  - a. 0
  - b. 1
  - c. 0 (첫 번째 "Apples"에는 공백이 있습니다.)
  - d. 1
  - e. 0
  - f. 1

## 논리(부울) 연산자

비교식과 같이 논리식은 처리될 때 1(true) 또는 0(false)을 리턴합니다. 논리 연산자는 두 개의 비교를 결합시키고, 비교의 결과에 따라 1 또는 0을 리턴합니다.

### 논리 연산자

연산자  
의미

**&**

AND

양쪽 비교가 true이면 1을 리턴합니다. 예를 들어, 다음과 같습니다.

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

**|**

포함적 OR

한 개 이상의 비교가 true이면 1을 리턴합니다. 예를 들어, 다음과 같습니다.

```
(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 */
```

**&&**

배타적 OR

양쪽 모두가 아닌 단 하나의 비교가 true이면 1을 리턴합니다. 예를 들어, 다음과 같습니다.

```
(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
*/
```

접두부 `\,`  
논리 NOT

부정 — 반대 응답을 리턴합니다. 예를 들어, 다음과 같습니다.

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

## 논리식 사용

원하지 않는 조건을 차단하기 위해 복합 조건부 명령어에서 체크포인트로 논리식을 사용할 수 있습니다. 일련의 논리식을 가지고 있는 경우 설명을 위해 한 개 이상의 괄호 세트를 사용하여 각 표현식을 묶습니다.

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

다음 예는 논리 연산자를 사용하여 의사결정합니다.

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

### 그림 12. 논리식 사용 예

이 예제에 전달된 인수가 SPRING YES NO인 경우 IF 절은 다음과 같이 변환됩니다.

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

결과적으로 프로그램을 실행하면 다음 결과가 생성됩니다.

```
Go skiing.
```

## 연습: 논리식 사용

대학에 지원하는 학생은 다음 스펙에 따라 대학을 평가하기로 결정했습니다.

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

대학은 저렴하고 장학금을 제공하지 않았고 평판이 좋지만 1000마일 이상 떨어져 있습니다. 학생이 지원해야 합니까?

ANSWER

예. 조건부 명령은 다음과 같이 작동됩니다.

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

## 연결 연산자

연결 연산자는 두 개의 용어를 하나로 결합시킵니다. 용어는 문자열, 변수, 표현식 또는 상수일 수 있습니다. 연결은 출력 형식에 중요할 수 있습니다.

두 개의 항을 조인하는 방법을 표시하는 연산자는 다음과 같습니다.

### 연산자 의미

#### blank

항을 연결하고 중간에 하나의 공백을 둡니다. 두 개 이상의 공백이 항을 분리하면 단일 공백이 됩니다. 예를 들어, 다음과 같습니다.

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

#### ||

항 사이에 공백없이 항을 연결시킵니다. 예를 들어, 다음과 같습니다.

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

#### abuttal

항목 사이에 공백없이 용어를 연결시킵니다. 예를 들어, 다음과 같습니다.

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

리터럴 문자열 및 기호와 같이 다른 유형의 용어, 또는 주석만 두 용어를 분리하는 경우에만 abuttal을 사용할 수 있습니다.

## 연결 연산자 사용

출력을 형식화하는 한 가지 방법은 다음 예와 같이 변수 및 연결 연산자를 사용하는 것입니다.

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

그림 13. 연결 연산자를 사용하는 예

이 예의 결과는 다음과 같습니다.

```
baseball      $5
```

정보를 형식화하는 보다 정교한 방법은 구문 분석 및 템플릿을 사용하는 것입니다. 77 페이지의 『데이터 구문 분석』의 내용을 참조하십시오.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## 연산자의 우선순위

표현식에 여러 유형의 연산자가 있으면 언어 프로세서는 모든 연산자를 포함하는 전체 우선순위를 따릅니다.

연산자의 우선 순위는 다음과 같습니다(높은 순서대로).



표 2. 전체 연산자 우선순위	
연산자 기호	연산자 설명
\ 또는 ~ - +	접두부 연산자
**	거듭제곱(지수)
* / % //	곱하기 및 나누기
+ -	더하기 및 빼기
<i>blank</i>    <i>abuttal</i>	연결 연산자
== = > < 등등	비교 연산자
&	논리 AND
&&	포함 OR 및 독점 OR

## 예

다음 표현식은 둘 이상의 연산자 유형을 포함합니다.

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

다음 값이 주어지면

A = 8

B = 2

C = 10

언어 프로세서는 이 예를 다음과 같이 평가합니다.

- 첫 번째 소괄호 세트 안에 무엇이 있는지 평가합니다.
  - A를 8로 평가합니다.
  - B를 2로 평가합니다.
  - 7\*\*2를 평가합니다.
  - 8 > 49는 false(0)로 평가합니다.
- 다음 소괄호 세트 안에 무엇이 있는지 평가합니다.
  - B를 2로 평가합니다.
  - 2 < 3은 true(1)로 평가합니다.
- 0 & 1을 0으로 평가합니다.

## 연습: 연산자의 우선순위

- 다음 예제에 대한 답은 무엇입니까?
  - 22 + (12 \* 1)
  - 6 / -2 > (45 % 7 / 2) - 1
  - 10 \* 2 - (5 + 1) // 5 \* 2 + 15 - 1
- 24 페이지의 『논리(부울) 연산자』의 이전 연습에서 나온 학생과 대학의 예에서 소괄호가 학생의 공식에서 제거되면 대학의 결과는 어떻게 됩니까?

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

대학은 저렴하고 장학금을 제공하지 않았고 평판은 좋지만 1000마일 떨어져 있다는 것을 기억하십시오.

## ANSWERS

1. 결과는 다음과 같습니다.

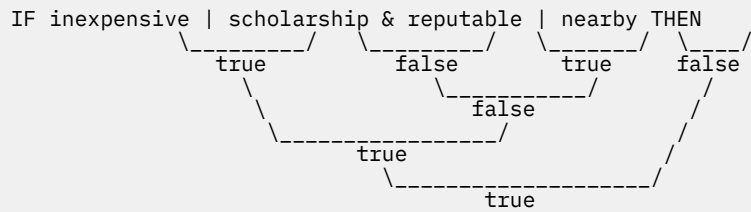
a. 34 ( $22 + 12 = 34$ )

b. 1 (true) ( $3 > 3 - 1$ )

c. 32 ( $20 - 2 + 15 - 1$ )

2. 이를 고려해 보겠습니다.

& 연산자에는 다음과 같은 우선순위가 있지만 결과는 괄호가 있는 이전 버전과 동일합니다.



## 제 4 장 프로그램 내 플로우 제어

일반적으로 프로그램이 실행되면 첫 번째로 시작하여 마지막으로 끝나는 명령어가 차례로 실행됩니다. 별도의 지시가 없는 한 언어 프로세서는 명령어를 순차적으로 실행합니다. 언어 프로세서가 일부 명령어를 건너뛰거나 다른 명령어를 반복하거나 제어를 프로그램의 다른 파트로 전송하게 하는 REXX 명령어를 사용하여 프로그램 내에서 실행 순서를 변경할 수 있습니다.

### 이 태스크 정보

REXX 프로그램의 순차 실행을 변경하는 REXX 명령어는 다음과 같이 분류할 수 있습니다.

#### 조건부 명령어

조건부 명령어는 표현식 양식으로 하나 이상의 조건을 설정합니다. 조건이 true이면 언어 프로세서는 해당 조건을 따르는 경로를 선택합니다. 그렇지 않으면 언어 프로세서는 다른 경로를 선택합니다. REXX 조건부 명령어는 다음과 같습니다.

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

#### 루프 명령어

루프 명령어는 언어 프로세서에 명령어 세트를 반복하도록 지시합니다. 루프는 지정된 횟수만큼 반복하거나 조건을 사용하여 반복을 제어할 수 있습니다. REXX 루프 명령어는 다음과 같습니다.

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

#### 인터럽트 명령어

인터럽트 명령어는 언어 프로세서가 프로그램을 완전히 떠나거나 프로그램의 한 파트를 떠나 다른 파트로 영구적으로 또는 일시적으로 이동하도록 지시합니다. REXX 인터럽트 명령어는 다음과 같습니다.

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

## 조건부 명령어

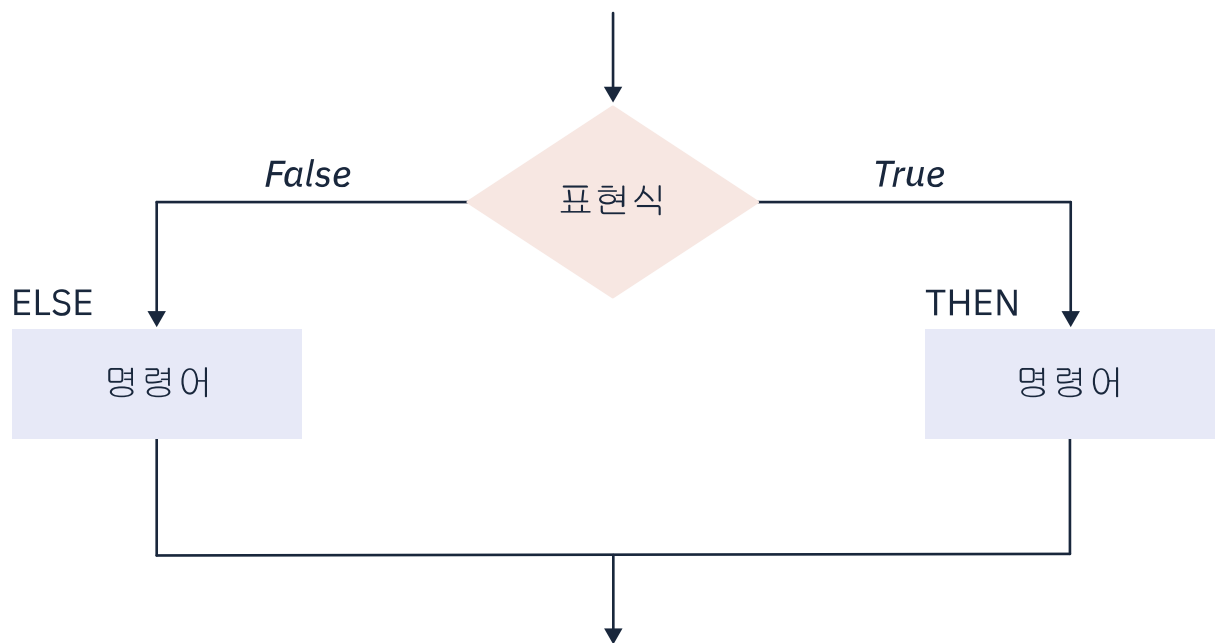
두 가지 유형의 조건부 명령어가 있습니다.

- IF...THEN...ELSE는 프로그램의 실행을 두 가지 옵션 중 하나로 지정할 수 있습니다.
- SELECT WHEN...OTHERWISE...END는 해당 실행을 여러 옵션 중 하나로 지정할 수 있습니다.

### IF...THEN...ELSE 명령어

코드 예제는 IF...THEN...ELSE 명령어를 지정하는 방법을 배우는 데 도움이 됩니다.

다음 플로우차트는 IF...THEN...ELSE 명령어를 설명합니다.



REXX 명령어로서 코드는 다음과 같습니다.

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

가독성을 높이기 위해 다음 방법 중 하나로 절을 정렬할 수도 있습니다.

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

또는

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

전체 명령을 한 행에 넣을 경우 ELSE 앞에 세미콜론을 사용하여 THEN 절과 ELSE 절을 구분해야 합니다.

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

일반적으로, 하나 이상의 명령어는 THEN 및 ELSE 절을 따라야 합니다. 어느 한 절에 명령어가 없으면 절 옆에 NOP(오퍼레이션 없음)를 포함시키는 것이 좋은 프로그래밍 습관입니다.

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

조건에 대해 둘 이상의 명령어가 있는 경우 DO로 명령어 세트를 시작하고 END로 종료하십시오.

```
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
```

DO와 END를 묶지 않으면 언어 프로세서는 ELSE 절에 대해 하나의 명령어만 가정합니다.

## 중첩된 IF...THEN...ELSE 명령어

코드 예제는 IF...THEN...ELSE 명령어를 지정하는 방법을 배우는 데 도움이 됩니다.

때때로 다른 IF...THEN...ELSE 명령어 내에 하나 이상의 IF...THEN...ELSE 명령어가 있어야 합니다. 한 유형의 명령어 내에 다른 명령어가 있으면 이를 중첩이라고 합니다. 중첩된 IF 명령어를 사용하면 각 IF를 ELSE로, 각 DO를 END로 맞추는 것이 중요합니다.

```
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!'
```

중첩된 IF를 ELSE로, DO를 END로 맞추지 않으면 몇 가지 놀라운 결과가 발생할 수 있습니다. 다음 예와 같이 DO와 ND 및 ELSE NOP를 제거하면 결과는 어떻습니까?

```

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

```

그림 14. 누락된 명령어의 예제

프로그램을 살펴보면 ELSE가 첫 번째 IF에 속한다고 가정할 수 있습니다. 그러나 언어 프로세서는 ELSE를 가장 가까운 쌍이 아닌 IF와 연관시킵니다. 결과는 다음과 같습니다.

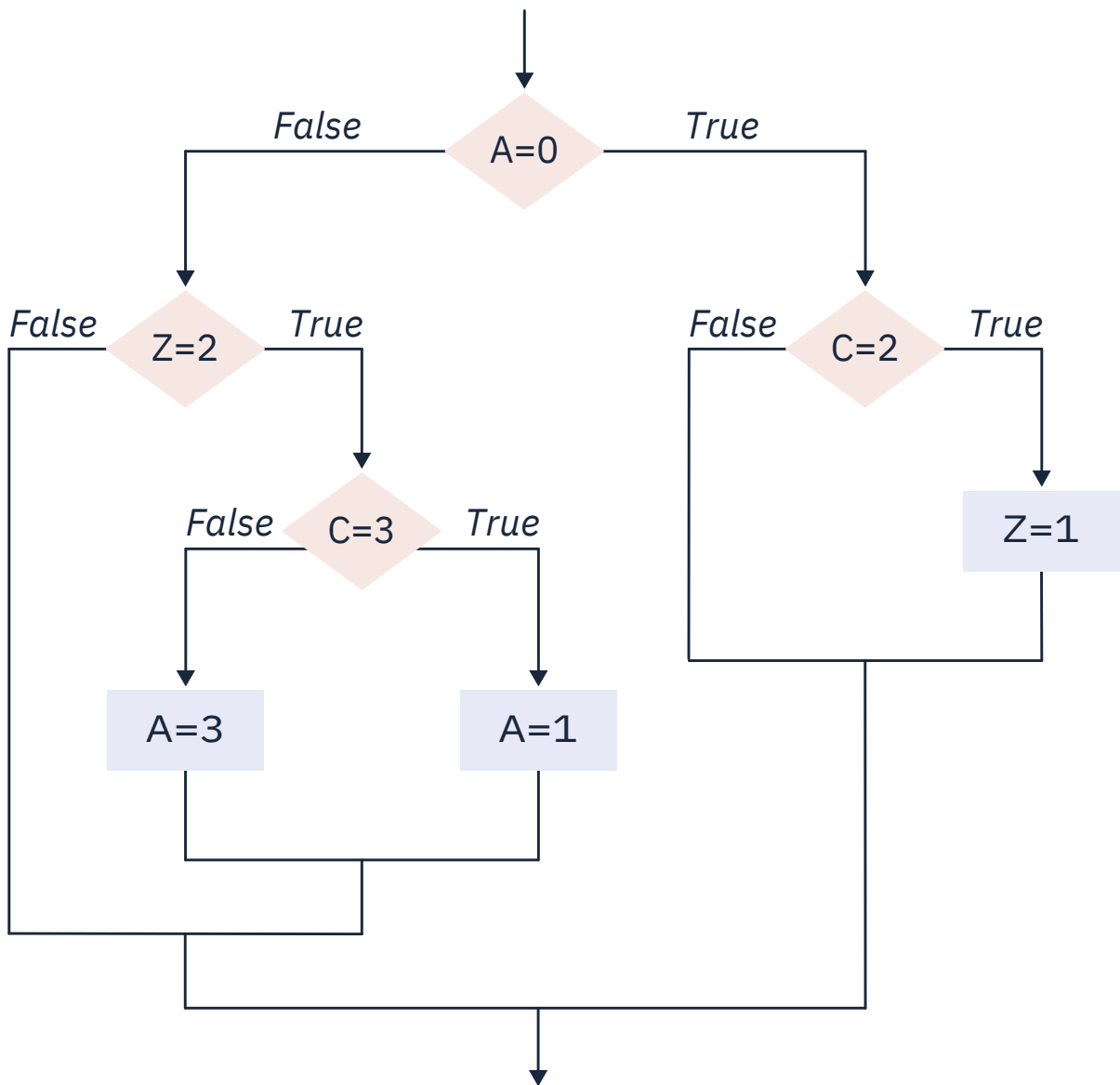
```

What a lovely day!
We should take our raincoats!

```

### 연습: IF...THEN...ELSE 명령어 사용

다음 플로우차트에 대한 REXX 명령어를 작성하십시오.



ANSWER

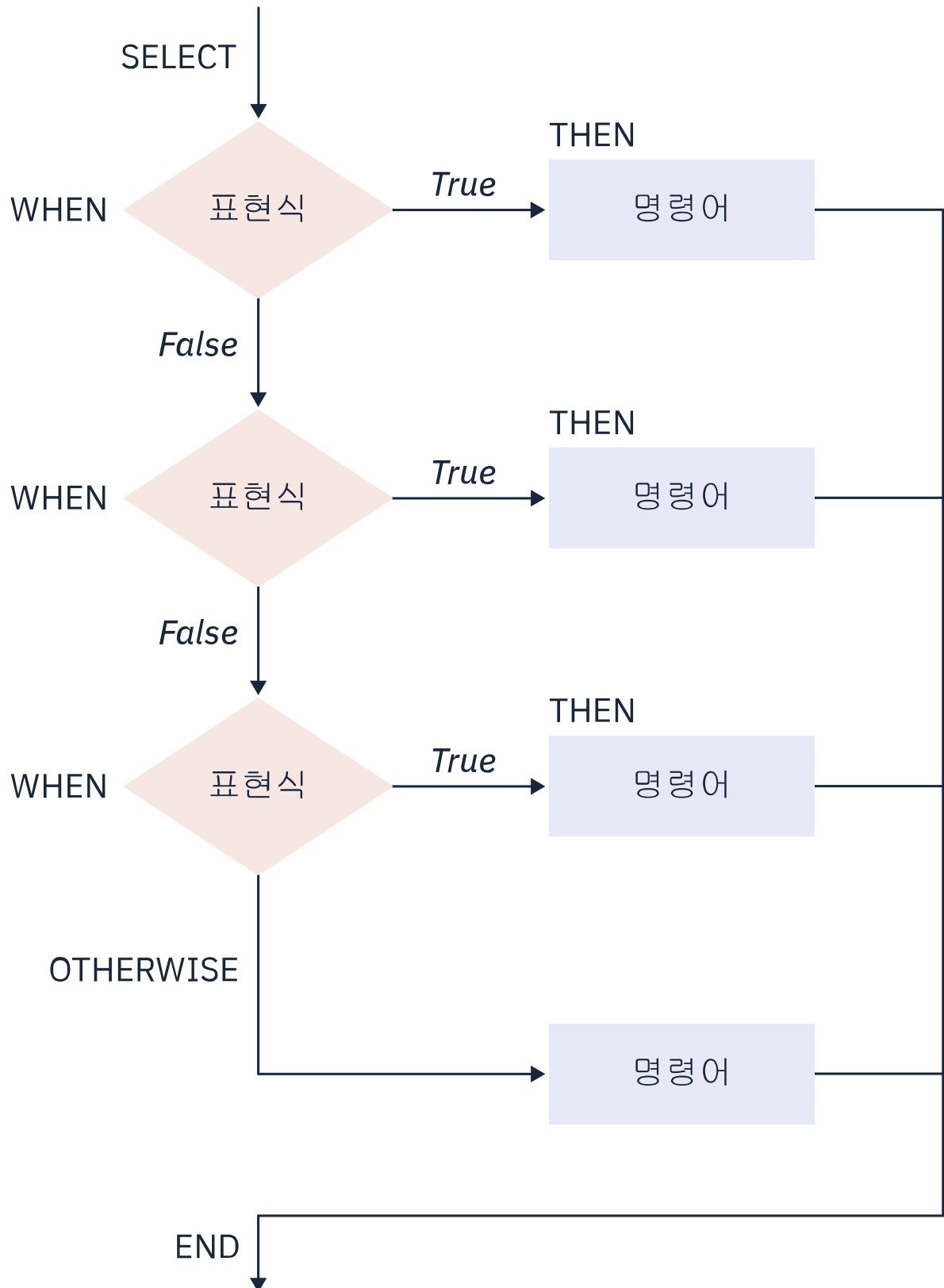
```
IF a = 0 THEN
  IF c = 2 THEN
    z = 1
  ELSE NOP
ELSE
  IF z = 2 THEN
    IF c = 3 THEN
      a = 1
    ELSE
      a = 3
    ELSE NOP
```

## **SELECT WHEN...OTHERWISE...END 명령어**

코드 예제는 SELECT WHEN...OTHERWISE...END 명령어를 지정하는 방법을 배우는 데 도움이 됩니다.

임의 수의 선택사항 중 하나를 선택하려면 SELECT WHEN...OTHERWISE...END 명령어를 사용하십시오. 플로 우차트에서는 다음과 같이 표시됩니다.





REXX 명령어로서 플로우차트 예제는 다음과 같습니다.

```

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

```

언어 프로세서는 처음부터 시작하여 true 표현식을 찾을 때까지 WHEN 절을 스캔합니다. true 표현식을 찾은 후에는 다른 모든 가능성(true일 수도 있지만)을 무시합니다. WHEN 표현식이 true이지 않으면 OTHERWISE 절 뒤에 오는 명령어를 처리합니다.

IF...THEN...ELSE와 마찬가지로 가능한 경로에 대한 명령어가 두 개 이상 있을 경우, DO로 명령어 세트를 시작하고 END로 종료하십시오. 그러나 OTHERWISE 키워드 뒤에 둘 이상의 명령어가 있으면 DO와 END가 필요하지 않습니다.

```

/***** 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'.'

```

그림 15. 예제 *SELECT WHEN...OTHERWISE...END* 사용

각 SELECT는 END로 끝나야 합니다. 각 WHEN을 들여쓰면 프로그램을 보다 쉽게 읽을 수 있습니다.

### 연습: *SELECT WHEN...OTHERWISE...END* 사용

"9월, 4월, 6월 및 11월에는 30일이 있습니다. 2월을 제외한 나머지 달에는 31일이 있습니다."

월을 표시하는 1- 12 사이의 숫자 입력을 사용하고 해당 월의 일 수를 생성하는 프로그램을 작성하십시오. 프로그램을 호출할 때 사용자가 인수로서 월을 나타내는 숫자를 지정한다고 가정하십시오. (프로그램에 month 숫자를 변수 month에 지정하는 ARG 명령어를 포함시키십시오). 그런 다음 프로그램에서 일 수를 생성하도록 하십시오. 2월의 경우, 이는 28 또는 29일 수 있습니다.

ANSWER

```

/***** 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'.'

```

그림 16. 가능한 솔루션

## 루프 명령어

두 가지 유형의 루프 명령어(반복 루프 및 조건부 루프)가 있습니다. 유형에 관계없이 모든 루프는 DO 키워드로 시작하고 END 키워드로 끝납니다.

반복 루프를 사용하면 특정 횟수만큼 명령을 반복할 수 있습니다. 조건부 루프는 조건을 사용하여 반복을 제어합니다. 두 가지 유형의 조건부 루프(DO WHILE 및 DO UNTIL)가 있습니다.

### 반복 루프

가장 간단한 반복 루프는 언어 프로세서에 명령 그룹을 특정 횟수만큼 반복하도록 지시합니다. 키워드 DO 뒤에 상수를 사용합니다.

다음 예제를 실행하면 다섯 행의 Hello!가 생성됩니다.

```

DO 5
SAY 'Hello!'
END

```

```

Hello!
Hello!
Hello!
Hello!
Hello!

```

다음 예제와 같이 상수 대신 변수를 사용하여 동일한 결과를 얻을 수 있습니다.

```

number = 5
DO number
  SAY 'Hello!'
END

```

루프 반복 횟수를 제어하는 변수를 제어 변수라고 합니다. 달리 지정하지 않으면 루프가 반복될 때마다 제어 변수가 1씩 증가합니다.

```

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

```

이 예의 결과는 루프 번호가 선행하는 다섯 행의 Hello!가 표시됩니다. 루프 하단에서 번호가 증가하고 상단에서 테스트됩니다.

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

다음과 같이 키워드 BY를 사용하여 제어 변수의 증분을 변경할 수 있습니다.

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

이 예에서는 루프의 번호가 2씩 증가한다는 점을 제외하면 이전 예와 유사한 결과가 발생합니다.

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

## 무한 루프

루프의 제어 변수가 마지막 숫자를 얻을 수 없으면 어떻게 됩니까? 예를 들어, 다음 프로그램 세그먼트에서 count는 1을 초과하여 증가하지 않습니다.

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

count가 1과 0 사이에서 번갈아 바뀌어 Number 1이라고 하는 끝없는 수의 행의 생성되므로 결과를 무한 루프라고 합니다.

프로그램이 무한 루프 상태인 경우 운영자에게 문의하여 취소하십시오. 권한이 있는 사용자는 **CEMT SET TASK PURGE** 명령을 발행하여 exec를 정지할 수 있습니다.

## DO FOREVER 루프

때로는 의도적으로 무한 루프를 작성할 수 있습니다(예: 파일 끝에 도달할 때까지 파일에서 레코드를 읽는 프로그램에서). 다음 예와 같이 조건이 충족되면 EXIT 명령어를 사용하여 무한 루프를 종료할 수 있습니다. EXIT 명령어에 대한 추가 정보는 [46 페이지의 『EXIT 명령어』](#)의 내용을 참조하십시오.

```

/***** REXX *****/
/* This program processes strings until the value of a string is */
/* a null string. */
/***** REXX *****/
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

```

그림 17. DO FOREVER 루프를 사용하는 예

이 예제는 처리를 위해 사용자 작성 함수에 문자열을 보낸 후 처리가 성공적으로 완료되었거나 실패했다는 메시지를 발행합니다. 입력 문자열이 공백이면 루프가 종료되고 프로그램도 종료됩니다. LEAVE 명령어를 사용하여 프로그램을 종료하지 않고 루프를 종료할 수도 있습니다.

## LEAVE 명령어

LEAVE 명령어는 반복 루프에서 즉시 빠져 나옵니다. 제어는 루프의 END 키워드 다음에 나오는 명령으로 이동합니다. LEAVE 명령어를 사용하는 예는 다음과 같습니다.

```

/***** REXX *****/
/* This program uses the LEAVE instruction to exit from a DO */
/* FOREVER loop. */
/***** REXX *****/
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.'

```

그림 18. LEAVE 명령어를 사용하는 예

## ITERATE 명령어

ITERATE 명령어는 루프 내에서 실행을 중지하고 루프 상단의 DO 명령어로 제어를 전달합니다. DO 명령어의 유형에 따라, 언어 프로세서는 제어 변수를 증가 및 테스트하거나 루프를 반복할지 여부를 판별하는 조건을 테스트합니다. LEAVE와 마찬가지로 ITERATE는 루프 내에서 사용됩니다.

```

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

```

이 예는 숫자 8을 제외하고 1에서 10까지의 숫자 목록을 생성합니다.

```

Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7

```

```
Number 9
Number 10
```

### 연습: 루프 사용

다음 루프의 결과는 무엇입니까?

```
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
```

때때로 루프를 종료하기 위한 입력이 예상과 일치하지 않을 때 무한 루프가 발생할 수 있습니다. 예를 들어, LEAVE 명령어 사용의 예에서 입력이 Quit이고 PARSE PULL 명령어가 PULL 명령어를 대체하면 어떻게 됩니까?

```
PARSE PULL file_name
```

### ANSWERS

1. 반복 루프의 결과는 다음과 같습니다.

a. 123Digit is now 4

b. 10  
86  
Count is now 4

c. Index is now 10

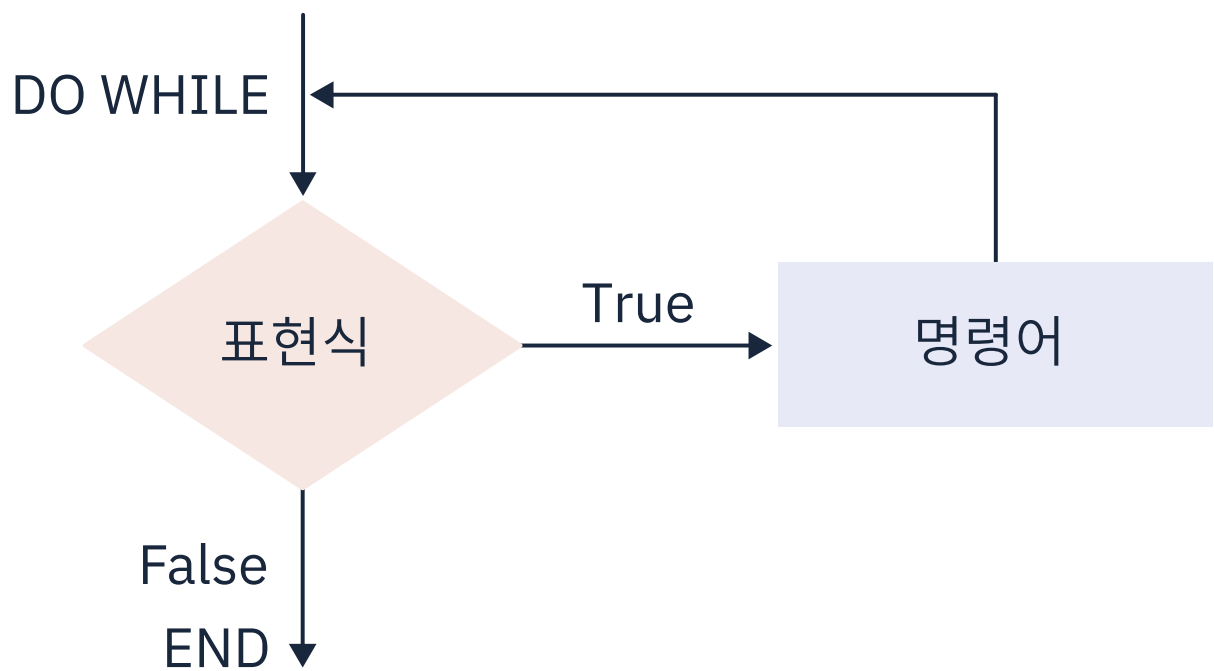
2. Quit가 QUIT와 같지 않으므로 프로그램이 루프를 벗어날 수 없습니다. 이 경우 입력이 quit, QUIT 또는 Quit인지에 관계없이 언어 프로세서는 입력을 QUIT와 비교하기 전에 대문자로 변환하므로 PARSE 키워드를 생략하는 것이 좋습니다.

### 조건부 루프

두 가지 유형의 조건부 루프(DO WHILE 및 DO UNTIL)가 있습니다. 하나 이상의 표현식은 두 유형의 루프를 제어합니다. 그러나 DO WHILE 루프는 루프가 처음 실행되기 전에 표현식을 테스트하고 표현식이 true인 경우에만 반복합니다. DO UNTIL 루프는 루프가 적어도 한 번 실행된 후 표현식을 테스트하고 표현식이 false인 경우에만 반복합니다.

#### DO WHILE 루프

플로우차트에서 DO WHILE 루프는 다음과 같이 표시됩니다.



REXX 명령어로서 플로우차트 예제는 다음과 같습니다.

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

조건이 true인 동안 루프를 실행하려면 DO WHILE 루프를 사용하십시오. DO WHILE은 루프의 상단에서 조건을 테스트합니다. 조건이 처음에 false이면 언어 프로세서는 루프를 실행하지 않습니다.

37 페이지의 『반복 루프』에서 LEAVE 명령어를 사용하는 예에서 DO FOREVER 루프 대신 DO WHILE 루프를 사용할 수 있습니다. 그러나 루프에 들어가기 전에 조건이 테스트될 수 있도록 첫 번째 케이스의 루프를 초기화해야 합니다. 다음 예제의 첫 번째 PULL에서 첫 번째 케이스 초기화에 주목하십시오.

```
/****** 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.'
```

그림 19. DO WHILE 사용 예

### 연습: DO WHILE 루프 사용

통근 항공사 승객이 창가 자리를 원하는지 여부에 대한 응답 목록을 입력으로 사용하는 DO WHILE 루프가 있는 프로그램을 작성하십시오. 항공편에는 8명의 승객과 4개의 창가 자리가 있습니다. 모든 창가 자리가 잡힐 때 루프를 중지하십시오. 루프가 종료되면 잡힌 창가 좌석 수와 처리된 응답 수를 생성하십시오.

#### 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

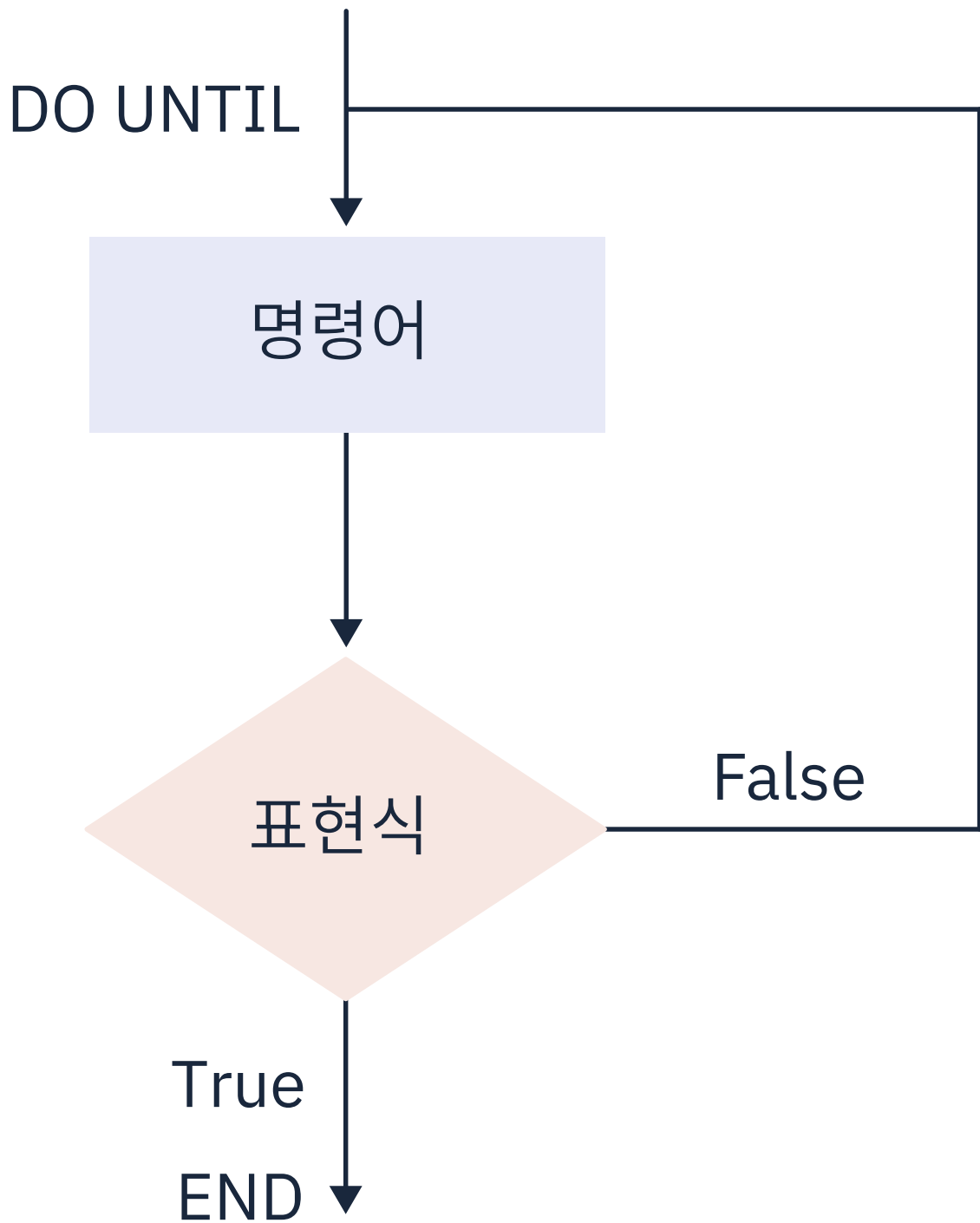
SAY window_seats 'window seats were assigned.'
SAY passenger 'passengers were questioned.'
```

그림 20. 가능한 솔루션

### DO UNTIL 루프

플로우차트에서 DO UNTIL 루프는 다음과 같이 표시됩니다.





REXX 명령어로서 플로우차트 예제는 다음과 같습니다.

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

조건이 true가 아니고 조건이 true가 될 때까지 루프를 실행하려면 DO UNTIL 루프를 사용하십시오. DO UNTIL 루프는 루프 끝에서 조건을 테스트하고 조건이 false인 경우에만 반복합니다. 그렇지 않으면, 루프는 한 번 실행되고 종료됩니다. 예를 들어, 다음과 같습니다.

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

그림 21. DO UNTIL 사용 예제

### 연습: DO UNTIL 루프 사용

이전 연습의 프로그램을 DO WHILE에서 DO UNTIL 루프로 변경하고 동일한 결과를 얻으십시오. DO WHILE 루프는 true 표현식을 확인하고 DO UNTIL 루프는 false 표현식을 확인합니다. 즉, 해당 논리 연산자가 종종 반전됨을 의미합니다.

#### ANSWER

```
/****** REXX ******/
/* This program uses a DO UNTIL 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 UNTIL (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.'
```

그림 22. 가능한 솔루션

## 복합 루프

반복 루프와 조건부 루프를 결합하여 복합 루프를 작성할 수 있습니다.

다음 루프는 quantity가 50 미만인 동안 10회 반복되도록 설정되며, 50인 경우 중지됩니다.

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

이 예의 결과는 다음과 같습니다.

```
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 루프를 대체하고 비교식을 <에서 >로 변경하고 동일한 결과를 얻을 수 있습니다.

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

## 중첩된 DO 루프

중첩된 IF...THEN...ELSE 명령어와 같이 DO 루프는 다른 DO 루프를 포함할 수 있습니다.

간단한 예는 다음과 같습니다.

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

이 예의 출력은 다음과 같습니다.

```
HIP
HIP
HURRAH
HIP
HIP
HURRAH
```

특정 조건이 발생할 때 루프를 떠나야 하는 경우 LEAVE 명령어 다음에 루프의 제어 변수 이름을 사용하십시오. LEAVE 명령어가 내부 루프용인 경우 처리는 내부 루프를 떠나 외부 루프로 이동합니다. LEAVE 명령어가 외부 루프용인 경우 처리는 두 루프 모두 떠납니다.

앞의 예제에서 내부 루프를 떠나려면 IF 명령어 뒤에 LEAVE 명령어가 포함된 IF... THEN... ELSE 명령어를 추가하십시오.

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

결과는 다음과 같습니다.

```
HIP
HURRAH
HIP
HURRAH
```

## 연습: 루프 결합

1. 다음 프로그램이 실행되면 어떻게 됩니까?

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

## 2. 이제 LEAVE 명령어가 추가되면 어떻게 됩니까?

```
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
```

### ANSWERS

#### 1. 이 예제가 실행되면 다음이 생성됩니다.

```
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. 결과는 각 내부 루프에 대한 출력의 한 행입니다.

```
Outer 1 Inner 1

Outer 2 Inner 1

Outer 3 Inner 1
```

## 인터럽트 명령어

인터럽트 명령어에는 EXIT, SIGNAL, CALL 및 RETURN이 포함됩니다.

프로그램의 플로우를 인터럽트하는 명령어로 인해 프로그램이 다음과 같이 작동될 수 있습니다.

- 종료(EXIT)
- 레이블로 표시된 프로그램의 다른 파트로 건너뛰기(SIGNAL)
- 임시로 프로그램 내 또는 프로그램 외부의 서브루틴으로 이동(CALL 또는 RETURN).

### EXIT 명령어

EXIT 명령어는 REXX 프로그램이 무조건 종료되고 프로그램이 호출된 위치로 돌아갑니다. 다른 프로그램이 REXX 프로그램을 호출한 경우 EXIT는 해당 호출 프로그램으로 리턴합니다.

EXIT는 프로그램 종료 외에도 프로그램 호출자에게 값을 리턴할 수 있습니다. 프로그램이 다른 REXX 프로그램에서 서브루틴으로 호출된 경우 값은 REXX 특수 변수 RESULT에서 수신됩니다. 프로그램이 함수로 호출된 경우 함수가 호출된 지점에서 원래 표현식으로 값이 수신됩니다. 그렇지 않으면, 값은 REXX 특수 변수 RC에 수신됩니다. 값은 리턴 코드를 표시할 수 있으며 계산되는 상수 또는 표현식의 양식일 수 있습니다.

```

/***** 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

```

그림 23. EXIT 명령어를 사용하는 예

외부 루틴 호출에 대한 자세한 정보는 58 페이지의 『서브루틴 및 함수』의 내용을 참조하십시오. EXIT 명령어에 대한 세부사항은 EXIT를 참조하십시오.

## CALL 및 RETURN 명령어

CALL 명령어는 내부 또는 외부 서브루틴에 대한 제어를 전달하여 프로그램의 플로우를 인터럽트합니다. 내부 서브루틴은 호출 프로그램의 일부입니다. 외부 서브루틴은 다른 프로그램입니다. RETURN 명령어는 서브루틴에서 호출 프로그램으로 다시 제어를 리턴하고 선택적으로 값을 리턴합니다.

내부 서브루틴을 호출하면 CALL은 CALL 키워드 뒤에 지정된 레이블로 제어를 전달합니다. 서브루틴이 RETURN 명령어로 종료되면 CALL 다음의 명령어가 처리됩니다.

명령어

CALL sub1

명령어

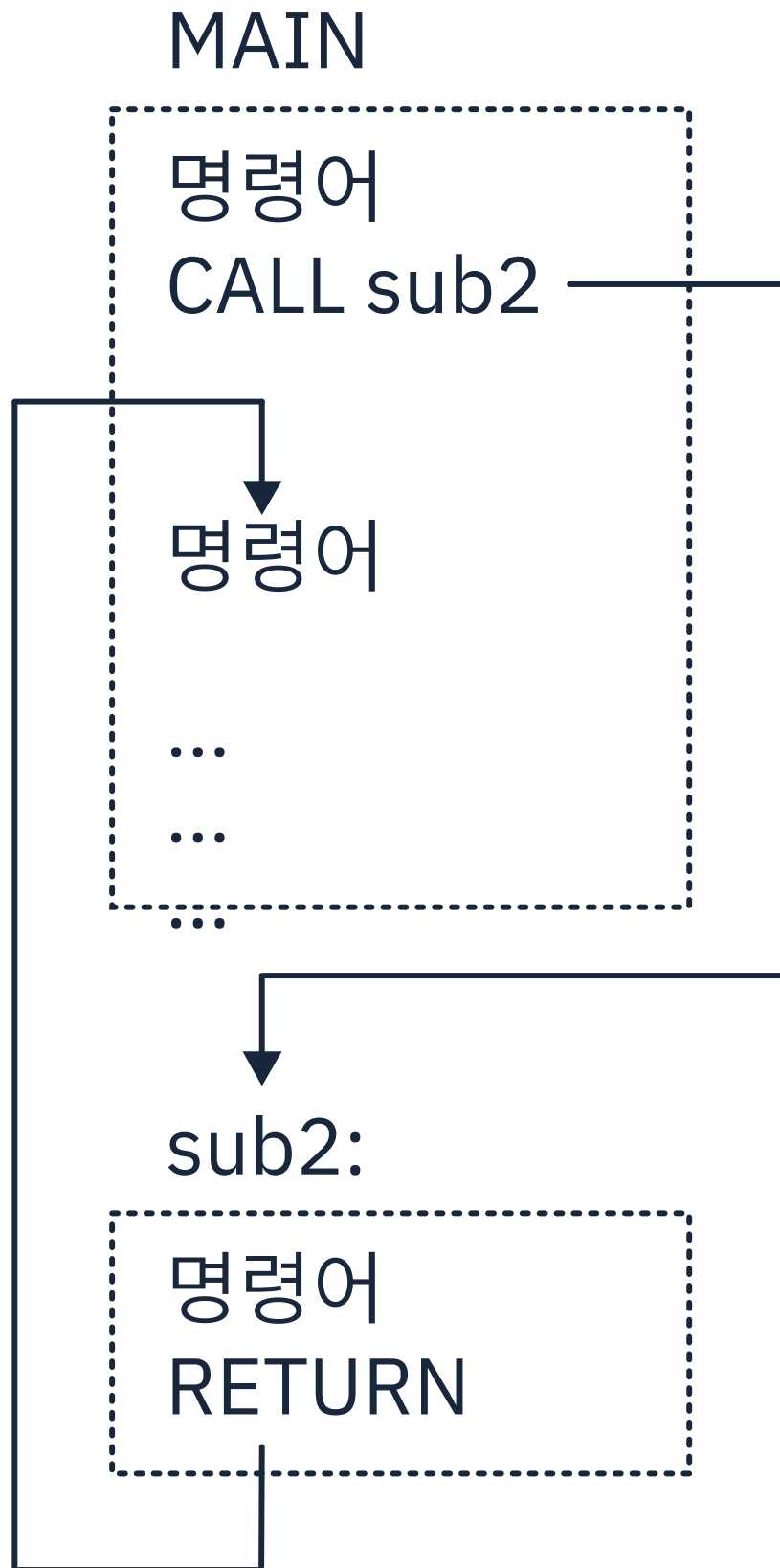
EXIT

sub1:

명령어

RETURN

외부 서브루틴을 호출하면 CALL은 CALL 키워드 뒤에 지정된 프로그램 이름으로 제어를 전달합니다. 외부 서브루틴이 완료되면 RETURN 명령어를 사용하여 호출 프로그램에서 중단한 위치로 돌아갈 수 있습니다.



서브루틴 호출에 대한 자세한 정보는 58 페이지의 『서브루틴 및 함수』의 내용을 참조하십시오.

REXX가 예외 조건(예: 오류 또는 실패 조건)을 감지하면 제어를 수신하는 루틴을 구현하기 위해 **ON** 매개변수와 함께 CALL 명령어를 사용할 수 있습니다. [조건 및 조건 트랩의 내용을 참조하십시오.](#)

CALL 및 RETURN 명령어에 대한 세부사항은 [CALL](#) 및 [RETURN](#)의 내용을 참조하십시오.

## SIGNAL 명령어

CALL과 같이 SIGNAL 명령어는 프로그램의 일반적인 플로우를 중단시키고 지정된 레이블로 제어를 전달합니다. 제어가 전달되는 레이블은 SIGNAL 명령어 전후에 있을 수 있습니다. CALL과 달리 SIGNAL은 실행을 재개하기 위해 특정 명령으로 돌아가지 않습니다.

루프 내에서 SIGNAL을 사용하면 루프가 자동으로 종료됩니다. 내부 루틴에서 SIGNAL을 사용하면 내부 루틴이 호출자에게 리턴되지 않습니다.

SIGNAL은 프로그램을 테스트하거나 응급 조치를 제공하는 데 유용합니다. 프로그램의 한 곳에서 다른 곳으로 이동하는 편리한 방법으로서 사용해서는 안 됩니다. SIGNAL은 [CALL](#) 명령어와 마찬가지로 리턴하는 방법을 제공하지 않습니다.

REXX가 예외 조건(예: 오류 또는 실패 조건)을 감지하면 제어를 수신하는 루틴을 구현하기 위해 **ON** 매개변수와 함께 SIGNAL 명령어를 사용할 수 있습니다. [조건 및 조건 트랩의 내용을 참조하십시오.](#)

SIGNAL 명령어에 대한 자세한 정보는 91 페이지의 『REXX 특수 변수 RC 및 SIGL 사용』 및 [SIGNAL](#)의 내용을 참조하십시오.


## 예

다음 예제에서 표현식이 true인 경우 언어 프로세서는 레이블 Emergency:로 이동하고 그 사이의 모든 명령어를 건너뜁니다.



IF 표현식 THEN  
    SIGNAL 긴급  
ELSE  
    명령어

긴급 :  
명령어





## 제 5 장 함수

함수는 데이터를 수신하고 처리하고 값을 리턴할 수 있는 일련의 명령어입니다. REXX에는 여러 가지 유형의 함수가 있습니다.

- 기본 제공 함수가 언어 프로세서에 빌드되어 있습니다. 추가 정보는 [54 페이지의 『내장 함수』](#)의 내용을 참조하십시오.
- 사용자 작성 함수는 개별 사용자가 작성하거나 설치에서 제공하는 함수입니다. 이는 내부 또는 외부일 수 있습니다. 내부 함수는 레이블에서 시작하는 현재 프로그램의 일부입니다. 외부 함수는 프로그램 외부의 자체 포함 프로그램 또는 프로그램입니다. 사용자 작성 함수에 대한 정보는 [59 페이지의 『서브루틴 및 함수 작성』](#)의 내용을 참조하십시오.

함수의 유형과 관계없이 모든 함수는 함수 호출을 발행한 프로그램에 값을 리턴합니다. 함수를 호출하려면 함수 이름을 입력한 다음 함수에 인수를 묶는 소괄호를 붙입니다(있는 경우). **함수 이름과 왼쪽 괄호 사이에 공백을 넣을 수 없습니다.**

```
function(arguments)
```

함수 호출은 쉼표로 구분된 최대 20개의 인수를 포함할 수 있습니다. 인수는 다음과 같을 수 있습니다.

- 상수

```
function(55)
```

- 기호

```
function(symbol_name)
```

- 함수가 인식하는 옵션

```
function(option)
```

- 리터럴 문자열

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

- 특별히 지정되지 않았거나 생략됨

```
function()
```

- 다른 함수

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

- 인수 유형 조합

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

모든 함수는 값을 리턴해야 합니다. 함수가 값을 리턴할 때 값은 함수 호출을 대체합니다. 다음 예에서 언어 프로세서는 7에 함수가 리턴하는 값을 더해 합계를 생성합니다.

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

함수는 일반적으로 표현식에 표시됩니다. 즉, 표현식과 같은 함수 호출은 일반적으로 저절로 명령어에 표시되지 않습니다.

### 함수의 예

함수가 표시하는 계산에는 종종 많은 명령어가 필요합니다. 예를 들어, 세 개의 숫자 그룹에서 가장 높은 숫자를 찾기 위한 간단한 계산은 다음과 같이 작성될 수 있습니다.

```

/***** 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

```

그림 24. 최대 수 찾기

세 개 숫자 그룹의 최대값을 찾을 때마다 여러 개의 명령어를 쓰지 않고 계산을 수행하고 최대값을 리턴하는 기본 제공 함수를 사용할 수 있습니다. 해당 함수를 MAX라고 하며 다음과 같이 사용할 수 있습니다.

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

45, -2, number 및 199의 최대값을 찾고 기호 biggest에 최대를 넣으려면 다음 명령어를 작성하십시오.

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

## 내장 함수

50개 이상의 함수가 언어 프로세서에 빌드되어 있습니다.

기본 제공 함수는 다음 카테고리에 속합니다.

### 산술 함수

인수에서 숫자를 평가하고 특정 값을 리턴합니다.

### 비교 함수

수 또는 문자열 또는 둘 다를 비교하고 값을 리턴합니다.

### 변환 함수

한 유형의 데이터 표현을 다른 유형의 데이터 표현으로 변환합니다.

### 형식화 함수

인수에 제공된 문자열에서 문자와 간격을 조작합니다.

### 문자열 조작 함수

인수에 제공된 문자열(또는 문자열을 나타내는 변수)을 분석하고 특정 값을 리턴합니다.

### 기타 기능

다른 카테고리에 명확하게 맞지 않습니다.

다음 표에서는 각 카테고리의 함수를 간략하게 설명합니다. 이러한 함수에 대한 전체 설명은 [함수의 내용을 참조](#) 하십시오.

### 산술 함수

기능	설명
ABS	입력 숫자의 절대값을 리턴합니다.
DIGITS	NUMERIC DIGITS의 현재 설정을 리턴합니다.
FORM	NUMERIC FORM의 현재 설정을 리턴합니다.
FUZZ	NUMERIC FUZZ의 현재 설정을 리턴합니다.

기능	설명
MAX	지정된 목록에서 현재 NUMERIC 설정에 따라 형식화된 가장 큰 숫자를 리턴합니다.
MIN	지정된 목록에서 현재 NUMERIC 설정에 따라 형식화된 가장 작은 숫자를 리턴합니다.
RANDOM	지정된 범위에서 음수가 아닌 준랜덤 정수를 리턴합니다.
SIGN	입력 숫자의 부호를 표시하는 숫자를 리턴합니다.
TRUNC	입력 숫자의 정수 부분과 선택적으로 지정된 소수 자릿수를 리턴합니다.

#### 비교 함수

기능	설명
COMPARE	두 개의 입력 문자열이 동일하면 0을 리턴합니다. 그렇지 않으면, 일치하지 않는 첫 번째 문자의 위치를 리턴합니다.
DATATYPE	입력 문자열이 숫자 또는 문자와 같은 특정 데이터 유형임을 표시하는 문자열을 리턴합니다.
SYMBOL	기호의 상태(가변, 리터럴 또는 잘못된)를 표시하기 위해 VAR, LIT 또는 BAD를 리턴합니다.

#### 변환 함수

기능	설명
B2X	입력 2진 문자열의 16진 표현을 리턴합니다. (2진을 16진으로).
C2D	입력 문자열의 10진수 표현을 리턴합니다. (문자를 10진수로)
C2X	입력 문자열의 16진수 표현을 리턴합니다. (문자를 16진수로)
D2C	입력 10진 문자열의 문자 표현을 리턴합니다. (10진수를 문자로).
D2X	입력 10진 문자열의 16진 표현을 리턴합니다. (10진을 16진으로).
X2B	입력 16진 문자열의 2진 표현을 리턴합니다. (16진을 2진으로).
X2C	입력 16진 문자열의 문자 표현을 리턴합니다. (16진을 문자로).
X2D	입력 16진 문자열의 10진 표현을 리턴합니다. (16진을 10진으로).

#### 형식화 함수

기능	설명
CEnter 또는 CENTRE	입력 문자열을 중앙에 두고 길이를 구성하는 데 필요한 만큼 채움 문자를 추가하여 지정된 길이의 문자열을 리턴합니다.
COPIES	지정된 입력 문자열의 연결된 사본 수를 리턴합니다.
FORMAT	반올림되고 형식화된 입력 숫자를 리턴합니다.
JUSTIFY <sup>1</sup>	단어 사이에 채움 문자를 추가하여 양쪽 여백에 맞도록 지정된 문자열을 리턴합니다.
LEFT	필요에 따라 오른쪽에서 잘리거나 채워진 지정된 길이의 문자열을 리턴합니다.
RIGHT	필요에 따라 왼쪽에서 잘리거나 채워진 지정된 길이의 문자열을 리턴합니다.
SPACE	각 단어 사이에 지정된 수의 채움 문자를 사용하여 입력 문자열의 단어를 리턴합니다.

1. REXX/CICS가 제공하는 SAA가 아닌 기본 제공 함수입니다.

## 문자열 조작 함수

기능	설명
ABBREV	한 문자열이 다른 문자열의 지정된 선행 문자 수와 같은지 여부를 표시하는 문자열을 리턴합니다.
DELSTR	입력 문자열의 지정된 지점에서 시작하여 지정된 수의 문자를 삭제한 후 문자열을 리턴합니다.
DELWORD	입력 문자열의 지정된 단어에서 시작하여 지정된 수의 단어를 삭제한 후 문자열을 리턴합니다.
FIND <sup>1</sup>	입력 문자열에서 찾은 지정된 구문의 첫 번째 단어의 단어 번호를 리턴합니다.
INDEX <sup>1</sup>	입력 문자열에서 찾은 지정된 문자열의 첫 번째 문자의 문자 위치를 리턴합니다.
INSERT	지정된 문자 위치에서 다른 입력 문자열을 삽입한 후 문자열을 리턴합니다.
LASTPOS	한 문자열이 다른 문자열에서 마지막으로 발생한 시작 문자 위치를 리턴합니다.
LENGTH	입력 문자열의 길이를 리턴합니다.
OVERLAY	두 번째 입력 문자열로 오버레이된 대상 문자열인 문자열을 리턴합니다.
POS	한 문자열의 문자 위치를 다른 문자열로 리턴합니다.
REVERSE	원본과 반대의 문자열을 리턴합니다.
STRIP	입력 문자열에서 선행 또는 후행 문자 또는 둘 다를 제거한 후 문자열을 리턴합니다.
SUBSTR	지정된 문자 위치에서 시작하는 입력 문자열의 일부를 리턴합니다.
SUBWORD	지정된 단어 번호에서 시작하는 입력 문자열의 일부를 리턴합니다.
TRANSLATE	입력 문자열의 각 문자가 다른 문자로 변환되거나 변경되지 않은 문자열을 리턴합니다.
VERIFY	입력 문자열이 다른 입력 문자열의 문자로만 구성되는지 여부를 표시하는 숫자를 리턴하거나 일치하지 않는 첫 번째 문자의 문자 위치를 리턴합니다.
WORD	지정된 숫자가 표시하는 대로 입력 문자열로부터 단어를 리턴합니다.
WORDINDEX	지정된 단어에서 첫 번째 문자의 입력 문자열에서 문자 위치를 리턴합니다.
WORDLENGTH	입력 문자열에서 지정된 단어의 길이를 리턴합니다.
WORDPOS	입력 문자열에서 지정된 구문의 첫 번째 단어의 단어 번호를 리턴합니다.
WORDS	입력 문자열의 단어 수를 리턴합니다.

1. REXX/CICS가 제공하는 SAA가 아닌 기본 제공 함수입니다.

## 기타 기능

기능	설명
ADDRESS	현재 명령이 전송되는 환경의 이름을 리턴합니다.
ARG	프로그램 또는 내부 루틴에 인수 문자열 또는 인수 문자열에 대한 정보를 리턴합니다.
BITAND	두 개의 입력 문자열이 논리적으로 함께 AND된 비트 단위로 구성된 문자열을 리턴합니다.
BITOR	두 개의 입력 문자열이 논리적으로 함께 OR된 비트 단위로 구성된 문자열을 리턴합니다.
BITXOR	두 개의 입력 문자열이 독점적으로 OR된 비트 단위로 구성된 문자열을 리턴합니다.
CONDITION	현재 트랩된 조건과 연관된 이름 및 상태와 같은 조건 정보를 리턴합니다.
DATE	기본 형식(dd mon yyyy) 또는 다양한 선택적 형식 중 하나로 날짜를 리턴합니다.

기능	설명
ERRORTEXT	지정된 오류 번호와 연관된 오류 메시지를 리턴합니다.
EXTERNALS <sup>1</sup>	이 함수는 항상 0을 리턴합니다.
LINESIZE <sup>1</sup>	현재 출력 디바이스의 너비를 리턴합니다.
QUEUED	함수가 호출될 때 외부 데이터 큐에 남아 있는 행 수를 리턴합니다.
SOURCELINE	소스 파일에서 마지막 행의 행 번호 또는 숫자가 지정하는 소스 행을 리턴합니다.
TIME	기본 24시간 시계 형식(hh:mm:ss) 또는 다양한 선택적 형식 중 하나로 현지 시간을 리턴합니다.
TRACE	현재 유효한 추적 조치를 리턴합니다.
USERID <sup>1</sup>	현재 사용자 ID를 리턴합니다. 이는 SETUID 명령에 지정된 마지막 사용자 ID, 한 프로그램이 다른 프로그램을 호출하는 경우 호출 REXX 프로그램의 사용자 ID, 작업이 실행 중인 사용자 ID 또는 작업 이름입니다.
VALUE	지정된 기호의 값을 리턴하고 선택적으로 이 값에 새 값을 지정합니다.
XRANGE	지정된 시작 및 끝 값 사이에 포함된 모든 1바이트 코드(오름차순)의 문자열을 리턴합니다.

1. REXX/CICS가 제공하는 SAA가 아닌 기본 제공 함수입니다.

### 내장 함수를 사용한 입력 테스트

기본 제공 함수 중 일부는 입력을 테스트하기 위한 편리한 방법을 제공합니다. 프로그램이 입력을 사용할 때 사용자가 유효하지 않는 입력을 제공할 수 있습니다. 예를 들어 22 페이지의 『비교 연산자』에서의 비교 표현식을 사용하는 예에서 프로그램은 다음 명령어에서 달러 금액을 사용합니다.

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

프로그램이 숫자만 가져오면 프로그램은 해당 정보를 올바르게 처리합니다. 그러나 프로그램이 달러 부호 앞에 오는 숫자를 가져오거나 nothing과 같은 단어를 가져오면 오류를 리턴합니다. 오류가 발생하지 않도록 다음과 같이 DATATYPE 함수를 사용하여 입력을 확인할 수 있습니다.

```
IF DATATYPE(yesterday) \= 'NUM'
THEN DO
  SAY 'The input amount was in the wrong format.'
EXITEND
```

입력을 테스트하는 다른 유용한 내장 함수는 WORDS, VERIFY, LENGTH 및 SIGN입니다.

### 연습: 기본 제공 함수를 사용하여 프로그램 작성

8자 길이의 파일 이름인지 확인하는 프로그램을 작성하십시오. 이름이 8자를 초과하면 프로그램은 이름을 8까지 자르고 단축된 이름을 표시하는 메시지를 보냅니다. 기본 제공 함수 LENGTH 함수 및 SUBSTR(하위 문자열) 함수를 사용하십시오.

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

```

그림 25. 가능한 솔루션

## 서브루틴 및 함수

서브루틴 및 함수는 데이터를 수신하여 처리하고 값을 리턴할 수 있는 일련의 명령어로 구성된 루틴입니다.

루틴은 다음과 같을 수 있습니다.

### 내부

루틴은 현재 프로그램 내에 있으며 레이블로 표시되며 해당 프로그램만 루틴을 사용합니다.

### 외부

별도의 파일로 존재하는 REXX 서브루틴.

다양한 측면에서, 서브루틴과 함수는 동일합니다. 그러나 호출 방법 및 값을 리턴하는 방법과 같은 몇 가지 주요 측면에서 서로 다릅니다.

#### • 서브루틴 호출

서브루틴을 호출하려면 CALL 명령어와 서브루틴 이름(레이블 또는 프로그램 멤버 이름)을 사용하십시오. 선택적으로 쉼표로 구분된 최대 20개의 인수를 사용하여 이를 따를 수 있습니다. 서브루틴 호출은 전체 명령어입니다.

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

#### • 함수 호출

함수를 호출하려면 함수 이름(레이블 또는 프로그램 멤버 이름)과 인수를 포함할 수 있는 괄호를 바로 뒤에 사용하십시오. 함수 이름과 왼쪽 괄호 사이에 공백을 넣을 수 없습니다. 함수 호출은 명령어(예: 지정 명령어)의 파트입니다.

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

#### • 서브루틴에서 값 리턴

서브루틴은 값을 리턴할 필요가 없지만 이를 수행하면 RETURN 명령으로 값을 다시 보냅니다.

```
RETURN value
```

호출 프로그램은 RESULT라는 REXX 특수 변수의 값을 수신합니다.

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

#### • 함수에서 값 리턴

함수는 값을 리턴해야 합니다. 함수가 REXX 프로그램인 경우 해당 값은 RETURN 또는 EXIT 명령어와 함께 리턴됩니다.

```
RETURN value
```

호출 프로그램은 함수 호출 시 값을 수신합니다. 이 값은 함수 호출을 대체하므로 다음 예제에서 z = 값입니다.



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

### 함수 대신 서브루틴을 작성하는 경우

서브루틴 또는 함수를 구성하는 실제 명령어는 동일할 수 있습니다. 서브루틴 또는 함수로 변환하는 프로그램에서 사용할 방법입니다. 예를 들면, 기본 제공 함수 SUBSTR을 함수 또는 서브루틴으로 호출할 수 있습니다. 다음은 단어를 처음 8자로 줄이도록 SUBSTR을 함수로 호출하는 방법입니다.

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

SUBSTR을 서브루틴으로 호출하면 동일한 결과를 얻습니다.

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

서브루틴 또는 함수 작성 여부를 결정할 때 다음과 같은 질문을 스스로에게 하십시오.

- 리턴 값은 선택적입니까? 그렇다면, 서브루틴을 작성하십시오.
- 명령어 내에서 표현식으로 리턴된 값이 필요합니까? 그렇다면, 함수를 작성하십시오.

### 서브루틴 및 함수: 유사성과 차이점

서브루틴과 함수에는 다음 유사성이 있습니다.

- 내부 또는 외부일 수 있습니다.
  - 내부
    - 공통 변수를 사용하여 정보를 전달할 수 있습니다.
    - PROCEDURE 명령어를 사용하여 변수를 보호할 수 있음
    - 인수를 사용하여 정보를 전달할 수 있습니다.
  - 외부
    - 인수를 사용하여 정보를 전달해야 함
    - ARG 명령어 또는 ARG 기본 제공 함수를 사용하여 인수를 수신할 수 있습니다.
- 호출자로 돌아가려면 RETURN 명령어를 사용합니다.

표 3. 서브루틴과 함수 간의 차이		
	서브루틴	함수
호출	CALL 명령어, 서브루틴 이름 및 선택적으로 최대 20개의 인수를 사용하여 호출합니다.	함수 이름을 지정하고 바로 뒤에 선택적으로 최대 20개의 인수를 포함하는 소괄호를 지정하여 호출합니다.
값 리턴	값을 호출자로 리턴할 수 있습니다. RETURN 명령어에 값을 포함하면 언어 프로세서는 이 값을 REXX 특수 변수 RESULT로 지정합니다.	값을 리턴해야 합니다. RETURN 명령어에 값을 지정하십시오. 언어 프로세서는 함수 호출을 이 값으로 대체합니다.

### 서브루틴 및 함수 작성

서브루틴은 프로그램이 호출하는 일련의 명령어입니다. 특정 태스크를 수행합니다. 서브루틴을 호출하는 명령어는 CALL 명령어입니다. 동일한 서브루틴을 호출하기 위해 프로그램에서 여러 번 CALL 명령어를 사용할 수 있습니다.

서브루틴이 종료되면 서브루틴 호출 바로 뒤에 오는 명령어로 제어를 리턴할 수 있습니다. 제어를 리턴하는 명령어는 RETURN 명령어입니다.

명령어

CALL sub1

명령어

EXIT

sub1:

명령어

RETURN

함수는 프로그램이 특정 태스크를 수행하고 값을 리턴하기 위해 호출하는 일련의 명령어입니다. [53 페이지의 『제 5 장 함수』](#)에서 설명한 대로 함수는 내장되거나 사용자 작성될 수 있습니다. 내장 함수와 같은 방식으로 사

용자 작성 함수를 호출하십시오. 인수를 포함할 수 있는 소괄호가 바로 뒤에 오는 함수 이름을 지정하십시오. 함수 이름과 왼쪽 괄호 사이에 공백을 넣을 수 없습니다. 소괄호는 최대 20개의 인수를 포함하거나 인수를 전혀 포함하지 않을 수 있습니다.

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

또는

```
function()
```

함수 호출은 일반적으로 표현식에 표시되므로 함수에는 리턴 값이 필요합니다.

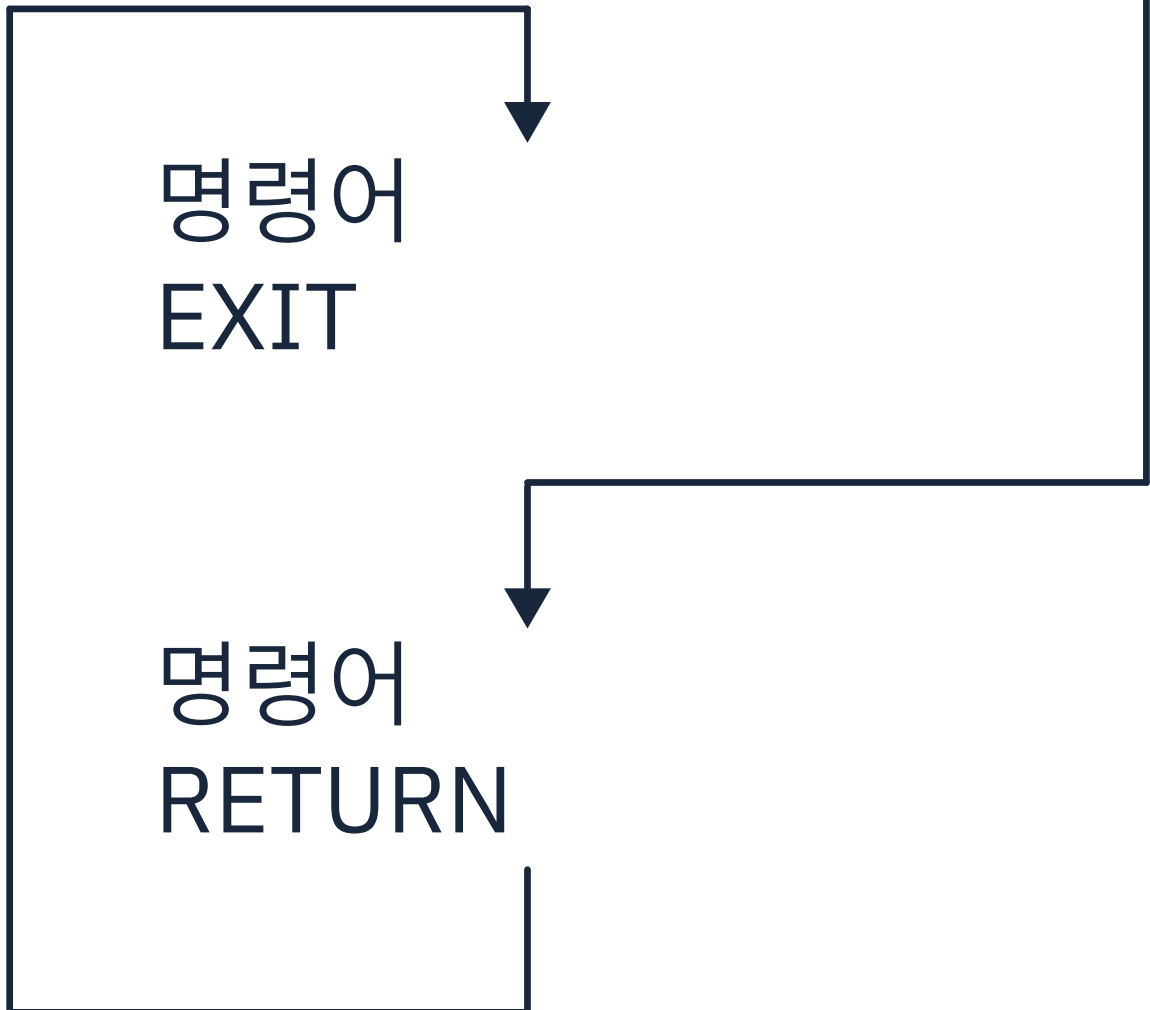
```
z = function(arguments1, argument2,...)
```

함수가 종료되면 RETURN 명령어를 사용하여 함수 호출을 대체할 값을 다시 전송할 수 있습니다.

명령어



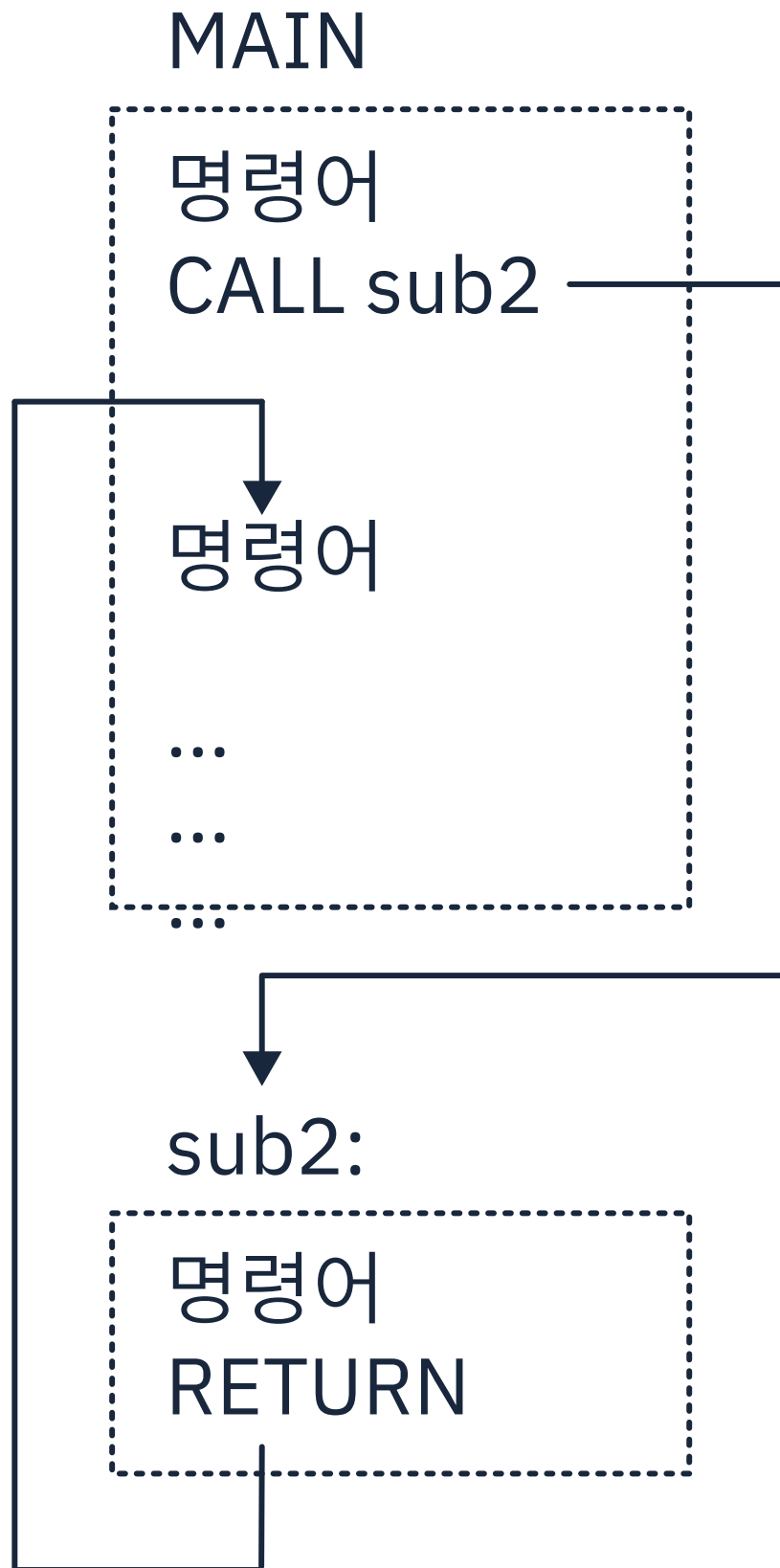
z=func1(arg1, arg2)



서브루틴 및 함수 모두 내부(레이블로 지정) 또는 외부일 수 있음 (서브루틴 또는 함수를 포함하는 REXX 파일 시스템 또는 파티션된 데이터 세트 멤버 이름으로 이를 지정합니다). 앞의 두 예제는 `sub1`이라는 내부 서브루틴과 `func1`이라는 내부 함수를 보여줍니다.

**중요사항:** 내부 서브루틴 및 함수는 일반적으로 프로그램의 주요 파트 다음에 표시되므로 내부 서브루틴 또는 함수가 있는 경우 EXIT 명령으로 프로그램의 주요 파트를 종료하는 것이 중요합니다.

다음은 `sub2`로 이름 지정된 외부 서브루틴을 설명합니다.



다음은 func2로 이름 지정된 외부 함수를 설명합니다.



## 내부 또는 외부 서브루틴 또는 함수 사용 선택

서브루틴 또는 함수를 내부 또는 외부로 작성할지 여부를 결정하기 위해 다음과 같은 요소를 고려할 수 있습니다.

- 서브루틴 또는 함수의 크기. 매우 큰 서브루틴 및 함수는 종종 외부이며 작은 서브루틴 및 함수는 호출 프로그램 내에 쉽게 맞습니다.
- 정보 전달 방법. 내부 서브루틴 또는 함수의 변수를 통해 정보를 빠르게 전달할 수 있습니다. 다음 주제에서는 이 방식으로 정보를 전달하는 방법에 대해 설명합니다.
- 서브루틴 또는 함수가 둘 이상의 프로그램 또는 사용자에게 가치가 있는지 여부. 그렇다면, 외부 서브루틴 또는 함수가 바람직합니다.
- 성능. 함수의 경우, 언어 프로세서는 외부 함수를 검색하기 전에 내부 함수를 검색합니다. 함수의 전체 검색 순서는 함수 및 서브루틴을 참조하십시오.

## 정보 전달

프로그램과 해당 내부 서브루틴 또는 함수는 동일한 변수를 공유할 수 있습니다. 따라서 공통 공유 변수를 사용하여 호출자와 내부 서브루틴 또는 함수 간에 정보를 전달할 수 있습니다. 인수를 사용하여 내부 서브루틴 또는 내부 함수와 정보를 주고 받을 수도 있습니다.

그러나 외부 서브루틴은 변수를 호출자와 공유할 수 없습니다. 정보를 전달하려면 인수 또는 데이터 스택과 같은 다른 외부 방법을 사용해야 합니다. (기억하십시오. 내부 함수는 함수 호출 뒤에 오는 소괄호 내의 인수는 전달할 필요가 없습니다. 그러나 모든 함수, 내부 및 외부 둘 다 값을 리턴해야 합니다.)

### 변수의 정보 전달

프로그램과 해당 내부 서브루틴 또는 함수가 동일한 변수를 공유할 때 변수 값은 마지막으로 지정된 값입니다. 이는 해당 지정이 프로그램의 주요 파트에 있었는지 또는 서브루틴이나 함수에 있었는지 여부와 관계없이 적용됩니다.

다음 예는 서브루틴에 정보를 전달하는 것을 보여줍니다. 변수 `number1`, `number2` 및 `answer`는 공유됩니다. `answer`의 값은 서브루틴에 지정되어 있고 프로그램의 주요 파트에서 사용됩니다.

```
/* ***** 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
```

그림 26. 서브루틴을 사용하여 변수의 정보를 전달하는 예제

다음 예제는 서브루틴이 아닌 함수에 정보를 전달한다는 점을 제외하고 동일합니다. 서브루틴은 `RETURN` 명령어에 변수 `answer`를 포함합니다. 언어 프로세서는 함수 호출을 `answer`의 값으로 대체합니다.

```
/* ***** 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
```

그림 27. 함수를 사용하여 변수의 정보를 전달하는 예제



프로그램과 해당 내부 서브루틴 또는 함수에서 동일한 변수를 사용하면 때때로 문제가 발생할 수 있습니다. 다음 예에서는 프로그램의 주요 파트와 서브루틴에서 해당 DO 루프에 대해 동일한 제어 변수 i를 사용합니다. 결과적으로 서브루틴이 i = 6을 사용하는 기본 프로그램으로 돌아가므로 DO 루프는 기본 프로그램에서 한 번만 실행됩니다.

```

/***** 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

```

그림 28. 서브루틴을 사용하여 변수의 정보를 전달하여 발생하는 문제점 예제

다음 예제는 서브루틴 대신 함수를 사용하여 정보를 전달한다는 점을 제외하고 동일합니다.

```

/***** 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

```

그림 29. 함수를 사용하여 변수의 정보를 전달하여 발생하는 문제점 예제

내부 서브루틴 또는 함수에서 이러한 유형의 문제점을 방지하려면 다음을 사용할 수 있습니다.

- 다음 절에 설명된 대로의 PROCEDURE 명령어.
- 서브루틴 또는 함수에서 프로그램의 주요 파트와 다른 변수 이름. 서브루틴의 경우, 인수를 CALL 명령어에 전달할 수 있습니다. 69 페이지의 『인수로 정보 전달』의 내용을 참조하십시오.

## PROCEDURE 명령어 내 변수 보호

서브루틴 또는 기능 바로 뒤에 PROCEDURE 명령어를 사용하는 경우 레이블, 서브루틴 또는 함수의 모든 변수는 서브루틴 또는 함수에 로컬이 되며, 프로그램의 주요 파트에서 실흔됩니다. PROCEDURE EXPOSE 명령어를 사용하여 지정된 몇 개의 변수를 제외한 모든 변수를 보호할 수 있습니다.

다음 예는 서브루틴 또는 함수가 PROCEDURE를 사용하거나 사용하지 않을 때 결과가 어떻게 다른지 보여줍니다.

```

/***** 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

```

그림 30. *PROCEDURE* 명령어를 사용한 서브루틴의 예

```

/***** 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

```

그림 31. *PROCEDURE* 명령어가 없는 서브루틴의 예

다음 두 개의 예는 서브루틴이 아닌 함수를 사용한다는 점을 제외하고 동일합니다.

```

/***** 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

```

그림 32. *PROCEDURE* 명령어를 사용한 함수의 예

```

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

```

그림 33. *PROCEDURE* 명령어가 없는 함수의 예

### PROCEDURE EXPOSE로 변수 공개

특정 변수를 제외한 모든 변수를 보호하려면 EXPOSE 옵션을 PROCEDURE 명령어와 함께 사용하고 그 뒤에 서브루틴 또는 함수에 공개된 변수를 사용하십시오.

다음 예는 서브루틴에서 PROCEDURE EXPOSE를 사용합니다.

```

/***** 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. */
/*****
number1 = 10
CALL subroutine
SAY number1 number2          /* produces 7 NUMBER2 */
EXIT
subroutine: PROCEDURE EXPOSE number1
number1 = 7
number2 = 5
RETURN

```

그림 34. 서브루틴에서 *PROCEDURE EXPOSE*를 사용하는 예

다음 예제는 *PROCEDURE EXPOSE*가 서브루틴 대신 함수에 있다는 점을 제외하고 동일합니다.

```

/***** REXX *****/
/* This program uses a PROCEDURE instruction with the EXPOSE option */
/* to expose one variable, number1, in its function. */
/*****
number1 = 10
SAY pass() number1          /* Produces 5 7 */
EXIT
pass: PROCEDURE EXPOSE number1
number1 = 7
number2 = 5
RETURN number2

```

그림 35. 함수에서 *PROCEDURE EXPOSE*를 사용하는 예

*PROCEDURE* 명령어에 대한 자세한 정보는 *PROCEDURE*의 내용을 참조하십시오.

## 인수로 정보 전달

내부 또는 외부 서브루틴, 또는 함수에 정보를 전달하는 방법은 인수를 사용하는 것입니다. 서브루틴을 호출할 때 다음과 같이 *CALL* 명령어에 쉼표로 구분된 최대 20개의 인수를 전달할 수 있습니다.

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

함수 호출에서 최대 20개의 인수를 쉼표로 구분하여 전달할 수 있습니다.

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

## ARG 명령어 사용

서브루틴 또는 함수는 *ARG* 명령어로 인수를 수신할 수 있습니다. *ARG* 명령어에서 쉼표도 인수를 구분합니다.

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

전달된 인수의 이름은 정보가 인수 이름이 아닌 위치에 의해 전달되므로 *ARG* 명령어의 이름과 같을 필요는 없습니다. 전송된 첫 번째 인수가 첫 번째 수신된 인수입니다. *CALL* 명령어 또는 함수 호출에 템플리트를 설정할 수도 있습니다. 그러면 언어 프로세서가 해당하는 *ARG* 명령어에서 이 템플리트를 사용합니다. 템플리트를 사용한 구문 분석에 대한 정보는 77 페이지의 『데이터 구문 분석』의 내용을 참조하십시오.

다음 예제에서 메인 루틴은 직사각형의 둘레를 계산하는 서브루틴으로 정보를 보냅니다. 서브루틴은 *RETURN* 명령어에 값을 지정하여 변수 *perim*의 값을 리턴합니다. 주 프로그램은 특수 변수 *RESULT*의 값을 수신합니다.

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

```
perimeter:  
  ARG length, width  
   $\text{perim} = 2 * \text{length} + 2 * \text{width}$   
  RETURN perim
```

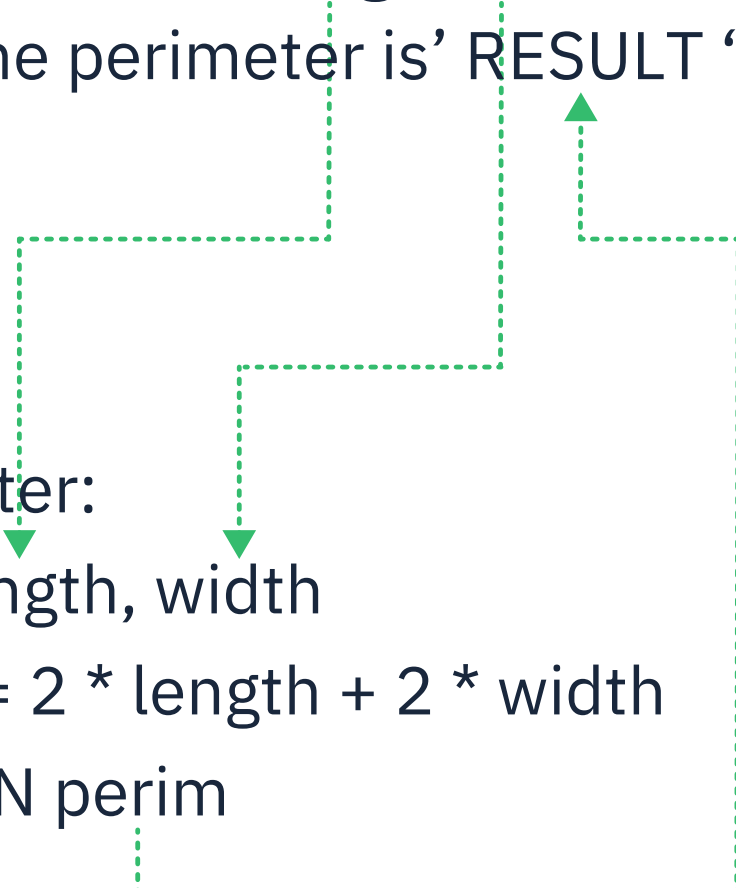


그림 36. CALL 명령어에서 인수를 전달하는 예

다음 예제는 서브루틴 대신 함수에서 ARG를 사용한다는 점을 제외하고 동일합니다.

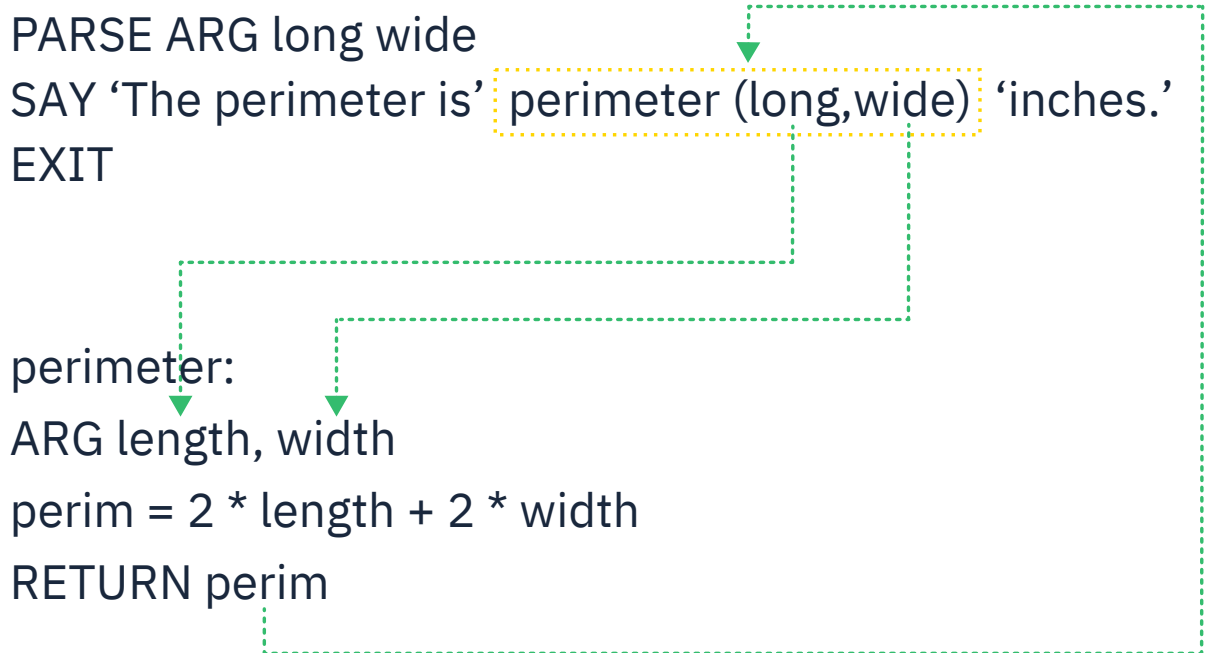


그림 37. 내부 루틴에 대한 호출 시 인수 전달 예제

두 가지 선행 예에서 long과 length 및 wide와 width 사이의 위치 관계에 유의하십시오. 또한 정보가 변수 perim에서 어떻게 수신되는지 확인하십시오. 두 프로그램 모두 RETURN 명령어에 perim이 포함되어 있습니다. 서브루틴이 있는 프로그램의 경우 언어 프로세서는 perim의 값을 특수 변수 RESULT로 지정합니다. 함수를 사용하는 프로그램의 경우, 언어 프로세서는 함수 호출 perimeter(long,wide)를 perim의 값으로 대체합니다.

### ARG 기본 제공 함수 사용

서브루틴 또는 함수가 인수를 수신하는 다른 방법은 ARG 기본 제공 함수를 사용하는 것입니다. 이 함수는 특정 인수의 값을 리턴합니다. 숫자는 인수 위치를 표시합니다.

예를 들어, 앞의 예에서

```
ARG length, width
```

ARG 명령어 대신 다음과 같이 ARG 함수를 사용할 수 있습니다.

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

ARG 함수에 대한 자세한 정보는 [ARG](#)의 내용을 참조하십시오.

### 서브루틴 또는 함수에서 정보 수신

서브루틴 또는 함수는 최대 20개의 인수를 수신할 수 있지만 RETURN 명령어에서는 하나의 표현식만 지정할 수 있습니다.

해당 표현식은 다음과 같을 수 있습니다.

- 숫자

```
RETURN 55
```

- 값이 대체된 하나 이상의 변수(또는 값이 지정되지 않은 경우의 이름).

```
RETURN value1 value2 value3
```

- 리터럴 문자열

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

- 값이 대체되는 산술, 비교 또는 논리식.

```
RETURN 5 * number
```

### 연습: 내부 및 외부 서브루틴 작성

모의 코인 던지기 게임을 하고 누적된 점수를 생성하는 프로그램을 작성하십시오.

네 개의 가능한 입력이 있어야 합니다.

- 'HEADS'
- 'TAILS'
- " (널 - 게임 중지)
- 이 세 항목이 아님(올바르지 않은 응답).

유효한 입력을 확인하도록 인수없이 내부 서브루틴을 작성하십시오. 임의의 결과를 생성하도록 RANDOM 기본 제공 함수를 사용하는 외부 서브루틴에 유효한 입력을 전송하십시오. HEADS = 0과 TAILS = 1을 가정하고, 다음과 같이 RANDOM을 사용하십시오.

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

유효한 입력을 RANDOM의 값과 비교하십시오. 동일한 경우, 사용자는 1점을 획득합니다. 다른 경우, 컴퓨터가 1점을 획득합니다. 결과가 계산되는 주 프로그램으로 결과를 리턴하십시오.

## ANSWER

```

/***** 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. */
/*****
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

```

그림 38. 가능한 솔루션(주 프로그램)

```

/***** 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 */
/*****
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

```

그림 39. 가능한 솔루션(이름이 CHECK인 내부 서브루틴)

```

/***** 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

```

그림 40. 가능한 솔루션(이름이 *THROW*인 외부 서브루틴)

### 연습: 함수 작성

공백으로 구분된 숫자 목록을 수신하여 평균을 계산하는 *AVG*라는 함수를 작성합니다. 최종 응답은 10진수일 수 있습니다. 이 함수를 호출하려면 다음을 사용할 수 있습니다.

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

*WORDS*(*WORDS* 함수 참조) 및 *WORD*(*WORD* 함수 참조) 기본 제공 함수를 사용하십시오.

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

```

그림 41. 가능한 솔루션

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ €. [CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.](#)



## 제 6 장 데이터 조작

이 절에서는 복합 변수 및 스템을 사용하여 데이터에 액세스하는 방법과 구문 분석을 사용하는 방법에 대해 설명합니다.

### 복합 변수 및 스템 사용

때로는 데이터를 쉽게 검색할 수 있는 방법으로 관련 데이터 그룹을 저장하는 것이 유용합니다. 예를 들어, 직원 이름 목록을 배열로 저장하고 번호로 검색할 수 있습니다. 배열은 단일 이름으로 식별되는 하나 이상의 차원의 요소 배열입니다. `employee`라는 배열에는 다음과 같은 이름이 포함될 수 있습니다.

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

일부 컴퓨터 언어에서는 요소의 수를 사용하여 배열의 요소에 액세스합니다. 예를 들어, `employee(1)`은 Adams, Joe를 검색합니다. REXX에서는 복합 변수를 사용합니다.

### 복합 변수의 개념

복합 변수를 사용하여 REXX에서 배열 또는 변수의 목록을 작성할 수 있습니다. 복합 변수(예: `employee.1`)는 스템과 후미로 구성됩니다.

스템은 끝에 마침표가 있는 기호입니다. 스템의 일부 예는 다음과 같습니다.

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

후미는 아래첨자와 유사합니다. 스템을 따르며 상수 기호(`employee.1`에서와 같은), 단순 기호(`employee.n`) 또는 널일 수 있는 이름의 추가 파트로 구성됩니다. 따라서 REXX에서 아래첨자는 반드시 숫자일 필요가 없습니다. 복합 변수에는 양쪽에 문자가 있는 마침표가 하나 이상 있습니다. 복합 변수의 예는 다음과 같습니다.

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

스템의 이름은 대체할 수 없지만 후미는 대체할 수 있습니다. 예를 들어, 다음과 같습니다.

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

다른 REXX 변수와 마찬가지로 이전에 후미의 변수에 값을 지정하지 않은 경우 해당 고유 이름의 값이 대문자로 사용됩니다.

```
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 */
```

복합 변수의 그룹을 초기화하고 배열을 설정하기 위해 DO 루프를 사용할 수 있습니다.

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

직원 배열의 예에서 사용된 동일한 이름을 사용하면 다음과 같은 복합 변수 그룹이 있습니다.

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

이름이 복합 변수 그룹에 있으면 해당 번호 또는 해당 번호를 표시하는 변수로 이름에 쉽게 액세스할 수 있습니다.

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

복합 변수에 대한 자세한 정보는 [복합 기호](#)의 내용을 참조하십시오.

## 스텝 사용

복합 변수에 대한 작업을 수행할 때 전체 변수 컬렉션을 동일한 값으로 초기화하는 것이 유용한 경우가 많습니다. 스텝이 포함된 지정을 사용하면 이 작업을 쉽게 수행할 수 있습니다. 예를 들어, `number.=0`은 `number.`라는 배열의 모든 배열 요소를 0으로 초기화합니다.

배열의 모든 복합 변수 값을 같은 방식으로 변경할 수 있습니다. 예를 들어, 모든 직원 이름을 `Nobody`로 변경하려면 다음 지정 명령어를 사용하십시오.

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

결과적으로 스텝 `employee.`으로 시작하는 모든 복합 변수(이전에 지정되지 않거나 지정되지 않음) 값은 `Nobody`입니다. 스텝 지정 후 개별 복합 변수에 새 값을 지정할 수 있습니다.

```
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' */
```

파일을 읽고 쓸 때 EXECIO 및 RFS 명령과 함께 스텝을 사용할 수 있습니다. [EXECIO](#) 및 [RFS](#)를 참조하십시오. RFS는 CICS에서 선호되는 I/O 메소드입니다.

## 연습 - 복합 변수 및 스텝 사용

1. 이러한 지정 명령어 뒤에 다음 SAY 명령어는 무엇을 생성합니까?

```
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. 이러한 지정 명령어 뒤에 SAY 명령어가 생성하는 결과는 무엇입니까?

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

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

#### ANSWERS

- 1. a. 3
- b. 4
- c. last
- d. A.3
- e. 2
- f. D.last
- g. C.3
- h. A.FIRST
- i. cv3d
- 2. a. empty
- b. 전체
- c. empty

## 데이터 구문 분석

구문 분석은 데이터를 분리하고 데이터의 일부를 하나 이상의 변수로 지정합니다. 구문 분석은 데이터의 각 단어를 변수로 지정하거나 데이터를 더 작은 파트로 나눌 수 있습니다. 구문 분석은 데이터를 열로 형식화하는 데에도 유용합니다.

데이터를 수신할 변수는 템플릿에서 이름 지정됩니다. 템플릿은 데이터 분할 방법을 알려주는 모델입니다. 데이터를 수신하는 변수 목록만큼 간단할 수 있습니다. 보다 복잡한 템플릿에는 패턴이 포함될 수 있습니다. 79 페이지의 [『패턴으로 구문 분석』](#)의 내용을 참조하십시오.

## 구문 분석 명령어

REXX 구문 분석 명령어는 PULL, ARG 및 PARSE입니다. (PARSE에는 여러 가지 변형이 있습니다.)

### PULL 명령어

PULL은 입력을 읽고 하나 이상의 변수에 지정하는 명령어입니다. 프로그램 스택에 정보가 포함되는 경우 PULL 명령어는 프로그램 스택에서 정보를 가져옵니다. 프로그램 스택이 비어 있으면 PULL은 현재 터미널 입력 디바이스에서 정보를 가져옵니다. 데이터 스택에 대한 정보는 14 페이지의 [『프로그램 스택 또는 터미널 입력 디바이스에서 정보 가져오기』](#)의 내용을 참조하십시오.

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

PULL은 문자 정보를 변수에 지정하기 전에 대문자로 변환합니다. 대문자 변환을 원하지 않는 경우 PARSE PULL 명령어를 사용하십시오.

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

```
/* word2 contains 'is' */
/* word3 contains 'power.' */
```

PARSE 명령어의 변형에 선택적 키워드 UPPER를 포함할 수 있습니다. 이는 언어 프로세서가 문자 정보를 변수에 지정하기 전에 대문자로 변환하도록 합니다. 예를 들어, PARSE UPPER PULL을 사용하면 PULL 사용과 동일한 결과를 제공합니다.

## ARG 명령어

ARG 명령어는 프로그램, 함수 또는 서브루틴에 인수로 전달된 정보를 가져와서 하나 이상의 변수에 넣습니다. 세 개의 인수 Knowledge is power.를 이름이 sample인 REXX 프로그램에 전달하려면 다음을 수행하십시오.

1. 프로그램을 호출하고 인수를 exec 이름 뒤에 문자열로 전달하십시오.

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

2. ARG 명령어를 사용하여 세 개의 인수를 변수로 받으십시오.

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

ARG는 인수를 변수에 지정하기 전에 문자 정보를 대문자로 변환합니다.

대문자 변환을 원하지 않는 경우 ARG 대신 PARSE ARG 명령어를 사용하십시오.

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

PARSE UPPER ARG에는 ARG와 동일한 결과가 있습니다. 문자 정보를 변수로 지정하기 전에 대문자로 변환합니다.

## PARSE VALUE ... WITH 명령어

PARSE VALUE...WITH 명령어는 리터럴 문자열과 같은 지정된 표현식을 이름이 WITH 서브키워드 뒤에 오는 하나 이상의 변수로 구문 분석합니다.

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

PARSE VALUE는 문자 정보를 변수로 지정하기 전에 대문자로 변환하지 않습니다. 대문자 변환을 원하면, PARSE UPPER VALUE를 사용하십시오. PARSE VALUE에서 문자열 대신 변수를 사용할 수 있습니다(먼저 변수에 값을 지정함).

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

또는 PARSE VAR을 사용하여 변수를 구문 분석할 수 있습니다.

## PARSE VAR 명령어

PARSE VAR 명령어는 지정된 변수를 하나 이상의 변수로 구문 분석합니다.

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

```
/* word2 contains 'is' */
/* word3 contains 'power.' */
```

PARSE VAR은 문자 정보를 변수로 지정하기 전에 대문자로 변환하지 않습니다. 대문자 변환을 원하면, PARSE UPPER VAR을 사용하십시오.

## 단어로 구문 분석에 대한 추가 정보

구문 분석할 데이터의 단어 수가 템플리트의 변수 수와 동일하지 않은 경우의 작동에 대해 설명합니다.

이전 주제의 예에서 구문 분석할 데이터의 단어 수는 항상 템플리트의 변수 수와 같습니다. 구문 분석은 항상 템플리트에서 이름이 지정된 모든 변수에 새 값을 지정합니다. 구문 분석할 데이터에 단어보다 많은 변수 이름이 있는 경우 나머지 변수는 널(비어 있음) 값을 수신합니다. 데이터에 템플리트의 변수 이름보다 구문 분석할 단어가 더 많은 경우 각 변수는 마지막 변수를 제외한 한 단어의 데이터를 순서대로 가져옵니다. 마지막 변수는 데이터의 나머지를 가져옵니다.

다음 예에서는 데이터의 단어보다 템플리트의 변수 이름이 더 많습니다. 나머지 변수는 널 값을 수신합니다.

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

다음 예에서는 템플리트의 변수 이름보다 데이터의 단어가 더 많습니다. 마지막 변수는 데이터의 나머지를 가져옵니다. 마지막 변수 이름에는 여러 단어와 선행 공백과 후행 공백이 포함될 수 있습니다.

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

단어로 구문 분석하면 일반적으로 각 단어에서 선행 공백 및 후행 공백을 제거하여 변수에 넣습니다. 그러나 데이터를 마지막 변수에 넣을 때 구문 분석은 하나의 단어 분리자 공백을 제거하지만 추가 선행 또는 후행 공백은 유지합니다. words 전에 2개의 선행 공백이 있습니다. 구문 분석은 'words'를 var2에 넣기 전에 단어 분리자 공백과 추가 선행 공백을 모두 제거합니다. in 전에 네 개의 선행 공백이 있습니다. var3이 마지막 변수이므로 구문 분석은 단어 분리자 공백을 제거하지만 추가 선행 공백은 유지합니다. 그러므로, var3은 ' in data'을 수신합니다(3개의 선행 공백과 함께).

템플리트에서 마침표는 플레이스홀더의 역할을 합니다. 데이터를 수신하지 않습니다. 변수 그룹 내에서 또는 템플리트의 끝에서 마침표를 "더미 변수"로 사용하여 원치 않는 정보를 수집할 수 있습니다.

```
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' */
```

구문 분석 명령어에 대한 자세한 정보는 [PARSE](#)를 참조하십시오.

## 패턴으로 구문 분석

가장 간단한 템플리트는 공백으로 구분된 변수 이름 그룹입니다. 데이터를 공백으로 구분된 단어로 구문 분석합니다. 템플리트는 패턴을 포함할 수도 있습니다. 패턴은 문자열, 숫자 또는 이들 중 하나를 표시하는 변수일 수 있습니다.

### 문자열

템플리트에서 문자열을 사용하는 경우 구문 분석은 입력 데이터에서 일치하는 문자열을 확인합니다. 데이터를 변수에 지정할 때 구문 분석은 일반적으로 템플리트의 문자열과 일치하는 입력 문자열 파트를 건너뜁니다.

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

이 예에서 쉼표는 문자열 구분 기호이므로 'To be'에는 쉼표가 포함되지 않습니다. (또한 part2에는 공백으로 시작하는 값이 포함되어 있습니다. 구문 분석은 입력 문자열을 일치하는 텍스트로 분할합니다. 한 변수에 일치 시작까지의 데이터를, 다음 변수에 일치 후 데이터를 넣습니다.

## 변수

템플릿에서 구분 기호로 지정할 문자열을 미리 모르는 경우 괄호로 묶인 변수를 사용할 수 있습니다.

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

다시, 이 예에서 쉼표는 문자열 구분 기호이므로 'To be'에는 쉼표가 포함되지 않습니다.

## 숫자

템플릿에서 숫자를 사용하여 데이터를 분리할 열을 표시할 수 있습니다. 부호 없는 정수는 절대 열 위치를 표시합니다. 부호 있는 정수는 상대 열 위치를 표시합니다.

부호 없는 정수 또는 등호(=) 접두부가 있는 정수는 절대 열 위치에 따라 데이터를 분리합니다. 첫 번째 세그먼트는 열 1에서 시작하여 지정된 열 번호의 정보로 이동하지만 포함되지는 않습니다. 후속 세그먼트는 지정된 열 번호에서 시작합니다.

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

다음 코드는 동일한 결과를 갖습니다.

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

숫자 패턴 1의 지정은 선택적입니다. 숫자 패턴을 사용하여 구문 분석의 시작점을 표시하지 않으면 기본값은 1입니다. 이 예제에서는 숫자 패턴 5가 =5와 동일함을 보여줍니다.

템플릿에 여러 개의 숫자 패턴이 있고 이후의 패턴이 이전 패턴보다 낮은 경우 구문 분석은 더 낮은 숫자가 지정하는 열로 루프를 반복합니다.

```
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.' */
```

템플릿의 각 변수에 그 앞뒤에 열 번호가 있는 경우 두 숫자는 변수 데이터의 시작과 끝을 표시합니다.

```
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.' */
```

따라서 숫자 패턴을 사용하여 데이터의 일부를 건너뛸 수 있습니다.

```
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. */

```

템플릿에서 부호 있는 정수는 상대 열 위치에 따라 데이터를 분리합니다. 더하기 또는 빼기 기호는 시작 위치에서 각각 오른쪽 또는 왼쪽으로 이동함을 표시합니다. 다음 예에서는 part1이 열 1에서 시작함을 기억하십시오 (기본적으로 시작점을 표시하는 숫자가 없기 때문에).

```

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 part2는 열 6(1+5=6)에서 시작하여 part2 데이터에 넣는 구문 분석을 의미합니다. +5 part3은 part3에 넣는 데이터가 열 11(6+5=11)로 시작하는 등을 의미합니다. 빼기 기호 사용은 더하기 기호 사용과 유사합니다. 데이터 문자열에서 상대 위치를 식별합니다. 빼기 기호는 데이터 문자열에 "백업됩니다"(왼쪽으로 이동).

```

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.' */

```

이 예에서 part1은 열 1부터(기본적으로) 문자를 수신합니다. +10 part2는 열 11(1+10=11)에서 시작하는 문자를 수신합니다. +3 part3는 열 14(11+3=14)에서 시작하는 문자를 수신합니다. -3 part4는 열 11(14-3=11)에서 시작하는 문자를 수신합니다.

유연성을 높이기 위해 구문 분석 명령어에서 변수 숫자 패턴을 정의하고 사용할 수 있습니다. 이를 수행하려면 먼저 구문 분석 명령 전에 변수를 부호 없는 정수로 정의하십시오. 그런 다음 구문 분석 명령어에서 변수를 괄호로 묶고 왼쪽 괄호 앞에 다음 중 하나를 지정하십시오.

- 오른쪽으로의 열을 표시하는 더하기 부호(+)
- 왼쪽으로의 열을 표시하는 빼기 부호(-)
- 절대 열 위치를 표시하는 등호(=).

(왼쪽 괄호 앞에 +, - 또는 =가 없으면 언어 프로세서는 변수를 문자열 패턴으로 간주합니다.) 다음 예제는 변수 숫자 패턴 movex를 사용합니다.

```

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.' */

```

구문 분석에 대한 자세한 정보는 [구문 분석](#)의 내용을 참조하십시오.

## 인수로서 여러 문자열 구문 분석

인수를 함수 또는 서브루틴에 전달할 때 구문 분석될 다중 문자열을 지정할 수 있습니다. ARG, PARSE ARG 및 PARSE UPPER ARG 명령어는 인수를 구문 분석합니다. 다중 문자열에서 작동하는 유일한 구문 분석 명령어입니다.

다중 문자열을 전달하려면 쉼표를 사용하여 인접 문자열을 구분하십시오.

다음 예제는 세 개의 인수를 내부 서브루틴에 전달합니다.

```
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' */
```

첫 번째 인수는 두 개의 단어 String One이며 세 개의 변수 이름, word1, word2 및 word3 으로 구문 분석됩니다. 세 번째 단어가 없으므로 세 번째 변수, word3이 널로 설정됩니다. 두 번째 및 세 번째 인수는 변수 이름 string2 및 string3으로 완전히 구문 분석됩니다.

프로그램 또는 서브루틴에 전달된 다중 인수 분석에 대한 자세한 정보는 [여러 문자열 구문 분석](#)의 내용을 참조하십시오.

## 연습 - 구문 분석 연습

다음 구문 분석 예의 결과는 무엇입니까?

1. 

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

  - a) word1 =
  - b) word2 =
  - c) word3 =
2. 

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

  - a) word1 =
  - b) word2 =
  - c) word3 =
  - d) word4 =
  - e) word5 =
  - f) 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 =

## ANSWERS

- 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(v2에 'is '가 포함됩니다.)  
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

## 제 7 장 프로그램에서 명령 사용

이 절에서는 REXX 프로그램에서의 명령 사용에 대해 설명합니다.

명령의 기본 카테고리는 다음과 같습니다.

### REXX/CICS 명령

이 명령은 기타 REXX/CICS 기능에 대한 액세스를 제공합니다. [REXX/CICS 명령의 내용](#)을 참조하십시오.

### CICS 명령

이 명령은 CICS 서비스에 액세스하기 위해 애플리케이션 프로그램이 사용하는 EXEC CICS 명령을 구현합니다. [CICS 명령 요약](#)의 내용을 참조하십시오.

제한된 명령과 키워드가 CICS TS5.5 이상에 정의된 경우 주의하십시오. 이 제한사항은 REXX 프로그램에서 호출된 EXEC CICS 명령에 적용되지 않습니다. 세부사항은 [특정 CICS API 및 SPI 명령 사용 제어](#)의 내용을 참조하십시오.

### SQL문

이러한 명령문은 동적으로 준비되고 실행됩니다. [REXX/CICS DB2 인터페이스](#)의 내용을 참조하십시오.

### EDIT 명령

이 명령은 REXX/CICS 매크로에서 편집기 기능을 호출합니다. [REXX/CICS 텍스트 편집기](#)의 내용을 참조하십시오.

### RFS 명령

이 명령은 RFS(REXX File System)용입니다. [REXX/CICS 파일 시스템](#)의 내용을 참조하십시오.

### RLS 명령

이 명령은 RLS(REXX List System)용입니다. [REXX/CICS 목록 시스템](#)의 내용을 참조하십시오.

프로그램에서 명령을 실행하면 REXX 특수 변수 RC가 리턴 코드로 설정됩니다. 프로그램은 리턴 코드를 사용하여 프로그램 내의 조치 과정을 판별할 수 있습니다. 명령이 실행될 때마다 RC가 설정됩니다. 따라서 RC에는 가장 최근에 실행된 명령의 리턴 코드가 포함됩니다.

## 명령에서 따옴표 사용

일반적으로 명령을 다른 유형의 명령과 구별하려면 명령을 작은따옴표 또는 큰따옴표로 묶어야 합니다. 명령이 따옴표로 묶이지 않으면 표현식으로 처리되어 오류로 끝날 수 있습니다. 예를 들면, 언어 프로세서는 별표(\*)를 곱셈 연산자로 처리합니다.

많은 CICS 명령은 명령 내에 작은따옴표를 사용합니다. 이런 이유로, 당연히 CICS 명령을 큰따옴표 표시로 둘러싸는 것이 좋습니다.

다음 예제는 임시 저장영역 큐 ABC에 단어 test를 배치합니다.

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

## 명령에서 변수 사용

명령에 변수가 포함된 경우 변수가 따옴표 안에 있으면 변수 값이 대체되지 않습니다. 언어 프로세서는 따옴표 외부의 변수에만 변수 값을 사용합니다.

## 다른 REXX 프로그램을 명령으로 호출

다른 프로그램을 외부 루틴으로 호출하거나 EXEC 명령을 사용하여 명시적으로 다른 프로그램에서 프로그램을 호출할 수 있습니다.

58 페이지의 『서브루틴 및 함수』에서는 다른 프로그램을 외부 루틴으로 호출하는 방법에 대해 설명합니다. EXEC 명령을 사용하여 명시적으로 다른 프로그램에서 프로그램을 호출할 수도 있습니다. 외부 루틴과 같이 명시적으로 또는 내재적으로 호출된 프로그램은 RETURN 또는 EXIT 명령어로 호출자에 값을 리턴할 수 있습니다. 특수 변수 RESULT에 값을 전달하는 외부 루틴과 달리 호출되는 프로그램은 REXX 특수 변수 RC에 값을 전달합니다.

프로그램 내에서 다른 프로그램을 명시적으로 호출하려면 다른 REXX/CICS 명령과 마찬가지로 EXEC 명령을 사용하십시오. 호출된 프로그램은 RETURN 또는 EXIT 명령으로 끝나 제어는 호출자에게 리턴되도록 합니다. REXX 특수 변수 RC는 EXEC 명령의 리턴 코드로 설정됩니다. 선택적으로 RETURN 또는 EXIT 명령어로 값을 호출자에게 리턴할 수 있습니다. 제어가 호출자로 다시 전달되면 REXX 특수 변수 RC는 RETURN 또는 EXIT 명령에서 리턴된 표현식의 값으로 설정됩니다.

예를 들어, CALC라는 프로그램을 호출하고 4개의 인수로 전달하기 위해 다음 명령어를 포함할 수 있습니다.

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

CALC에는 다음 명령어가 포함될 수 있습니다.

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

## 프로그램에서 명령 실행

호스트 명령 환경의 개념, 명령이 호스트 명령 환경으로 전달되는 방법 및 호스트 명령 환경을 변경하는 방법을 알아봅니다.

### 이 태스크 정보

명령을 실행하기 위한 환경을 호스트 명령 환경이라고 합니다. 프로그램이 실행되기 전에 명령을 처리하도록 활성 호스트 명령 환경이 정의됩니다. 언어 프로세서는 명령을 발견하면 처리를 위해 명령을 호스트 명령 환경에 전달합니다.

REXX 프로그램이 호스트 시스템에서 실행될 때 명령을 실행하기 위해 사용 가능한 기본 환경이 하나 이상 있습니다.

호스트 명령 환경은 다음과 같습니다.

#### REXXCICS

기본 REXX/CICS 명령 환경입니다. 모든 REXX/CICS, SQL, EDIT, RFS 또는 RLS 명령은 이 환경에서 실행될 수 있습니다. 그러나 CICS 명령에는 CICS 명령, EXEC SQL 앞에는 SQL, EDITSVR 앞에는 EDIT 명령, RFS에는 RFS 명령, 그리고 RLS에는 RLS 명령어가 접두부로 지정되어야 합니다.

#### CICS

CICS 명령만 실행하는 선택적 환경입니다. 호스트 명령 문자열의 첫 번째 단어는 명령 이름입니다(예: SEND, RECEIVE).

#### EXEC SQL

CICS/Db2® 인터페이스에 대한 SQL문(SELECT)을 실행하는 선택적 환경입니다.

#### EDITSVR

편집 세션을 작성하는 선택적 환경입니다.

#### FLSTSVR

파일 목록 유틸리티에 해당하는 명령을 실행하는 선택적 환경입니다.

#### RFS

REXX 파일 시스템에 대한 명령을 실행하는 선택적 환경입니다.

#### RLS

REXX 목록 시스템에 대한 명령을 실행하는 선택적 환경입니다.

**참고:** REXXCICS의 기본 환경을 모든 명령에 사용하는 것이 좋습니다(즉, ADDRESS 명령어가 지정되지 않아야 함).

## 명령이 호스트 환경으로 어떻게 전달됩니까?

언어 프로세서는 REXX 프로그램에서 각 표현식을 평가합니다. 이 평가는 결과적으로 문자열이 됩니다(널 문자열일 수 있음). 문자열은 적절하게 준비되어 호스트 명령 환경에 제출됩니다. 환경은 문자열을 명령으로 처리하고 처리가 완료되면 언어 프로세서로 제어를 리턴합니다. 문자열이 현재 호스트 명령 환경에 유효한 명령이 아닌 경우 실패가 발생하고 특수 변수 RC에는 호스트 명령 환경의 리턴 코드가 포함됩니다.

## 호스트 명령 환경 변경

기본 환경(REXXCICS) 또는 이전에 설정된 환경에서 호스트 명령 환경을 변경할 수 있습니다.

호스트 명령 환경을 변경하려면 ADDRESS 명령과 환경 이름을 차례로 사용하십시오.

ADDRESS 명령어는 두 가지 양식이 있습니다. 하나는 단일 명령에만 영향을 미치고 다른 하나는 해당 명령어 뒤에 발행된 모든 명령에 영향을 미칩니다.

- 단일 명령어

ADDRESS 명령어에 호스트 명령 환경 이름과 명령이 모두 포함된 경우 해당 명령만 지정된 환경으로 전송됩니다. 명령이 완료된 후 이전 호스트 명령 환경은 다시 활성화됩니다.

- 전체 명령

ADDRESS 명령어에 호스트 명령 환경의 이름만 포함된 경우 해당 호스트 명령 환경 내에서 이후에 발행된 모든 명령은 해당 환경의 명령으로 처리됩니다.

이 ADDRESS 명령어는 명령어를 사용하는 프로그램의 호스트 명령 환경에만 영향을 줍니다. 프로그램이 외부 루틴을 호출하면 호출 프로그램의 호스트 명령 환경에 관계없이 호스트 명령 환경이 기본 환경입니다. 원래 프로그램으로 돌아가면 ADDRESS 명령어가 이전에 설정한 호스트 명령 환경이 재개됩니다.

## 활성 호스트 명령 환경 판별

현재 활성인 호스트 명령 환경을 찾으려면 ADDRESS 기본 제공 함수를 사용하십시오. 예:

```
curenv = ADDRESS()
```

이 예에서 curenv는 활성 호스트 명령 환경(예: REXXCICS)으로 설정됩니다.



## 제 8 장 프로그램에서 문제점 진단

프로그램에서 오류가 발생하면 몇 가지 방법으로 오류를 찾을 수 있습니다.

- TRACE 명령어는 언어 프로세서가 각 오퍼레이션을 어떻게 평가하는지 보여줍니다. TRACE 명령어를 사용하여 표현식을 평가하는 방법에 대한 자세한 정보는 89 페이지의 『TRACE 명령어로 표현식 추적』의 내용을 참조하십시오. TRACE 명령어를 사용하여 호스트 명령을 평가하는 방법에 대한 자세한 정보는 90 페이지의 『TRACE 명령어로 명령 추적』의 내용을 참조하십시오.
- REXX/CICS는 다음과 같이 특수 변수 RC 및 SIGL을 설정합니다.

### RC

명령의 리턴 코드를 표시합니다.

### SIGL

함수 호출, SIGNAL 명령어 또는 CALL 명령어로 인해 제어가 전송된 행 번호를 표시합니다.

## TRACE 명령어로 표현식 추적

TRACE 명령어를 사용하여 언어 프로세서가 표현식을 읽을 때 각 오퍼레이션을 어떻게 평가하는지 표시하거나 표현식의 최종 결과를 표시할 수 있습니다. 이러한 두 가지 유형의 추적은 프로그램 디버깅에 유용합니다.

### 추적 오퍼레이션

표현식 내에서 오퍼레이션을 추적하려면 TRACE 명령어의 TRACE I(TRACE Intermediates) 양식을 사용하십시오. 언어 프로세서는 명령어 뒤에 오는 모든 표현식을 분류하고 다음과 같이 분석합니다.

#### >V> 변수값

추적된 데이터는 변수의 콘텐츠입니다.

#### >L> 리터럴 값

추적된 데이터는 리터럴(문자열, 초기화되지 않은 변수 또는 상수)입니다.

#### >O> 오퍼레이션 결과

추적된 데이터는 두 항에 대한 오퍼레이션의 결과입니다.

다음 예제는 TRACE I 명령어를 사용합니다. (행 번호는 프로그램의 파트가 아닙니다. 이는 뒤에 오는 예제에 대한 설명을 용이하게 합니다.)

```
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 */
```

그림 42. TRACE는 REXX가 표현식을 어떻게 평가하는지 표시함

예제를 실행하면 SAY 명령어는 다음을 생성합니다.

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

9는 행 번호입니다. \*-\*는 다음이 프로그램, IF a + 1 < 5 \* y의 데이터임을 표시합니다. 나머지 행은 모든 표현식을 분류합니다.

## 결과 추적

표현식의 최종 결과만 추적하려면 TRACE 명령어의 TRACE R(TRACE Results) 양식을 사용하십시오. 언어 프로세서는 다음과 같이 명령어 뒤에 오는 모든 표현식을 분석합니다.

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

이전 예에서 TRACE 명령어 피연산자를 I에서 R로 변경한 경우 다음 결과가 표시됩니다.

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

추적 오퍼레이션 및 결과 외에도 TRACE 명령어는 다른 유형의 추적을 제공합니다. [TRACE](#)의 내용을 참조하십시오.

## 연습: TRACE 명령어 사용

다음과 같은 복합 표현식으로 프로그램을 작성하십시오.

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

프로그램에서 *a*, *z*, *c* 및 *d* 를 정의하고 TRACE I 명령어를 사용하십시오.

ANSWER

```
/***** 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.'
```

그림 43. 가능한 솔루션

이 프로그램을 실행하면 다음이 생성됩니다.

```
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 명령어로 명령 추적

TRACE 명령어에는 명령에 대한 C 및 오류에 대한 E를 포함하여 다양한 유형의 추적을 위한 많은 옵션이 있습니다.



## TRACE C

TRACE C 이후에 언어 프로세서는 실행 전에 각 명령을 추적한 다음 실행하여 명령의 리턴 코드를 현재 터미널 출력 디바이스로 보냅니다. 현재 터미널 출력 디바이스 지정에 대한 자세한 정보는 SET TERMOUT 명령을 참조하십시오. [SET](#)의 내용을 참조하십시오.

## TRACE E

프로그램에서 TRACE E를 지정하면 언어 프로세서는 실행 후 리턴 코드가 0이 아닌 호스트 명령을 추적하고 명령에서 터미널로 리턴 코드를 보냅니다.

프로그램에 TRACE E가 포함되어 있고 올바르게 실행된 명령을 발행하면 프로그램은 오류 메시지, 행 번호, 명령 및 명령의 리턴 코드를 출력 스트림으로 보냅니다.

TRACE 명령어에 대한 자세한 정보는 [TRACE](#)를 참조하십시오.

## REXX 특수 변수 RC 및 SIGL 사용

REXX 언어에는 세 개의 특수 변수(RC, SIGL 및 RESULT)가 있습니다. REXX/CICS가 특정 상황 중 이 변수를 설정하고 언제든지 표현식에서 이를 사용할 수 있습니다.

REXX/CICS가 값을 설정하지 않은 경우, REXX의 다른 변수와 같이 특수 변수의 이름에는 대문자의 해당 고유 이름의 값이 있습니다. 두 가지 특수 변수(RC 및 SIGL)를 사용하여 프로그램의 문제점을 진단할 수 있습니다.

### RC

RC는 리턴 코드를 의미합니다. 프로그램이 명령을 실행할 때마다 언어 프로세서는 RC를 설정합니다. 명령이 오류 없이 종료되면 RC는 대개 0입니다. 명령이 오류로 종료되면 RC는 해당 오류에 지정된 리턴 코드입니다.

RC 변수는 프로그램이 어떤 경로를 사용해야 하는지 판별하도록 IF 명령어에 특히 유용할 수 있습니다.

**참고:** 모든 명령은 RC에 대한 값을 설정하므로 프로그램 기간 동안 동일하게 유지되지 않습니다. RC를 사용할 때 테스트하려는 명령의 리턴 코드를 포함하는지 확인하십시오.

### SIGL

언어 프로세서는 함수, SIGNAL 또는 CALL 명령어로 인해 프로그램 내에서 제어 전송과 관련하여 SIGL 특수 변수를 설정합니다. 언어 프로세서가 제어를 다른 루틴이나 프로그램의 다른 파트로 전송할 때 SIGL 특수 변수를 전송이 발생한 행 번호로 설정합니다. (다음 예제의 행 번호는 예제 뒤 설명에 도움이 됩니다. 프로그램의 파트가 아닙니다.)

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

호출된 루틴 자체가 다른 루틴을 호출하면 SIGL은 가장 최근 전송이 발생한 행 번호로 재설정됩니다.

SIGL 및 SIGNAL ON ERROR 명령어는 오류를 발생시킨 명령과 오류를 판별하는 데 도움이 됩니다. SIGNAL ON ERROR가 프로그램에 있는 경우 0이 아닌 리턴 코드를 리턴하는 호스트 명령은 error라는 루틴으로 제어를 전송합니다. 오류 루틴은 일반적으로 발생하는 다른 조치(예: 오류 메시지 전송)에 관계없이 실행됩니다.

SIGNAL 명령어에 대한 자세한 정보는 [SIGNAL](#)의 내용을 참조하십시오.

## 대화식 디버그 기능을 사용하여 추적

대화식 디버그 기능은 사용자가 프로그램의 실행을 제어하도록 합니다. 언어 프로세서는 터미널에서 읽고 출력을 터미널에 기록합니다.

## 대화식 디버그 시작

대화식 디버그를 시작하려면 TRACE 명령어의 옵션 앞에 ?를 지정하십시오. 예를 들면, TRACE ?A입니다. 물음표와 옵션 사이에는 공백이 있을 수 없습니다. 대화식 디버그는 호출된 외부 루틴으로 전달되지 않지만 루틴이 추적된 프로그램으로 리턴되면 재개됩니다.

## 대화식 디버그의 옵션

대화식 디버그가 시작된 후 일시정지 동안 또는 언어 프로세서가 입력 스트림에서 읽을 때마다 다음 중 하나를 제공할 수 있습니다.

- 추적을 계속하는 널 행. 언어 프로세서는 다음에 일시정지하거나 입력 스트림에서 읽을 때까지 실행을 계속합니다. 따라서 널 행을 반복해서 입력하면 프로그램이 종료될 때까지 일시정지 지점에서 일시정지 지점으로 단계적으로 이동합니다.
- 추적된 마지막 명령어를 다시 실행하는 등호(=). 언어 프로세서는 이전에 추적된 명령을 입력 스트림에서 읽은 명령에 의해 수정된 값으로 다시 실행합니다. (입력은 변수 값을 변경하는 지정일 수도 있습니다.)
- 추가 명령어. 이 입력은 명령 또는 다른 프로그램 호출을 포함한 모든 REXX 명령일 수 있습니다. 이 입력은 프로그램의 다음 명령어가 추적되기 전에 처리됩니다. 예를 들면, 입력은 추적 유형을 변경하는 TRACE 명령어일 수 있습니다.

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

입력은 지정 명령어일 수 있습니다. 이는 IF THEN ELSE 명령어에서 특정 분기를 강제 실행하도록 변수 값을 변경하여 프로그램 플로우를 변경할 수 있습니다. 다음 예에서 RC는 이전 명령으로 설정됩니다.

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

명령이 0이 아닌 리턴 코드로 끝나면 ELSE 경로가 사용됩니다. 첫 번째 경로를 강제로 사용하기 위해 대화식 디버그 중 입력은 다음과 같을 수 있습니다.

```
RC = 0
```

## 엔딩 대화식 종료

다음 방법 중 하나로 대화식 디버그를 종료할 수 있습니다.

- TRACE OFF 명령어를 입력으로 사용하십시오. 대화식 디버그 시작 시 메시지에 설명된 대로 TRACE OFF 명령어는 추적을 종료합니다.

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

- TRACE ? 명령어를 입력으로 사용하십시오.

TRACE 옵션 앞의 물음표 접두부는 대화식 디버그 및 시작을 종료할 수 있습니다. 물음표는 대화식 디버그에 대한 이전 설정(켜기 또는 끄기)을 반대로 바꿉니다. 따라서 프로그램 내에서 TRACE ?R을 사용하여 대화식 디버그를 시작하고 대화식 디버그를 종료하지만 지정된 옵션으로 추적을 계속하는 옵션 앞에 ?로 다른 TRACE 명령어의 입력을 제공할 수 있습니다.

- 입력으로서 옵션 없이 TRACE를 사용하십시오. 입력 스트림에서 옵션 없이 TRACE를 지정하면 대화식 디버그가 꺼지지만 TRACE Normal로 추적은 계속 유효합니다. (TRACE Normal은 실행 후 실패한 명령만 추적합니다.)
- 프로그램이 끝날 때까지 실행되게 합니다. 추적을 시작한 프로그램이 종료되면 대화식 디버그가 자동으로 종료됩니다. EXIT 명령을 입력으로 사용하여 프로그램을 조기에 종료할 수 있습니다. EXIT 명령어는 프로그램과 대화식 디버그를 모두 종료합니다.

## 대화식 TRACE 출력 저장

REXX/CICS는 추적 출력을 파일로 라우팅할 수 있는 기능을 제공합니다.

REXX 명령 SET TERMOUT은 라인 모드 출력(예: SAY 및 TRACE 출력)을 현재 터미널 디바이스 대신 파일로 경로 지정합니다. [SET](#)의 내용을 참조하십시오.



## 제 9 장 REXX/CICS 도움말 유틸리티 사용

REXX/CICS에는 REXX/CICS와 함께 제공되는 제품 문서를 검색하고 표시하는 데 사용할 수 있는 온라인 도움말 유틸리티가 포함되어 있습니다.

### 이 태스크 정보

도움말 유틸리티는 목차, 도움말 항목 및 검색 결과 패널을 제공합니다.

온라인 제품 문서보다 최신 업데이트에 대해서는 최신 릴리스의 [z/OS®용 CICS Transaction Server 제품 정보](#)와 함께 제공되는 REXX for CICS Transaction Server 제품 문서를 확인하는 것이 좋습니다.

### 프로시저

#### 도움말 액세스

- 다음 방법으로 도움말에 액세스할 수 있습니다.
  - CICS 환경에서 REXX CICHELP를 입력하십시오. 도움말 목차가 표시됩니다.
  - REXX/CICS 대화식 환경에서(이전에 매개변수 없이 CICS 트랜잭션 REXX를 입력한 경우) HELP를 입력하십시오. 도움말 목차가 표시됩니다.
  - REXX/CICS 텍스트 편집기에서 HELP를 입력하거나 F1(HELP)을 누르십시오. 해당 편집기에 대한 도움말이 표시됩니다.
  - 파일 목록 유틸리티에서 HELP를 입력하거나 F1(HELP)을 누르십시오. 해당 유틸리티에 대한 도움말이 표시됩니다.

#### 도움말 항목에 액세스

- 다음 방법으로 도움말 항목에 액세스할 수 있습니다.
  - REXX/CICS 대화식 환경에서 HELP와 주제 제목의 검색 문자열(예: HELP TRANSLATE)을 입력하십시오. 제목에 검색 문자열이 대문자인 첫 번째 도움말 항목이 표시됩니다.

이 방법은 특정 REXX 명령, 기능 또는 키워드에 대한 도움말을 찾는 데 유용할 수 있습니다.
  - 앞에서 설명한 대로 도움말 목차에 액세스한 후 목록에서 주제를 선택하십시오. 선택할 주제의 행에 커서를 놓거나 입력 필드에 주제의 행 번호를 입력한 후 Enter를 누르십시오. 선택한 주제가 표시됩니다.
  - 앞에서 설명한 대로 도움말 목차에 액세스한 후 검색 문자열을 입력하십시오. 검색 문자열을 포함하는 모든 주제가 검색 결과 패널에 나열됩니다. 검색은 대소문자를 구분하지 않습니다. 대부분의 경우 검색 결과 패널에 주제 이름과 상위 주제가 모두 표시됩니다. 예를 들어, ADDRESS를 검색하는 경우 결과는 다음과 같이 표시됩니다.

키워드 명령어 > ADDRESS

목록에서 주제를 선택하려면 선택할 주제의 행에 커서를 놓거나 입력 필드에 주제의 행 번호를 입력한 후 Enter를 누르십시오. 선택한 주제가 표시됩니다.

검색 결과 패널에 나열되는 모든 주제를 표시하려면 입력 필드에 ALL을 입력하십시오.

목차, 도움말 항목 또는 검색 결과 패널 스크롤링

- 다음 화면, 앞으로 스크롤하려면 F8(FORWARD)을 누르십시오.

현재 도움말 항목의 끝에 있으면 다음 메시지가 표시됩니다.

End of topic -- press F8 for next topic

목차 또는 검색 결과 패널의 끝에 있으면 다음 메시지가 표시됩니다.

Already at the bottom

- 화면의 일부를 기준으로 앞으로 스크롤하려면 도움말 화면의 행에 커서를 놓은 다음 F8(FORWARD)을 누르십시오.  
선택한 행이 도움말 화면에 표시되는 첫 번째 행이 됩니다.
- 이전 화면, 뒤로 스크롤하려면 F7(BACKWARD)을 누르십시오.  
현재 도움말 항목의 시작에 있으면 다음 메시지가 표시됩니다.

Start of topic -- press F7 for previous topic

목차 또는 검색 결과 패널의 시작에 있으면 다음 메시지가 표시됩니다.

Already at the top

- 화면의 일부를 기준으로 뒤로 스크롤하려면 도움말 화면의 행에 커서를 놓은 다음 F7(BACKWARD)을 누르십시오.  
선택한 행이 도움말 화면에 표시되는 마지막 행이 됩니다.
- 도움말, 도움말 항목 또는 검색 결과 패널에서 종료
- 도움말, 도움말 항목 또는 검색 결과 패널을 종료하려면 F3(END 기능)을 누르십시오. 사용한 이전 화면으로 돌아갑니다.

예를 들어, 목차가 표시되고 F3을 누르면 도움말에 액세스한 방법에 따라 CICS 환경 또는 REXX/CICS 환경으로 돌아갈 수 있습니다. 주제가 표시되고 F3을 누르면 도움말 항목에 액세스한 방법에 따라 검색 결과 패널 또는 도움말 목차로 돌아갈 수 있습니다.

## 제 10 장 프로그래밍 스타일 및 기술

프로그램을 구성하는 데 사용하는 방법은 프로그램을 작성하는 데 사용하는 언어만큼 중요합니다.

### 데이터 고려

프로그램 작성 태스크를 시작할 때 가장 먼저 고려해야 하는 사항은 처리해야 하는 데이터입니다. 입력 데이터 목록을 작성하십시오. 항목이 무엇이고 각각 가능한 값은 무엇입니까? 항목에 일종의 구조나 패턴이 있다면 이를 설명하기 위해 다이어그램을 그립니다. 그런 다음 출력 데이터에 대해 동일한 작업을 수행하십시오. 두 개의 다이어그램을 살펴보고 서로 맞는지 확인하십시오. 만약 그렇다면 프로그램 디자인을 잘 수행하고 있는 것입니다.

다음으로 사용자가 사용할 사양을 작성하십시오. 이는 작성 스펙, HELP 파일 또는 둘 다일 수 있습니다.

맨 마지막으로 프로그램을 작성하십시오.

다음은 예입니다.

"헤드 또는 후미"를 재생하도록 사용자를 초대하는 대화식 프로그램을 작성해야 합니다. 사용자가 좋아하는 한 게임을 즐길 수 있습니다. 게임을 종료하려면 사용자는 질문 "헤드 또는 후미?"에 대해 Quit를 응답해야 합니다. 이 프로그램은 컴퓨터가 항상 이기도록 구성되어 있습니다.

이 프로그램을 작성하는 방법에 대해 생각해 보십시오.

컴퓨터는 다음과 같이 시작합니다.

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

이는 네 개의 가능한 입력이 있음을 의미합니다.

- HEADS
- TAILS
- QUIT
- 이 세 항목이 아님.

따라서 해당 출력은 다음과 같아야 합니다.

- 죄송합니다. TAILS였습니다. 애석합니다!
- 죄송합니다. HEADS였습니다. 애석합니다!
- 정답이 아닙니다. 다시 시도하십시오!

그리고 이 시퀀스는 CICS에 리턴으로 끝나면서 끝없이 반복되어야 합니다.

이제 사양, 입력 데이터 및 출력 데이터를 이해했으므로 프로그램을 작성할 준비가 되었습니다.

데이터를 고려하지 않고 몇 가지 지침을 작성하여 시작한 경우 시간이 더 오래 걸렸을 수 있습니다.

### 직접 테스트

프로그램을 작성한 후 조심하려면 처음으로 실행해야 합니다!

```

/* 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

```

## 휴식 시간

잠시 재미있는 시간을 보낼 수 있는 기회가 있습니다. 매우 간단한 아케이드 게임입니다. 게임을 입력하고 친구들과 함께 즐기십시오. 나중에 향상시키고 싶을 수도 있습니다.



```

/* 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 */

```

수고하셨습니다! 이제 잠시 시간을 내어 새로운 스킬을 실행에 옮기거나 계속해서 읽으십시오.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## 프로그램 디자인

언어만큼이나 중요한 방법인 CATMOUSE를 다시 한 번 살펴보겠습니다.

이 프로그램은 고양이, 쥐 및 복도에서의 위치에 관한 것입니다. 일부 단계에서는 화면에 해당 위치가 표시되어야 합니다. 모든 것을 한 번에 생각하기에는 너무 복잡합니다. 첫 번째 단계는 다음과 같이 분류하는 것입니다.

- **주 프로그램:** 해당 위치 계산
- **서브루틴 표시:** 해당 위치 표시.

이제 주 프로그램을 보도록 합시다. 사용자(쥐를 플레이하는 사용자)는 이동하기 전에 모두 어디에 있는지 알고 자 할 것입니다. 고양이는 그렇지 않을 것입니다. 다음 단계는 주 프로그램을 다음과 같이 분류하는 것입니다.

```
Do forever
  call Display
  Mouse's move
  Cat's move
end
Conclusion
```

## 루프를 디자인하기 위한 메소드

루프를 디자인하는 방법은 두 가지 질문(항상 종료하는지, 종료될 때마다 데이터가 필요한 조건을 충족하는지)을 하는 것입니다.

CATMOUSE 예제에서 다음과 같은 경우 루프가 종료되고 게임이 종료됩니다.

1. 쥐가 구멍을 향해 달려갈 때
2. 쥐가 고양이를 향해 달려갈 때
3. 고양이가 쥐를 잡을 때.

## 결론

프로그램이 끝나면 사용자에게 발생한 상황에 대해 알려야 합니다.

```
call display
say who won
```

## 지금까지 수행한 작업

이 작업 항목을 모두 모으면 다음과 같습니다.

```

/*-----*/
/* 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:
...

```

방금 설명한 방법을 단계별 세분화라고 합니다. (불완전할 수 있는) 스펙으로 시작합니다. 그런 다음 제안된 프로그램 루틴으로 나누면 프로그램 전체보다 각 루틴을 코딩하는 것이 더 쉽습니다. 그런 다음 첫 번째 시도에서 올바르게 코딩할 수 있는 루틴에 도달할 때까지 이러한 각 루틴에 대해 프로세스를 반복하십시오.

이 작업을 수행하는 동안 스스로에게 두 가지 질문을 계속하십시오.

- 이 루틴이 처리하는 데이터는 무엇입니까?
- 스펙이 완료되었습니까?

## 단계별 세분화: 예제

작업을 더 간단한 작업으로 나누는 것을 단계별 세분화라고 합니다.

할머니가 손자가 향해하러 갈 때 입을 따뜻한 모직 옷을 뜨개질합니다. 이는 할머니가 수행할 작업입니다.

1. 앞면 뜨기
2. 뒷면 뜨기
3. 왼쪽 팔 뜨기
4. 오른쪽 팔 뜨기
5. 서로 꿰매 붙이기.

이러한 각 작업은 폴오버 뜨기 작업보다 설명하기가 더 간단합니다. 컴퓨터 전문 용어로 작업을 더 간단한 작업으로 나누는 것을 단계별 세분화라고 합니다.

이 단계에서, 다시 스펙을 보십시오. 선원은 앞면이나 뒷면을 걱정하지 않고 어둠 속에서 폴오버를 신속하게 착용해야 할 수도 있습니다. 따라서 앞면은 뒷면과 같아야 합니다. 두 슬리브도 동일해야 합니다. 이는 다음과 같이 프로그래밍될 수 있습니다.

```

do 2
  CALL Knit_body_panel
end

do 2
  CALL Knit_sleeve

```

```
end  
CALL sew_pieces_together
```

프로그래밍에서 가장 좋은 방법은 코딩하기 쉬운 것에 도달할 때까지 하향식으로 프로그램을 세분화하는 것입니다.

하향식이 가장 좋은 방법입니다.

## 데이터 재고려

프로그램을 세분화할 때 사용자의 목적은 각 부분을 더 단순하게 하는 것입니다.

이는 다음을 의미할 확률이 높습니다.

- 각 세그먼트 또는 루틴에 대해 더 간단한 입력 데이터
- 각 세그먼트 또는 루틴에 대해 더 간단한 출력 데이터
- 더 간단한 처리
- 그리고 결과적으로 더 간단한 코드입니다.

파트가 더 간단하면 이름도 더 간단할 수 있습니다. 예를 들면 다음과 같습니다.

- 소매 끝동 뜨기

다음보다 위의 표현식을 사용하는 것이 좋습니다.

- 소매 끝동과 허리 밴드를 위한 골 만들기

## 프로그램 수정

다양한 기술을 사용하여 프로그램을 수정할 수 있습니다.

프로그램이 잘못된 결과를 제공하는 이유를 파악할 수 없는 경우:

- 수행 중인 작업을 표시하도록 프로그램을 수정할 수 있습니다.
- REXX 대화식 추적 기능을 사용할 수 있습니다. [프로그램의 대화식 디버깅](#)의 내용을 참조하십시오.

사용하는 데 선호하는 기술을 평가할 수 있습니다.

## 프로그램 수정

프로그램을 수정하고 추가 명령어를 넣어 발생하는 상황을 사용자에게 알려줄 수 있습니다.

다음과 같은 추가 명령어를 프로그램에 넣을 수 있습니다.

```
...  
say "Checkpoint A. x =" x  
...  
say "End of first routine"  
...
```

## 프로그램 추적

REXX 대화식 추적 기능을 사용하여 프로그램에서 발생하는 상황을 파악할 수 있습니다.

TRACE 명령어에 대한 세부사항은 [TRACE](#)를 참조하십시오.

- 프로그램이 어떻게 진행되는지 확인하려면 TRACE Labels를 사용하십시오. 이 예제는 프로그램과 화면에 표시되는 추적을 표시합니다.

```

/* 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:

```

그림 45. ROTATE

이는 다음 추적을 제공합니다.

```

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:

```

- 언어 프로세서에서 표현식을 계산하는 방법을 확인하려면 TRACE Intermediates를 사용하십시오.
- 올바른 데이터를 명령 또는 서브루틴으로 전달하는지 확인하려면 TRACE Results를 사용하십시오.
- 명령에서 0이 아닌 리턴 코드가 표시되도록 하려면 TRACE Errors를 사용하십시오.

## 코딩 스타일

프로그램을 읽고 올바른지 파악할 수 있기 위해서는 읽기 쉬어야 합니다.

"가독성"은 프로그래머마다 다른 의미입니다. 여기에서는 다양한 스타일의 예를 제공합니다. 선호하는 스타일을 선택할 수 있습니다. 프로그램을 확인하기 위한 가장 좋은 방법은 동료에게 읽도록 요청하는 것입니다. 따라서 동료가 쉽게 읽을 수 있는 코딩 스타일을 사용하십시오.

98 페이지의 『휴식 시간』의 프로그램 예에서 가져온 다음 프로그램 단편은 대부분의 사람들이 읽기 어렵습니다.

```

/*****
/* 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

```

이 다음 예는 읽기가 더 쉽습니다. 각각 고유한 표제가 있는 세그먼트로 나뉘어져 있습니다. 오른쪽에 있는 주석을 종종 비교라고 합니다. 독자들이 무슨 상황이 발생하고 있는지 전반적인 아이디어를 얻는 데 도움이 될 수 있습니다.

```

/*****
/* 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

```

다음 예에는 일부 프로그래머에게 인기있는 추가 기능이 있습니다. 대문자로 작성된 키워드와 다양한 들여쓰기 스타일은 코드의 구조를 강조표시합니다. 풍부한 주석은 스펙에 대한 세부사항을 상기시킵니다.

```

/*****
/* 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

```

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.



---

## 제 2 부 REXX 구성

이 절에서는 REXX을 구성하고 관리하는 요소를 소개합니다.



## 제 11 장 REXX 지원 구성

REXX 프로그램을 실행하기 전에 REXX 지원을 구성해야 합니다.

다음 단계를 수행하십시오.

1. RFS 파일 폴을 작성합니다.
2. 자원 정의를 작성합니다.
3. LSRPOOL 정의를 검토합니다.
4. CICSTART 멤버를 업데이트합니다.
5. CICS 초기화 JCL을 수정하십시오.
6. 설치를 확인합니다.
7. RFS 파일 폴의 형식을 지정합니다.
8. 도움말 파일을 작성합니다.
9. REXX Db2 인터페이스를 구성하십시오.

### RFS 파일 폴 작성

RFS(REXX Filing System)에서는 두 개 이상의 파일 폴을 사용하여 데이터를 저장합니다.

파일 폴은 VSAM 클러스터 세트로 구현됩니다. CICSTS56.REXX.SCICJCL에서 제공되는 멤버인 CICVSAM은 두 RFS 파일 폴에 대한 VSAM 데이터 세트를 작성합니다. 사용자의 고유 환경을 나타내도록 필요한 대로 이 작업을 변경하십시오. 예를 들어 설치 표준에 맞게 데이터 세트 이름을 변경할 수 있습니다. 변경을 완료한 후 파일 폴을 정의하는 작업을 실행하십시오.

파일 폴 이름을 변경하거나 자원 정의에 파일 폴 구성요소를 추가하는 경우 CICSTART 멤버에서 파일 폴 정의를 이에 맞게 변경해야 합니다. [109 페이지의 『자원 정의 작성』](#) 및 [110 페이지의 『CICSTART 멤버 업데이트』](#)의 내용을 참조하십시오.

다음 정의가 일관된지 확인하십시오.

- CICSTART의 파일 폴 정의
- CICRDOD(개발 시스템의 경우) 또는 CICRDOR(런타임 기능의 경우)의 자원 정의
- CICVSAM의 VSAM 클러스터 정의

### 자원 정의 작성

REXX용 프로파일, 프로그램, 트랜잭션 및 파일 자원의 정의는 제공되었거나 업그레이드된 CSD에 있는 CICEXX 그룹에 있습니다.

CICSTS56.REXX.SCICJCL 데이터 세트에서 Runtime Facility의 경우 CICRDOR 작업, Development System의 경우 CICRDOD 작업은 REXX/CICS 프로파일, VSAM 파일, 프로그램, 트랜잭션 및 트랜지언트 데이터 큐를 포함하여 제품이 필요한 항목을 추가합니다.

트랜지언트 데이터 큐는 REXX/CICS IMPORT 및 EXPORT 명령에 사용됩니다. 작업은 Db2 계획에 대해 트랜잭션 권한을 부여하는 REXX/CICS SQL 인터페이스에 대한 정의도 포함합니다.

JCL 시작 부분의 주석에 설명된 대로 항목의 주석을 해제하여 JCL을 편집하고 작업을 실행하십시오.

#### IMPORT 및 EXPORT 명령의 TD 큐 수정

REXX/CICS 환경에서는 동적 할당을 사용하여 파티션된 데이터 세트에서 멤버 IMORT를 수행하고 RFS 파일을 파티션된 데이터 세트로 EXPORT를 수행합니다.

CICSTS56.REXX.SCICJCL 데이터 세트의 CICRDOD 또는 CICRDOR 멤버는 IMPORT의 입력으로 사용할 세 개의 트랜지언트 데이터 항목과 EXPORT의 출력으로 사용할 세 개의 트랜지언트 데이터 항목을 정의하므로, 세 명

의 사용자가 파티션된 데이터 세트에서 동시에 IMPORT를 수행하고 세 명의 사용자가 파티션된 데이터 세트에 동시에 EXPORT를 수행할 수 있습니다.

TDQ 항목 수를 사용자 요구사항에 맞게 수정하지만, 최소한 하나의 입력 및 하나의 출력 항목을 허용하십시오. TDQUEUE NAME은 REX로 시작하고 올바른 문자가 뒤에 붙어야 합니다. IMPORT와 EXPORT에서 사용하고 파일이 손상될 수 있으므로 REX로 시작하는 TDQUEUE 이름을 사용하는 다른 애플리케이션을 사용하지 마십시오.

## LSRPOOL 정의 검토

RFS 파일의 최대 키 길이는 252이고 제공된 VSAM 정의에서는 18K의 제어 간격 크기를 사용하므로, RFS에 사용되는 LSRPOOL에서 이 값을 지원하는지 확인하는 것이 중요합니다.

이 작업에 실패하면 시스템 콘솔에 OPEN 오류가 표시될 수 있습니다. 자세한 정보는 [LSRPOOL 자원 및 로컬 공유 자원\(LSR\)](#) 또는 [비공유 자원\(NSR\)](#)의 내용을 참조하십시오.

## CICSTART 멤버 업데이트

hlq.CICSTS56.REXX.SCICEXEC 데이터 세트의 CICSTART 멤버는 REXX/CICS 환경의 기본 정의를 포함합니다. CICS 시스템이 시작한 후 CICS/REXX 프로그램을 사용하는 첫 번째 트랜잭션이 실행되면 CICSTART가 실행됩니다.

CICSTART에서는 환경의 영구 구성을 보유하고 있습니다. 제공된 CICSTART 멤버를 편집하거나 고유 CICSTART exec의 모델로 사용하십시오.

- 초기화

의사 대화식 모드를 꺼짐으로 설정하는 초기화 명령문(PSEUDO OFF)을 사용하고 제공된 CICSTART 멤버 시작 시 표시된 것과 RLS 디렉토리(RLS CKDIR)가 똑같은지 확인하십시오.

- RFS 파일 풀 정의

제공된 CICSTART 멤버에서 다음 두 개의 파일 풀을 정의합니다.

```
'FILEPOOL DEFINE POOL1 RFSDIR1 RFSP00L1 (USER'
  IF RC -> 0 THEN EXIT RC
'FILEPOOL DEFINE POOL2 RFSDIR2 RFSP00L2 (USER'
  IF RC -> 0 THEN EXIT RC
```

109 페이지의 『RFS 파일 풀 작성』 단계 중에 파일 풀 이름을 변경했거나 파일 풀을 추가한 경우 CICSTART exec의 파일 풀 정의를 적절하게 변경하십시오.

- 사용자 ID 권한 부여

제공된 CICSTART 멤버가 사용자 ID RCUSER에 권한을 부여합니다. 필요한 조직의 사용자에게 권한을 부여하는 명령이 CICSTART exec에 포함되어 있는지 확인하십시오.

파일 풀을 형식화하기 위해 액세스해야 하는 모든 사용자를 포함하십시오. 그렇지 않으면 리턴 코드 -4를 표시하며, FILEPOOL FORMAT 명령이 실패합니다.

CICS 보안이 꺼져 있는 리전의 경우(즉 SIT 매개변수 SEC=NO), CICS용 CICS TART 기본 사용자 ID에 AUTHUSER 명령을 포함하십시오. 예를 들어 리전의 기본 사용자 ID가 SYSA이면 다음 명령을 포함시키십시오.

```
'AUTHUSER SYSA'
```

- 필요한 대로 CICS 트랜잭션 ID와 REXX exec 연관

CICSTART에는 CICS DEFINE TRANSACTION 명령에서 정의한 대로 CICS 트랜잭션 ID를 호출할 REXX exec와 연관시키는 DEFTRNID문이 포함되어 있습니다. 제공된 CICSTART 멤버는 REXX, EDIT 및 FLST 트랜잭션을 CICRXTRY, CICIPIT 및 CICFLST exec와 각각 연관시킵니다. 이 CICS 트랜잭션은 CICRDOR 및 CICRDOD JCL 파일에서 정의됩니다. 109 페이지의 『자원 정의 작성』의 내용을 참조하십시오.

CICSTART exec에는 REXX/CICS 트랜잭션을 실행하도록 정의한 모든 CICS 트랜잭션의 DEFTRNID문이 포함되어야 합니다.

- REXX exec 실행용 기본값 설정

환경에 필요한 SETSYS 명령을 포함시키십시오. 예를 들어 설치에서 REXX/CICS exec를 실행하는 데 사용할 언어, 검색 PF 키 및 의사 대화식 설정을 지정할 수 있습니다.

기본적으로 의사 대화식 모드는 켜져 있습니다(단, CICSTART exec 자체가 대화식 모드로 실행되어야 함). 의사 대화식 모드에 대한 자세한 정보는 [PSEUDO](#) 및 [사용자와 대화 상자 처리](#)의 내용을 참조하십시오.

- 도움말 경로 정의

제공된 CICSTART 멤버는 현재 도움말 유틸리티용으로 HELPPTH2 도움말 경로를 정의하고, 호환성을 위해 이전 도움말 기능용으로 HELPPTH 도움말 경로를 정의합니다. facility:

```
HELPPATH = 'POOL2:\BOOK'  
'RLS VARPUR HELPPATH \SYSTEM\DEFAULTS'  
IF RC ~= 0 THEN EXIT  
HELPPTH2 = 'POOL2:\HELP'  
'RLS VARPUR HELPPTH2 \SYSTEM\DEFAULTS'  
IF RC ~= 0 THEN EXIT RC
```

이전 도움말 기능이 설치되지 않은 새 설치의 경우 HELPPTH를 정의한 세 개의 명령문을 제거할 수 있습니다.

- EXECLOAD를 사용하여 필요한 대로 exec 사전 로드

exec를 사전 로드하도록 EXECLOAD문을 포함시킬 수 있습니다. exec가 사전 로드되면 사용자마다 exec를 로드하지 않아도 되므로 성능을 향상시킬 수 있고, 모든 동시 사용자가 동일한 사본을 공유할 수 있으므로 필요한 저장영역이 줄어듭니다.

제공된 CICSTART 멤버에서 EXECLOAD 명령 목록을 사용하거나 필요한 exec의 EXECLOAD 명령을 포함시키십시오.

제공된 CICSTART 멤버에는 CICEDIT, CICESVR 및 CICEPROF exec의 EXECLOAD 명령이 포함되어 있습니다. 이 exec는 고유 REXX 프로그램을 개발할 때 사용할 수 있고 REXX Development System만의 일부인 REXX/CICS 텍스트 편집기의 구성요소입니다. 설치에서 REXX/CICS 텍스트 편집기를 사용하면 이 EXECLOAD 명령을 포함할 수 있습니다.

자세한 정보는 [EXECLOAD](#)의 내용을 참조하십시오.

## CICS 초기화 JCL 수정

CICS 시동 작업 및 DFHRPL 연결에 DD문을 추가하십시오.

CICS 시동 작업에 다음 DD문을 추가하십시오.

```
//CICAUTH DD DSN=CICSTS56.REXX.SCICCMDS,DISP=SHR  
//CICEXEC DD DSN=CICSTS56.REXX.SCICEXEC,DISP=SHR  
//CICUSER DD DSN=CICSTS56.REXX.SCICUSER,DISP=SHR
```

다음과 같이 DFHRPL 연결에 REXX 데이터 세트의 DD문을 추가하십시오.

```
//DFHRPL DD DSN=CICSTS56.REXX.SCICLOAD,DISP=SHR
```

REXX/CICS 환경에서는 CICS에서 자원 정의가 없는 세 개의 데이터 세트 연결, 즉 CICCMDS, CICEXEC 및 CICUSER DD 이름을 사용합니다. 이 데이터 세트는 파티션된 데이터 세트이고, MVS 기능을 사용하여 액세스됩니다.

### CICCMDS

CICCMDS DD 이름 연결은 CICSTS56.REXX.SCICCMDS 데이터 세트를 참조하여 시작합니다. 이 데이터 세트에는 REXX/CICS 권한 부여된 명령을 구현하는 exec를 포함합니다. 권한 부여된 사용자 또는 권한 부여된 명령을 사용하도록 권한 부여된 exec만 이 exec에 액세스할 수 있습니다. 권한 부여된 고유 명령으로 REXX/CICS 환경을 확장하는 경우, 데이터 세트를 이 DD 이름 연결에 결합하십시오.

### CICEXEC

CICEXEC DD 이름 연결은 CICSTS56.REXX.SCICEXEC 데이터 세트를 참조하여 시작합니다. 이 데이터 세트는 권한 부여된 명령을 사용하는 REXX/CICS 환경에서 제공하는 exec를 포함합니다. 권한 부여된 명령을 사용하는 고유 exec로 REXX/CICS 환경을 확장하는 경우, 데이터 세트를 이 DD 이름 연결에 결합하십시오.

## CICUSER

CICUSER DD 이름 연결은 CICSTS56.REXX.SCICUSER 데이터 세트를 참조하여 시작합니다. 이 데이터 세트는 권한 부여된 명령을 사용하는 REXX/CICS 환경에서 제공하는 exec를 포함합니다. 권한 부여된 명령을 사용하지 않는 고유 exec로 REXX/CICS 환경을 확장하는 경우, 데이터 세트를 이 DD 이름 연결에 결합하십시오.

이러한 데이터 세트 연결에 액세스하는 데 사용되는 기능은 CICS WAIT EXTERNAL 기능을 사용하여 CICS 리전을 대기 상태로 만들지 않도록 합니다.

## RHS 파일 풀 형식화

CICSTART에 정의된 모든 RFS(REXX Filing System) 파일 풀을 형식화해야 합니다.

1. 필요한 모든 구성 태스크가 완료되었는지 확인하고 필요하면 CICS를 다시 시작하십시오.
2. CICSTART에 권한 부여된 사용자로 정의된 userid로 사인온하십시오.
3. REXX(CICRXTY exec와 연관된 기본 트랜잭션 id임)를 입력하십시오.

이 조치는 REXXTRY 유틸리티를 호출하며, 대화식으로 REXX 및 REXX/CICS 명령을 실행할 수 있는 대화식 환경을 제공합니다. 다음 행이 화면 맨 위에 표시되고, READ가 오른쪽 하단에 표시되며, 커서는 왼쪽 하단에 표시됩니다.

Enter a REXX command or EXIT to quit

4. CICSTART에 정의된 모든 파일 풀에 다음 명령을 입력하여 사용할 파일 풀을 준비하십시오.

'FILEPOOL FORMAT *poolname*'

*poolname*은 CICSTART exec에 지정한 파일 풀 이름입니다(기본값은 POOL1 및 POOL2임).

작은따옴표 또는 큰따옴표를 사용하여 REXX 명령을 구분하는 것이 좋습니다.

대화식 환경에서는 화면에서 사용 가능한 다음 행에 각 명령을 다시 표시하고 요청된 출력을 표시합니다.

**FILEPOOL FORMAT** 명령에서는 성공적으로 완료되었음을 나타내는 정보를 표시하지 않지만, 화면 오른쪽 하단에 READ라는 단어가 표시됩니다.

5. **FILEPOOL FORMAT** 명령에 성공했는지 판별하려면 SAY RC(어포스트로피 없이)를 입력하십시오. 사용 가능한 다음 행에 0이 표시되면 명령에 성공했음을 나타냅니다.

파일 풀을 다시 형식화하려고 하면 다음 메시지가 표시됩니다.

Subcommand return code = 1836

6. 모든 RFS 파일 풀이 형식화될 때까지 이 프로세스를 계속하십시오. 새 파일 풀을 정의하거나 기존 파일 풀의 클러스터를 삭제하고 다시 정의한 경우에만 파일 풀을 형식화합니다.

filepools의 형식을 지정하거나 REXX 또는 REXX/CICS 명령과 명령어를 대화식으로 실행한 경우, MORE 표시기가 맨 아래 오른쪽 구석에 표시됩니다. 화면을 지우려면 Enter 키를 누르십시오. 데이터 화면을 지우려면 언제든지 Clear 키를 누를 수 있습니다. PF3 키를 눌러 대화식 환경에서 나가십시오. 이 조치는 EXIT 지시어를 입력하는 경우를 시뮬레이션합니다. EXIT 지시어를 직접 입력할 수도 있습니다(어포스트로피가 없음).

대화식 환경에서 RETRIEVE 키를 눌러 이전에 입력한 명령을 재호출할 수 있습니다. 이 키의 기본 시스템 설정은 PF12입니다. RETRIEVE 키를 누르면 이전에 입력한 행을 입력 위치에 다시 표시합니다. 그런 다음 이 영역을 수정하고 Enter를 눌러 명령어를 다시 실행할 수 있습니다. RETRIEVE 키를 여러 번 누르면 이전에 입력한 명령을 입력 영역에 계속 표시합니다.

## 설치 확인

exec는 설치가 정상적으로 완료되었는지 확인하기 위해 제공됩니다.

1. REXX/CICS REXXTRY 유틸리티에 CALL CICIVP1을 입력하여 설치 검증 프로그램을 실행하십시오. exec의 출력은 다음과 같습니다(화면에 MORE가 표시되면 잊지 말고 Enter를 누르십시오).

```
EXEC CICIVP1
****-----***
*** This is a test REXX program running under CICS-TS ***
*** It was loaded from CICUSER-CICIVP1
```

\*\*\*-----\*\*\*

What is your name?

- 이름을 입력하고 Enter를 누르십시오. 출력은 다음과 같습니다.

```
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
```

## 도움말 파일 작성

REXX/CICS에는 REXX/CICS와 함께 제공되는 제품 문서를 검색하고 표시하는 데 사용할 수 있는 온라인 도움말 유틸리티가 포함되어 있습니다.

### 이 태스크 정보

설치의 일부로 CICS TS 제품 제공 데이터 세트의 정보를 REXX/CICS 파일 시스템에 로드해야 합니다.

### 프로시저

- RFS(REXX Filing System) 파일 폴을 정의하고 도움말 유틸리티의 도움말 경로를 설정하십시오.

*hlq.CICSTS56.REXX.SCICEXEC* 데이터 세트에 제공된 CICSTART 멤버에는 RFS 파일 폴을 정의하고 REXX 도움말의 도움말 경로를 설정하는 명령문이 포함되어 있습니다. 이러한 명령문을 확인하고 필요하면 요구사항에 맞게 변경하십시오. 세부사항은 [110 페이지의 『CICSTART 멤버 업데이트』](#)의 내용을 참조하십시오.

제공된 CICSTART 멤버가 현재 도움말 유틸리티의 도움말 경로 HELPPATH2를 정의합니다. 호환성을 위해 이전 도움말 기능의 도움말 경로 HELPPATH도 정의합니다. 이전 도움말 기능이 설치되지 않았으며 이 설치 단계가 완료될 때까지 HELPPATH2를 사용하려는 경우 HELPPATH는 필요하지 않습니다.

2. 114 페이지의 『3』 단계에서 도움말 정보를 로드할 사용자 ID가 CICS TS 제품 제공 라이브러리에서 REXX Filing System으로 관련 데이터 세트를 가져올 권한이 있는지 확인하십시오.

*hlq.CICSTS56.REXX.SCICEXEC* 데이터 세트의 *CICHLOAD* exec를 통해 제공된

*hlq.CICSTS56.REXX.SCICDOC* 및 *hlq.CICSTS56.REXX.SCICPNL* 데이터 세트에서 도움말 정보를 가져옵니다. REXX 도움말에서 사용하는 파일을 RFS에서 빌드하는 데 이 정보를 사용합니다.

이 권한을 확인하는 한 가지 방법은 관련 사용자 ID를 상위 레벨 규정자로 사용하여

*hlq.CICSTS56.REXX.SCICDOC* 및 *hlq.CICSTS56.REXX.SCICPNL*의 사본을 작성하는 것입니다.

3. REXX/CICS를 시작하고 EXEC CICHLOAD 명령을 실행하십시오.

CICHLOAD는 MVS 라이브러리에서 정보와 패널 정의를 가져옵니다. 다음 메시지가 표시되면 이전 단계에서 설정한 데이터 세트 이름을 제공하십시오.

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*

제공된 *CICSTS56.REXX.SCICDOC* 데이터 세트에는 온라인 도움말 정보를 포함하는 *CICRXHLP* 멤버가 포함되어 있습니다. *CICHLOAD* 프로그램에서 이 멤버를 처리하여 온라인 도움말 유틸리티에서 사용하는 파일을 작성합니다. 파일은 사용자가 지정한 도움말 경로 *HELPPTH2*에 작성됩니다.

이 데이터 세트에서 *CICR3270* 또는 *CICR3820* 멤버는 *CICRXHLP* 멤버 및 z/OS용 [CICS Transaction Server](#) 제품 정보와 함께 제공되는 PDF로 대체됩니다.

온라인 제품 문서보다 최신 업데이트에 대해서는 최신 릴리스의 z/OS용 [CICS Transaction Server](#) 제품 정보와 함께 제공되는 REXX for CICS Transaction Server 제품 문서를 확인하는 것이 좋습니다.

## REXX Db2 인터페이스 구성

이 단계는 REXX EXECSQL 명령 환경이 Db2 지원에 사용되는 경우에만 필요합니다. Db2 설치가 완료되어야 이 단계를 수행할 수 있습니다.

*CICSTS56.REXX.SCICJCL* 데이터 세트의 *CICRDOD* 또는 *CICRDOR* 멤버는 Db2 계획을 사용하도록 REXX 트랜잭션에 권한을 부여합니다.

REXX/CICS 환경에 제공된 트랜잭션을 수정하거나 Db2 인터페이스 코드를 사용하는 새 트랜잭션을 구현한 경우, Db2 항목 정의도 수정하거나 추가해야 합니다.

### CICSQL 프로그램을 Db2 계획에 바인딩

*CICSTS56.REXX.SCICJCL* 데이터 세트의 *CICBIND* 작업은 CICSQL을 올바른 Db2 패키지에 바인딩합니다. 작업을 편집하고 실행하십시오.

사용 중인 Db2 레벨에 따라 작업에서 조건 코드 4를 수신할 수 있습니다.



## 제 12 장 REXX/CICS 시스템 정의 및 관리

REXX/CICS의 시스템 정의, 사용자 정의 및 관리에 대해 설명합니다.

### 권한 부여된 REXX/CICS 명령 및 권한 부여된 명령 옵션

여러 REXX/CICS 명령 또는 명령 옵션이 권한 부여된 것으로 식별됩니다.

REXX/CICS 명령의 내용을 참조하십시오. 권한 부여된 REXX/CICS 명령은 다음과 같은 경우에만 실행할 수 있습니다.

- exec를 실행하는 사용자 ID는 권한 부여된 REXX/CICS 사용자입니다. 권한 부여된 사용자는 AUTHUSER 명령을 통해 정의합니다.
- exec는 REXX/CICS 권한 부여된 하위 라이브러리에서 로드되었습니다. 권한 부여된 라이브러리는 ddname CICAUTH 또는 CICEXEC에 할당된(CICS 시동 프로시저/JCL에서) MVS 파티션된 데이터 세트입니다.

이러한 규칙은 명령을 시도하는 exec가 권한 부여된 exec에서 실행되었는지에 상관없이 적용됩니다.

### 시스템 프로파일 exec

CICS 시스템이 재시작 후에 첫 번째 사용자 exec가 실행되기 전에 CICSTART라는 시스템 프로파일 exec가 실행됩니다.

일반적으로 시스템 프로파일 exec에는 ddname CICAUTH 또는 CICEXEC에 할당되어 있는 권한 부여된 MVS PDS REXX 라이브러리에 있어야 하는 시스템 사용자 정의 명령, 권한 부여된 하위 라이브러리 정의, 권한 부여된 사용자 정의 및 권한 부여된 명령 정의가 포함되어 있습니다.

CICS 시스템을 다시 시작한 이후 처음으로 REXX/CICS를 입력하면 CICSPROF라는 시스템 사용자 프로파일 exec가 실행됩니다. exec에는 모든 사용자가 실행해야 하는 설정 지시사항이 포함되어 있습니다. CICSPROF도 사용자 프로파일을 호출합니다.

사용자 프로파일은 사용자 작성 및 유지보수한 exec입니다. 프로파일을 사용하면 REXX/CICS 환경을 사용자 정의할 수 있습니다(예: 경로 설정, 검색 키 변경, 다른 exec 호출). 이 프로파일은 개인 RFS 디렉토리에 있어야 합니다.

#### 관련 참조

[110 페이지의 『CICSTART 멤버 업데이트』](#)

hlq.CICSTS56.REXX.SCICEXEC 데이터 세트의 CICSTART 멤버는 REXX/CICS 환경의 기본 정의를 포함합니다. CICS 시스템이 시작한 후 CICS/REXX 프로그램을 사용하는 첫 번째 트랜잭션이 실행되면 CICSTART가 실행됩니다.

### 권한 부여된 MVS PDS REXX 라이브러리

ddnames CICAUTH 및 CICEXEC에 할당되어 있는 모든 MVS 파티션된 데이터 세트는 REXX/CICS 권한 부여된 라이브러리로 간주됩니다.

여러 데이터 세트가 함께 연결되어 있으면 연결 순서대로 검색합니다. CICSTART exec가 CICEXEC에 있습니다.

- 사용자가 REXX/CICS PATH 명령을 사용하여 권한 부여되지 않은 고유 라이브러리를 동적으로 할당할 수 있습니다.
- 임의 사용자가 CICEXEC 데이터 세트로부터 exec를 실행할 수 있고 CICEXEC에서 로드된 모든 exec는 REXX/CICS 권한 부여된 명령을 사용할 수 있습니다.
- CICEXEC JCL DD문의 데이터 세트에서는 변수가 차단되어야 합니다.

### 권한 부여된 사용자 정의

사용자는 AUTHUSER 명령을 사용하여 권한 부여된 것으로 지정될 수 있습니다.

CICSTART exec 또는 CICSTART exec에서 실행된 exec에 모든 AUTHUSER 명령을 두는 것이 좋습니다. [110 페이지의 『CICSTART 멤버 업데이트』](#)의 내용을 참조하십시오.

## 시스템 옵션 설정

시스템 옵션은 REXX/CICS SETSYS 명령을 사용하여 지정됩니다.

CICSTART exec에 시스템 전체 SETSYS 명령을 두는 것이 좋습니다. [110 페이지의 『CICSTART 멤버 업데이트』](#)의 내용을 참조하십시오.

## REXX 파일 시스템(RFS) 파일 풀 정의

FILEPOOL 명령을 사용하여 파일을 정의하고 초기화하며 RFS 파일 풀에 추가합니다. RFS 파일 공유 권한을 부여하는 데 RFS AUTH 명령을 사용하십시오.

- FILEPOOL DEFINE 명령을 사용하여 RFS 파일 풀을 정의하십시오.
- FILEPOOL FORMAT 명령을 사용하여 각 파일 풀의 첫 번째 파일을 초기화하십시오.
- FILEPOOL ADD 명령을 사용하여 RFS 파일 풀에 VSAM 파일을 추가하십시오.

### RFS 파일 공유 권한 부여

RFS AUTH 명령을 사용하여 파일 공유 권한을 지정하십시오.

일반적으로 사용자가 소유한 자원을 공유할 수 있습니다. 권한 부여된 REXX/CICS 사용자로서 작성한 RFS 디렉토리를 공유하는 권한을 지정할 수 있습니다.

## CICSTART의 PLT 항목 작성

CICS 프로그램 로드 테이블(PLT) 항목을 작성함으로써 CICREXD 또는 CICREXR 프로그램을 호출하여 CICS 시스템 초기화 바로 후에 CICSTART exec를 실행할 수 있습니다.

그렇지 않으면 리전 시동 후 첫 번째 REXX/CICS 사용자를 사용하면 CICSTART exec가 실행됩니다.

## 보안 엑시트

교체 가능 보안 엑시트 CICSECX1 및 CICSECX2에 대해 설명합니다. IBM은 사용자 정의하거나 대체할 수 있는 샘플 어셈블러 엑시트를 제공합니다.

이 절에는 제품별 프로그래밍 인터페이스 정보가 포함되어 있습니다.

**참고:** 이러한 엑시트는 REXX/CICS와 동일한 리전에 있어야 합니다(예: DPL(Distributed Program Link) 사용은 허용되지 않음).

### CICSECX1

CICSECX1은 MVS 데이터 세트 액세스 보안 엑시트입니다. 이 엑시트는 EXEC CICS LINK에서 호출되며 매개변수는 COMMAREA에 전달됩니다.

#### 매개변수



**주의:** 이 주제에는 제품별 프로그래밍 인터페이스와 관련 지침 정보가 포함되어 있습니다.

엑시트에 입력 시 COMMAREA에 다음이 포함됩니다.

매개변수	바이트 수	데이터 유형	설명
1	4	전자	리턴 코드
2	8	문자	CICS 사인온 ID
3	4	문자	요청된 기능
4	3		IBM 사용을 위해 예약됨
5	44	문자	MVS 데이터 세트

## 리턴 코드

0

허용된 기능 요청

4

거부된 기능 요청

## 기능 ID

A

ALLOCATE REQUEST

E

EXPORT request

F

FREE REQUEST

I

IMPORT 요청

## CICSECX2

CICSECX2는 REXX 파일 시스템 액세스 보안 엑시트입니다.

### 매개변수



**주의:** 이 주제에는 제품별 프로그래밍 인터페이스와 관련 지침 정보가 포함되어 있습니다.

엑시트에 입력 시 COMMAREA에 다음이 포함됩니다.

매개변수	바이트 수	데이터 유형	설명
1	4	전자	리턴 코드
2	8	문자	CICS 사인온 ID
3	1	문자	요청된 기능
4	3		IBM 사용을 위해 예약됨
5	4	전자	완전한 RFS 파일 ID 문자열의 주소
6	4	전자	RFS 파일 ID 문자열의 디렉토리 경로 길이

## 리턴 코드

0

허용된 기능 요청

0이 아님

권한이 부여되지 않음

## 기능 ID

A

변경

R

읽기

U

갱신



---

## 제 13 장 성능 고려사항

클라이언트/서버 지원은 성능에 상당한 이점을 제공하는 REXX 기능입니다. 이 기능을 사용하면 중첩된 REXX exec가 아니라 서버 REXX exec를 자주 사용하여 애플리케이션 기능을 제공할 수 있습니다. 이러한 서버의 성능 특징은 더 효과적으로 관리할 수 있습니다. 중첩된 exec와 비교하여 서버 exec의 장점은 서버 exec의 경우 시작한 다음 종료하기 전에 여러 클라이언트 요청을 처리할 수 있다는 것입니다. 따라서 경로 길이가 짧고 응답 시간이 향상되며 종종 시스템 자원을 덜 사용합니다. 추가 정보는 [상위 레벨의 고유하고 투명한 REXX 클라이언트 인터페이스](#)의 내용을 참조하십시오.

일반적인 CICS 애플리케이션 프로그래밍과 달리 EXEC CICS SUSPEND 명령을 실행하지 않아도 됩니다. 왜냐하면 REXX/CICS에서 일정 간격으로 프로그램을 자동으로 일시중단하여 처리하기 때문입니다.

REXX/CICS exec는 VSAM 기반 REXX 파일 시스템 파일 또는 MVS 파티션 데이터 세트에 있을 수 있습니다. CICS 리전 시동 시에 CICS에서 특정 MVS 파티션 데이터 세트를 할당합니다. REXX PATH(데이터 세트 이름이 지정된 경우에만), IMPORT, EXPORT 및 ALLOC 명령은 SVC 99를 사용하여 MVS 파티션 데이터 세트를 동적으로 할당할 수 있습니다. SVC 99를 과도하게 사용하면 CICS 리전의 성능이 저하될 수 있습니다.



## 제 14 장 보안

REXX 트랜잭션은 CICS 트랜잭션 보안을 사용하여 제어될 수 있습니다. REXX 트랜잭션은 실행 중인 CICS 리전의 특성에 따라 광범위하게 사용하거나 몇몇으로만 사용을 제한할 수 있습니다.

REXX/CICS는 CICS 제공 CECI(Command Level Interpreter Transaction)의 고급 버전으로 볼 수 있습니다. REXX 트랜잭션(REXX exec를 실행하는 데 사용)은 CECI 트랜잭션과 마찬가지로 CICS 트랜잭션 보안을 사용하여 제어할 수 있습니다.

**참고:** REXX 트랜잭션은 기존 REXX exec를 실행하는 데는 필요하지 않지만, 사용자나 프로그래머가 REXX exec를 작성하거나 수정한 다음 테스트하는 기능을 원하면 필요합니다.

### REXX/CICS는 다중 트랜잭션 ID를 지원합니다.

REXX/CICS는 REXX 이외의, 트랜잭션 ID(TRANID)를 REXX/CICS 지원 프로그램과 연관시키는 기능을 지원합니다.

이 경우 실행되는 REXX exec의 이름은 이전 DEFTRNID 명령을 통해 판별됩니다. 그러면 여전히 exec별로 REXX에서 트랜잭션 보안을 사용할 수 있습니다.

### REXX/CICS 파일 보안

RFS 디렉토리 레벨에서의 액세스는 RFS AUTH 명령 및 RFS 대체 가능 보안 엑시트로 제어합니다.

CICAUTH ddname으로 액세스하는 권한 부여된 exec에 대한 액세스는 DEFCMD 및 DEFSCMD 명령의 AUTH 매개변수를 사용하여 제어합니다.

동적으로 할당된 데이터 세트에 대한 액세스는 데이터 세트 대체 가능 보안 엑시트를 통해 제어합니다.

### REXX/CICS 명령 레벨 보안

명령 레벨 보안은 CICS(및 기타 제품 또는 시스템) 기능에 대한 액세스를 제어하는 데 사용할 수 있습니다.

일부 상황에서, 현재 소프트웨어 연습은 CICS 자원 보안에만 의존하는 유효성을 제한합니다. 추가 보안 제어를 위해 REXX/CICS는 명령 레벨 보안의 개념을 사용하여 설계되었습니다. REXX/CICS의 대부분 기능은 명령으로 액세스되므로 명령 레벨 보안을 사용하여 CICS(및 기타 제품 또는 시스템) 기능에 대한 액세스를 제어할 수 있습니다. 예를 들어 VSAM 파일 액세스는 READ, WRITE 및 REWRITE 명령을 통해 수행합니다.

REXX/명령 레벨 보안은 DEFSCMD 및 DEFCMD AUTH 매개변수를 통해 제어하거나 권한 부여된 REXX/CICS 라이브러리 지원을 프로비저닝하여 제어합니다.

명령 실행 보안은 특정 REXX/CICS 명령 또는 명령 키워드의 사용을 제어합니다. 일반적으로 특정 명령(또는 명령 옵션)을 권한 부여된 것으로 지정하여 수행합니다. 이러한 명령 지정은 DEFCMD 및 DEFSCMD 명령으로 수행합니다. 권한 부여된 명령이 제대로 실행되려면 다음 조건 중 하나가 적용되어야 합니다.

1. 명령은 CICS 시동 JCL 프로시저의 ddname CICAUTH 또는 CICEXEC에 할당된 MVS PDS에서 로드된 exec를 통해 실행해야 합니다.
2. 명령은 권한 부여된 사용자가 실행해야 합니다. AUTHUSER 명령을 통해 사용자에게 권한을 부여할 수 있습니다.

### REXX/CICS 권한 부여된 명령 지원

REXX/CICS 명령은 REXX/CICS 시스템 관리자에서 권한 부여된 것으로 식별될 수 있습니다. 권한 부여된 명령은 권한 부여된 REXX/CICS 사용자가 실행하거나 권한 부여된 REXX/CICS 라이브러리에서 로드된 exec에서만 성공적으로 실행될 수 있습니다.

권한 부여된 REXX/CICS 사용자만 CICAUTH 권한 부여된 exec 라이브러리에 액세스할 수 있습니다. 모든 사용자가 CICEXEC 권한 부여된 라이브러리에서 exec를 실행할 수 있습니다. 모든 사용자가 권한이 없는 CICUSER 라이브러리에서 exec를 실행할 수 있습니다. 권한 부여된 사용자는 권한 부여된 기존 사용자나 권한 부여된 exec를 통해 정의할 수 있습니다. REXX/CICS 초기화 시(CICS를 다시 시작한 후 첫 번째 REXX/CICS 트랜잭션 시) 호

출된 REXX/CICS CICSTART exec에는 자동으로 권한이 부여됩니다. 여기서 권한 부여된 사용자와 라이브러리를 정의하는 것이 적합합니다.

REXX/CICS 라이브러리에 대한 액세스는 쉽게 제어할 수 있으므로, CICS 프로덕션 프로그램 라이브러리에 대한 액세스를 제어하는 데 적합합니다. 사이트에서 민감하다고 생각하는 모든 명령(예: READ, WRITE 및 DELETE)은 프로덕션 리전에서 권한 부여된 것으로 정의할 수 있습니다. 즉, 권한 부여된 사용자만 권한 부여된 명령을 실행한 exec를 작성할 수 있고, 모든 사용자가 권한 부여된 명령을 포함하는 exec를 호출할 수 있는지 아니면 기타 권한 부여된 사용자만 호출할 수 있는지 결정할 수 있습니다.

**참고:** CICS START, LINK 및 XCTL 명령을 REXX/CICS 권한 부여된 명령으로 재정의하여 REXX/CICS exec가 외부 API에 액세스하는 기능을 제어할 수 있습니다.

## 보안 정의

일반 사용자, 권한 부여된 사용자, 권한 부여된 명령, 권한 부여된 exec 및 시스템 라이브러리와 같은 REXX/CICS의 보안 정의에 대해 설명합니다.

### REXX/CICS 일반 사용자

AUTHUSER 명령을 통해 권한 부여된 것으로 정의되지 않은 REXX/CICS 사용자는 REXX/CICS 권한 부여된 명령을 사용할 수 없습니다. 그러나 이러한 사용자는 사용자 명령(DEFECMD 명령)과 exec를 정의, 쓰기, 변경 및 사용할 수 있습니다. 또한 CICEXEC 라이브러리에 있는 REXX/CICS 권한 부여된 exec를 사용할 수 있습니다(그러나 정의, 작성 또는 변경할 수 없음).

개별 사용자의 정보는 지정된 사용자 ID별로 REXX/CICS 환경에서 유지보수합니다. 각 사용자는 유일하게 식별되어야 하고 각 사용자가 한 번만 REXX/CICS 환경에만 사인온해야 합니다. 동시에 작동하는 동일한 사용자 ID를 갖는 두 사용자는 비정상적인 결과를 만들 수 있습니다.

### REXX/CICS 권한 부여된 사용자

권한 부여된 사용자는 AUTHUSER 명령으로 정의되며, 권한 부여된 REXX/CICS 명령(AUTH 옵션을 지정하여 DEFECMD 또는 DEFSCMD 명령으로 정의된 명령)을 사용할 수 있습니다.

### REXX/CICS 권한 부여된 명령

권한 부여된 명령은 권한 부여된 사용자 또는 권한 부여된 exec를 통해서만 사용될 수 있는 REXX/CICS 명령입니다. 권한 부여된 명령은 AUTH 옵션이 지정된 DEFECMD 또는 DEFSCMD 명령을 사용하여 정의합니다.

### REXX/CICS 권한 부여된 exec

권한 부여된 exec는 ddname CICEXEC 또는 CICAUTH에서 로드된 프로그램(exec)이고 권한 부여된 것으로 간주됩니다. 즉, 이러한 프로그램은 권한 부여된 REXX/CICS 명령을 사용할 수 있습니다. CICEXEC 권한 부여된 프로그램에는 모든 REXX/CICS 사용자가 액세스할 수 있지만, CICAUTH 권한 부여된 프로그램에는 권한 부여된 사용자만 액세스할 수 있습니다.

### REXX/CICS 시스템 라이브러리

REXX 언어로 작성된 권한 부여된 모든 명령은 ddname CICAUTH와 연결되어 있는 MVS 파티션된 데이터 세트에서 로드해야 합니다. IBM과 고객(또는 벤더) 둘 다 제공할 수 있습니다.

권한 부여된 모든 exec는 ddname CICEXEC 또는 CICAUTH와 결합되어 있는 MVS 파티션된 데이터 세트에서 로드해야 합니다. IBM과 고객(또는 벤더) 둘 다 제공할 수 있습니다.

권한 부여되지 않았지만 모든 REXX/CICS 사용자가 공유 중인 사용자 exec는 ddname CICUSER에 할당되거나 연결된 MVS 파티션된 데이터 세트에 둘 수 있습니다.

### 참고:

1. DEFECMD 또는 DEFSCMD의 AUTH 옵션 자체는 권한 부여된 명령 옵션입니다. 즉, 발행하는 사용자가 권한 부여된 사용자이거나 권한 부여된 라이브러리에서 로드된 exec에서 발행하는 경우에만 AUTH를 사용할 수 있습니다.



2. EXECLOAD 및 EXECDROP 명령은 권한이 부여되어 있습니다. 따라서 권한 부여된 사용자나 exec만 권한 부여된 하위 라이브러리에서 exec를 EXECLOAD할 수 있습니다.



---

## 제 3 부 CICS Transaction Server용 REXX: 참조

REXX/CICS 또는 CICS Transaction Server용 REXX에서는 애플리케이션 개발, 사용자 정의, 프로토타입 적성 및 프로시저에 대한 고유 REXX 기반 언어 환경을 연관된 런타임 기능과 함께 제공합니다.

REXX/CICS는 모든 지원되는 CICS Transaction Server 릴리스에서 실행됩니다.

이 참조는 CICS Interpreter용 REXX(지금부터 해석기 또는 언어 프로세서로 참조됨) 및 REstructured eXtended eXecutor(REXX라고 함) 언어를 설명합니다. 이 참조는 경험이 있는 프로그래머 특히, 블록 구조화된 고급 언어(예: PL/I, Algol 또는 Pascal)를 사용한 경험이 있는 프로그래머를 대상으로 합니다.

설명에는 언어의 사용 및 구문과 프로그램이 REXX/CICS 해석기에서 실행되고 있는 동안 언어 프로세서가 언어를 "해석"하는 방식이 포함됩니다. 이 참조는 다음도 설명합니다.

- REXX 및 언어 프로세서와 인터페이스할 수 있도록 하는 프로그래밍 서비스.
- REXX 처리를 사용자 정의할 수 있도록 하는 서비스 및 언어 프로세서가 스토리지 및 I/O 요청과 같은 시스템 서비스에 액세스하고 사용하는 방식을 사용자 정의.



## 제 15 장 제품 기능의 개요

REXX/CICS의 제품 기능에 대해 설명합니다.

다음 제품 기능을 설명합니다.

- [127 페이지의 『REXX/CICS에서의 SAA 레벨 2 REXX 언어 지원』](#)
- [127 페이지의 『REXX exec의 해석적 실행에 대한 지원』](#)
- [127 페이지의 『REXX exec 및 데이터용 CICS 기반 텍스트 편집기』](#)
- [127 페이지의 『REXX exec 및 데이터용 VSAM 기반 파일 시스템』](#)
- [128 페이지의 『EXEC CICS 명령에 대한 동적 지원』](#)
- [128 페이지의 『CEDA 및 CEMT 트랜잭션 프로그램에 대한 REXX 인터페이스』](#)
- [128 페이지의 『상위 레벨 클라이언트/서버 지원』](#)
- [128 페이지의 『REXX로 작성된 명령에 대한 지원』](#)
- [128 페이지의 『REXX 명령의 명령 정의』](#)
- [128 페이지의 『시스템 프로파일과 사용자 프로파일 exec에 대한 지원』](#)
- [128 페이지의 『가상 스토리지의 공유 exec』](#)
- [128 페이지의 『Db2/SQL 인터페이스』](#)

### REXX/CICS에서의 SAA 레벨 2 REXX 언어 지원

REXX/CICS는 현재 REXX 언어 레벨 3.48이고 스트림 I/O 및 REXX 언어 프로세서 종료를 제외한 모든 SAA(Systems Application Architecture®) REXX 레벨 2 기능을 제공합니다.

### REXX exec의 해석적 실행에 대한 지원

REXX exec의 해석적 실행은 REXX exec을 먼저 컴파일하지 않고도 이를 작성하고 실행할 수 있는 기능을 제공합니다. 해석기의 사용은 매우 생산적인 개발, 사용자 정의, 프로토타입 작성 및 명령 목록(CLIST) 처리 환경을 제공합니다. 해석기는 하나의 통합된 패키지로 빠른 개발 주기, 소스 레벨 대화식 디버그 및 고유 CICS 기반 개발 환경을 제공하기 때문입니다.

**참고:** REXX exec은 CICS에서 지원되는 모든 언어로 작성된 CICS 프로그램 및 트랜잭션을 자유롭게 호출할 수 있습니다.

### REXX exec 및 데이터용 CICS 기반 텍스트 편집기

TSO ISPF/PDF 및 VM/CMS XEDIT 편집기와 유사한 고유 CICS 텍스트 편집기는 REXX/CICS의 일부로 제공되므로 CICS에서 CICS 기반 애플리케이션 플랫폼으로부터 직접 exec(및 기타 데이터)을 작성하고 수정할 수 있습니다. 제공된 VSAM 기반 RFS(REXX File System)에 상주하는 파일 및 기존의 MVS(Multiple Virtual Storage) 파티션된 데이터 세트에 있는 파일에 대해 편집 지원이 제공됩니다.

**참고:** SVC 99는 PATH, IMPORT, EXPORT, ALLOC 및 편집기 GETPDS 명령에 지정된 파티션된 데이터 세트를 동적으로 할당하는 데 사용됩니다. 기타 파티션된 데이터 세트(ddnames CICAUTH, CICEXEC 및 CICUSER에 연결된 데이터 세트)는 리전 시동 시 할당됩니다. 이 기술은 주로 시스템 성능에 대한 영향을 최소화하기 위해 파일 마이그레이션 용도로 사용하는 것이 좋습니다.

### REXX exec 및 데이터용 VSAM 기반 파일 시스템

REXX/CICS에는 RFS(REXX File System), 계층 구조 형식으로 구성되고 AIX®(Advanced Interactive Executive)와 유사한 상위 레벨 파일 시스템 및 VM 공유 파일 시스템이 포함됩니다. RFS는 각 REXX 사용자에게 exec 및 데이터를 저장할 파일 시스템을 제공합니다. 파일 목록 유틸리티는 이 파일 시스템에 대한 작업을 지원하고, 텍스트 편집기는 이 파일 시스템의 멤버 편집을 지원하며, 실행될 exec은 이 파일 시스템으로부터 로드됩니다. 이 파일 시스템은 성능, 보안 및 이식성을 이유로 VSAM을 기반으로 합니다.

## EXEC CICS 명령에 대한 동적 지원

대부분의 EXEC CICS 애플리케이션 프로그래밍 명령에 대한 지원은 REXX/CICS에 포함됩니다. 이는 동적 인터페이스입니다(EXEC CICS 명령 변환의 사전 처리 단계는 필요하지 않음). 이 지원은 ADDRESS CICS 명령 환경의 추가를 통해 제공됩니다.

## CEDA 및 CEMT 트랜잭션 프로그램에 대한 REXX 인터페이스

이 인터페이스를 사용하면 후속 출력이 터미널에 표시되는 대신 REXX 변수로 배치되는 CEDA 및 CEMT 명령을 REXX exec에서 쉽게 발행할 수 있습니다. 이 인터페이스는 여러 CICS 관리 및 조작 활동의 자동화를 용이하게 하고 프로그래머에게 도움이 됩니다.

## 상위 레벨 클라이언트/서버 지원

REXX/CICS는 REXX exec이 클라이언트 역할을 할 수 있게 하는 기능(REXX/CICS 서버에 대한 요청 작성)을 제공하고 REXX exec이 서버 역할을 할 수 있게 하는 기능(REXX/CICS 클라이언트로부터의 요청을 대기 및 처리하는 기능 포함)을 제공하여 REXX exec에 통합된 클라이언트/서버 지원을 제공합니다.

REXX/CICS 서버가 클라이언트로부터의 요청을 대기하고(WAITREQ) 클라이언트 REXX 변수의 콘텐츠를 검색하고(C2S) 설정(S2C)할 수 있도록 하는 REXX/CICS 기능이 제공됩니다.

**참고:** 서버는 클라이언트의 중첩된 exec으로 실행되지 않는 대신 병렬 엔티티로 실행됩니다.

서버는 ASI(Automatic Server Initiation)를 사용하여 첫 번째 요청 수신 시 자동으로 시작됩니다.

## REXX로 작성된 명령에 대한 지원

REXX/CICS는 사용자가 REXX로 새 REXX/CICS 명령을 작성할 수 있도록 하는 기능을 제공합니다. 이러한 명령은 중첩된 REXX exec으로 작동하지 않고 중첩된 REXX exec과는 달리 명령을 발행한 사용자 exec에서 REXX 변수의 값을 가져오고 설정하는 기능을 가집니다. 따라서 REXX로 작성된 명령은 Assembler나 다른 언어로 작성된 명령과 유사한 기능을 가질 수 있습니다. 또한 명령은 시스템 개발의 속도를 높이기 위해(빌딩 블록 구조에서) REXX로 빠르게 작성할 수 있고 나중에 성능 요구사항에서 요구되는 경우 Assembler(또는 다른 CICS 지원 언어)로 선택적으로 재작성할 수 있습니다.

## REXX 명령의 명령 정의

REXX/CICS에는 시스템 관리자 및 사용자가 시스템 전체 또는 사용자별 기반으로 새 REXX 명령을 동적으로 정의할 수 있도록 하는 기능이 포함되어 있습니다. REXX는 다른 제품, 애플리케이션 및 시스템 서비스와 명확하게 인터페이스할 수 있습니다. 신규 또는 기존의 명령에 대한 명령 정의 기능을 제공하는 목적은 REXX의 사용을 통해 다양한 제품 및 서비스의 빠르고 일관적인 상위 레벨 통합을 용이하게 하는 것입니다. REXX 명령 정의는 REXX/CICS DEFCMD 및 DEFSCMD 명령을 사용하여 구현됩니다.

## 시스템 프로파일과 사용자 프로파일 exec에 대한 지원

REXX/CICS 시스템 및 사용자 환경 사용자 조정을 용이하게 하기 위해서 REXX/CICS는 CICSTART, CICS PROF 및 사용자 PROFILE exec(존재하는 경우)을 실행하려고 시도합니다. CICSTART는 시스템 프로파일 exec(STARTUP 프로파일)이고 CICS 시스템이 다시 시작된 후 첫 번째 사용자 exec이 실행되기 전에 발행됩니다. CICS PROF는 시스템 사용자 프로파일 exec이고 사용자가 CICS 시스템이 다시 시작된 이후 처음으로 REXX/CICS를 입력하는 경우 발행됩니다. CICS PROF는 또한 사용자 PROFILE을 호출합니다.

## 가상 스토리지의 공유 exec

REXX/CICS는 가상 스토리지에서 REXX exec의 공유 사본을 지원합니다. 공유 exec은 REXX 애플리케이션의 대화식 응답 시간을 향상시키고 공유는 총 가상 스토리지 요구사항을 감소시킵니다. Exec은 EXECLOAD 명령을 사용하여 미리 로드할 수 있습니다. 공통 시스템 범위 exec은 CICSTART exec에서 EXECLOAD 명령의 배치를 통해 사전 로드하기 위한 좋은 후보입니다.

## Db2/SQL 인터페이스

REXX 프로그램은 SQL문 및 Db2 명령을 포함할 수 있습니다. 이러한 명령문은 동적으로 해석 및 실행됩니다. SQL문과 Db2 명령의 결과는 REXX 프로그램 내에서 사용할 REXX 변수에 배치됩니다.

## 제 16 장 구문 다이어그램을 읽는 방법

REXX 명령 구문은 다음 구조를 사용하여 설명됩니다.

- 구문 다이어그램은 왼쪽에서 오른쪽으로, 위에서 아래로, 행의 경로를 따라 읽으십시오.

>>- 기호는 명령문의 시작을 표시합니다.

--> 기호는 명령문 구문이 다음 행에서 계속됨을 표시합니다.

>-- 기호는 명령문이 이전 행에서 계속됨을 표시합니다.

-->< 기호는 명령문의 끝을 표시합니다.

완전한 명령문 외 구문 단위의 다이어그램은 >-- 기호로 시작되고 --> 기호로 끝납니다.

- 필요한 항목은 가로 행(기본 경로)에 있습니다.

▶▶ STATEMENT — *required\_item* ▶▶

- 선택적 항목은 기본 경로 아래에 있습니다.

▶▶ STATEMENT — *optional\_item* ▶▶

- 둘 이상의 항목에서 선택할 수 있는 경우 세로 대체 세트로 표시됩니다.  
항목 중 하나를 선택해야 하는 경우 세트의 한 항목은 기본 경로에 있습니다.

▶▶ STATEMENT — *required\_choice1*  
— *required\_choice2* ▶▶

선택이 선택적이면 전체 세트가 기본 경로 아래 있습니다.

▶▶ STATEMENT — *optional\_choice1*  
— *optional\_choice2* ▶▶

- 일련의 대체 중 한 항목이 기본값이면 나머지 선택사항은 기본 경로 아래에 표시되고 기본 항목은 기본 경로 위에 표시됩니다.

▶▶ STATEMENT — *default\_choice*  
— *optional\_choice*  
— *optional\_choice* ▶▶

- 기본 행 위에 있는 왼쪽 방향 화살표는 반복할 수 있는 항목을 표시합니다.

▶▶ STATEMENT — *repeatable\_item* ▶▶

스택 위 반복 화살표는 세트에서 항목을 반복할 수 있음을 표시합니다.

- 항목 주변에 있는 일련의 세로 막대는 단편 즉, 항목이 기본 다이어그램 뒤에 오는 세부사항에 표시되는 구문 다이어그램의 부분임을 표시합니다.

►► STATEMENT — fragment ►►

### **fragment**

►► expansion\_provides\_greater\_detail ►►

- 키워드는 대문자로 표시됩니다(예: PARM1). 표시된 대로 정확히 철자가 입력되어야 하지만 대문자 또는 소문자로 지정할 수 있습니다. 변수는 모든 소문자(예: *parm**x*)로 표시됩니다. 사용자가 제공하는 이름 또는 값을 나타냅니다.
- 구두점 표시, 소괄호, 산술 연산자 또는 그러한 기호가 표시되면 구문의 일부로 입력해야 합니다.



## 제 17 장 REXX 일반 개념

REXX(REstructured eXtended eXecutor) 언어는 명령 프로시저, 애플리케이션 프론트 엔드, 사용자 정의 매크로(예: 편집기 하위 명령), 사용자 정의 XEDIT 하위 명령, 프로토타입 작성과 개인 컴퓨팅에 특히 적합합니다.

REXX는 PL/I와 같은 일반 용도 프로그래밍 언어입니다. REXX에는 IF, SELECT, DO WHILE, LEAVE 및 유용한 기본 제공 함수와 같은 유용한 구조화 프로그래밍 명령어가 있습니다.

언어는 프로그램 형식에 제한이 두지 않습니다. 한 행에 둘 이상의 절이 있을 수 있거나 단일 절이 둘 이상의 행을 차지할 수 있습니다. 들여쓰기는 허용됩니다. 그러므로 구조를 강조화하는 형식으로 프로그램을 코딩하여 더 읽기 쉬울 수 있습니다.

모든 변수가 사용 가능한 스토리지에 일치하는 한, 변수의 값의 길이에 한계가 없습니다.

**구현 최대:** 스토리지에 대한 단일 요청도 16MB의 수정 한계를 초과할 수 없습니다. 이 한계는 제어 정보 외에 변수의 크기에 적용됩니다. 숫자 결과를 유지하도록 획득된 버퍼에도 적용됩니다.

기호(변수 이름)의 길이의 한계는 250자입니다.

구성 배열 및 다른 용도로 복합 기호를 사용할 수 있습니다. 예를 들면 다음과 같습니다.

```
NAME.Y.Z
```

Y 및 Z는 변수의 이름이거나, 상수 기호일 수 있습니다.

REXX 프로그램은 REXX 파일 시스템디렉토리 또는 MVS 파티션된 데이터 세트에 상주할 수 있습니다. REXX 프로그램에는 일반적으로 EXEC의 파일 유형이 있습니다.

언어 프로세서(해석기)는 REXX 프로그램을 실행합니다. 즉, 프로그램은 먼저(컴파일됨) 다른 양식으로 변환되지 않고, 행별과 단어별로 처리됩니다. 사용자에게 대한 장점은 프로그램이 구문 오류로 실패하면 오류 지점이 분명하게 표시된다는 것이며, 이는 어려움을 이해하고 수정하는 데 도움이 됩니다.

### 구조 및 일반 구문

REXX 프로그램은 주석으로 시작하는 것이 좋습니다. REXX 프로그램은 일련의 절에서 빌드됩니다.

REXX 프로그램은 주석으로 시작하는 것이 좋습니다. REXX/CICS에서는 REXX 프로그램이 주석으로 시작하지 않아도 됩니다. 그러나 이식성의 이유로 다음 예제에 표시된 대로 첫 번째 행에서 시작되고 단어 REXX를 포함하는 주석으로 각 REXX 프로그램을 시작하는 것이 좋습니다.

```
/* REXX program */
...
...
EXIT
```

그림 46. REXX 프로그램 ID의 사용 예제

REXX 프로그램은 다음으로 구성된 일련의 절에서 빌드됩니다.

- 0 또는 추가 공백(무시됨)
- 일련의 토큰(132 페이지의 『토큰』 참조)
- 0 또는 추가 공백(무시됨)
- 행 끝, 특정 키워드 또는 콜론(:)으로 나타낼 수 있는 세미콜론(;) 구분 기호.

개념상 각 절은 처리 전에 왼쪽에서 오른쪽으로 스캔되고 절을 구성하는 토큰이 식별됩니다. 이 단계에서 명령어 키워드가 인식되고 주석이 제거되며 여러 개의 공백(리터럴 문자열에서는 제외됨)은 하나의 공백으로 변환됩니다. 연산자 문자 및 특수 문자에 인접한 공백도 제거됩니다(132 페이지의 『토큰』 참조).

**구현 최대:** 절의 길이는 16K를 초과할 수 없습니다.

## 문자

문자는 데이터의 제어 또는 표현에 사용되는 정의된 요소 세트의 멤버입니다.

일반적으로 한 번의 키 입력으로 문자를 입력할 수 있습니다. 문자의 코드화된 표현은 디지털 양식으로 된 해당 표현입니다. 문자(예: 문자 A)는 해당 코드화된 표현 또는 인코딩과 다릅니다. 다양한 코드화된 문자 세트(예: ASCII 및 EBCDIC)는 문자 A에 대해 다른 인코딩을 사용합니다(각각 10진수 값 65 및 193). 이 정보는 별도로 명시된 경우를 제외하고 특정 문자 코드를 나타내기 위해서가 아니라 의미를 전달하기 위해 문자를 사용합니다. 예외는 문자와 해당 표현 사이에서 변환하는 특정 기본 제공 함수입니다. 함수 C2D, C2X, D2C, X2C 및 XRANGE는 사용 중인 문자 세트에 대한 종속성을 가집니다.

코드 페이지는 한 세트에 있는 각 문자에 대해 인코딩을 지정합니다. 다음에 유의하십시오.

- 일부 코드 페이지는 REXX가 올바른 것으로 정의하는 모든 문자(예: ¬, 논리 NOT 문자)를 포함하지는 않습니다.
- REXX가 올바른 것으로 정의하는 일부 문자는 코드 페이지에 따라 다른 인코딩을 가집니다(예: !, 느낌표).

2바이트 문자 세트 문자에 대한 정보는 [431 페이지](#)의 『제 33 장 2바이트 문자 세트(DBCS) 지원』의 내용을 참조하십시오.

## 설명

주석은 /\* 및 \*/로 구분된 일련의 문자입니다(하나 이상의 행). 모든 문자는 구분 기호 내에 허용됩니다.

주석은 필요한 구분 기호로 시작하고 끝나는 한 다른 주석을 포함할 수 있습니다. 중첩 주석으로 불립니다. 주석은 어디에도 있을 수 있으며 어떠한 길이도 가능합니다. 프로그램에 영향을 미치지 않지만 구분 기호로 역할을 수행합니다. 주석이 사이에 하나인 두 토큰은 단일 토큰으로 처리되지 않습니다.

```
/* This is an example of a valid REXX comment */
```

리터럴 문자열의 일부로 /\* 또는 \*/를 포함하는 코드 행 밖에 주석을 지정하면 특별히 주의하십시오. 다음 프로그램 세그먼트를 고려하십시오.

```
01  parse pull input
02  if substr(input,1,5) = '/*123'
03    then call process
04  dept = substr(input,32,5)
```

2와 3행에 대해 주석 처리하려면, 언어 프로세서가 중첩된 주석의 시작으로 리터럴 문자열 /\*123의 일부인 /\*를 해석하기 때문에 다음 변경사항이 부정확합니다. 일치하는 주석 끝(\*/)에 대해 검색하기 때문에 프로그램 나머지를 처리하지 않습니다.

```
01  parse pull input
02  /* if substr(input,1,5) = '/*123'
03    then call process
04  */ dept = substr(input,32,5)
```

/\* 또는 \*/을 포함하는 리터럴 문자열에 대해 연결을 사용하여 이 유형의 문제를 피할 수 있습니다. 2행은 다음과 같습니다.

```
if substr(input,1,5) = '/' || '*123'
```

다음과 같이 2와 3행에 대해 주석 처리할 수 있습니다.

```
01  parse pull input
02  /* if substr(input,1,5) = '/' || '*123'
03    then call process
04  */ dept = substr(input,32,5)
```

2바이트 문자 세트 문자에 관한 정보는 [431 페이지](#)의 『제 33 장 2바이트 문자 세트(DBCS) 지원』 및 [165 페이지](#)의 『OPTIONS』 명령어를 참조하십시오.

## 토큰

토큰은 절이 빌드되는 하위 레벨 구문의 장치입니다.

REXX로 작성된 프로그램은 토큰으로 구성됩니다. 공백이나 주석 또는 토큰 자체의 네이처로 분리됩니다. 토큰의 클래스는 다음과 같습니다.

#### 리터럴 문자열:

리터럴 문자열은 모든 문자를 포함하는 시퀀스이고 작은따옴표 ' 또는 큰따옴표 "로 구분됩니다. 연속되는 두 큰따옴표 ""를 사용하여 큰따옴표 표시로 구분된 문자열에 " 문자를 표시합니다. 마찬가지로, 연속되는 두 작은따옴표 ''를 사용하여 작은따옴표 표시로 구분된 문자열에 ' 문자를 표시합니다. 리터럴 문자열은 상수이고 처리 시 콘텐츠는 수정될 수 없습니다.

문자가 없는 리터럴 문자열(즉, 0 길이의 문자열)은 *null* 문자열이라고 합니다.

다음은 유효한 문자열의 예입니다.

```
'Fred'
"Don't Panic!"
'You shouldn't'      /* Same as "You shouldn't" */
''                  /* The null string      */
```

- (가 바로 다음에 오는 문자열이 함수 이름으로 간주됩니다.
- 기호 X 또는 x가 바로 다음에 오는 문자열이 16진 문자열로 간주됩니다.
- 기호 B 또는 b가 바로 다음에 오는 문자열이 2진 문자열로 간주됩니다.

이러한 양식에 대한 설명이 뒤따릅니다.

**구현 최대:** 리터럴 문자열이 최대 250자까지 포함할 수 있습니다. 계산 결과의 길이는 사용 가능한 스토리지 양에 의해서만 제한됩니다. 자세한 정보는 [131 페이지의 『제 17 장 REXX 일반 개념』](#)의 노트를 참조하십시오.

#### 16진 문자열:

16진 문자열은 인코딩의 16진 표기법을 사용하여 표현된 리터럴 문자열입니다. 쌍으로 그룹화된 0개 이상의 16진 숫자(0-9, a-f, A-F) 순서입니다. 필요한 경우 16진 숫자의 짝수 번호를 작성하는 문자열 앞에 단일 선행 0이 가정됩니다. 숫자 그룹은 선택적으로 한 개 이상의 공백으로 구분되고, 전체 순서는 작은따옴표 또는 큰따옴표로 구분되며 바로 뒤에 기호 X 또는 x가 옵니다. (x 또는 X는 더 긴 기호의 일부일 수 없습니다.) 바이트 경계에만 표시될 수 있는 공백(및 문자열의 시작이나 끝에 없는)은 가독성을 지원합니다. 언어 프로세서는 이를 무시합니다. 16진 문자열은 제공된 16진 숫자를 압축하여 작성된 리터럴 문자열입니다. 16진 숫자를 압축하면 공백을 제거하며 'C1'X를 A로와 같이 동등한 문자로 16진 숫자의 각 쌍을 변환합니다.

프로그램에 문자를 직접 입력할 수는 없지만 16진 문자열을 사용하여 문자를 포함할 수 있습니다. 다음은 유효한 16진수 문자열의 예입니다.

```
'ABCD'x
"1d ec f8"X
"1 d8"x
```

**참고:** 16진수 문자열은 수에 대한 표시가 아닙니다. 오히려, 사용자가 해당 인코딩의 관점에서 문자를 설명할 수 있도록 하는 이스케이프 메커니즘입니다(그러므로 머신 종속적인). EBCDIC에서 '40'X는 공백에 대한 인코딩입니다. 모든 경우 양식 '.....'x의 문자열은 간단한 문자열에 대한 대체입니다. EBCDIC에서 'C1'x 및 'A'는 '40'x 및 공백과 동일하며 동일하게 처리되어야 합니다.

또한 어셈블러 언어로, 16진 수가 숫자 앞에 X로 표시되는 것에 유의하십시오. REXX는 이전에 설명된 형식으로 16진 수를 승인합니다. 이 정보는 두 방법으로 표시되는 16진 숫자를 표시할 수 있지만 REXX에서 16진수 문자열을 코드화할 때 숫자 다음에 X를 배치하십시오.

**구현 최대:** 16진수 문자열(공백이 제거된 문자열)의 팩형 길이가 250바이트를 초과할 수 없습니다.

#### 2진 문자열:

2진 문자열은 인코딩의 2진 표기법을 사용하여 표현된 리터럴 문자열입니다. 8바이트나 4 니블의 그룹에서 0 이상의 2진 숫자(0 또는 1)의 순서입니다. 첫 번째 그룹이 4자리수보다 적을 수 있습니다. 이 경우, 최대 세 개의 0 자리수가 처음 숫자의 왼쪽으로 가정되어 4개의 총 자리수를 작성합니다. 숫자 그룹은 선택적으로 하나 이상의 공백으로 분리되며 전체 순서는 일치되는 작은따옴표나 큰따옴표로 분리되며 바로 다음에 기호 b 또는 B가 옵니다. (b 또는 B는 더 긴 기호의 일부일 수 없습니다.) 바이트나 니블 경계에서만 표시될 수 있는 공백(문자열의 시작이나 끝이 아닌) 가독성을 지원합니다. 언어 프로세서는 이를 무시합니다.

2진 문자열은 제공된 2진 숫자를 압축하여 작성된 리터럴 문자입니다. 2진 숫자의 수가 8의 배수가 아닌 경우, 선행 0이 왼쪽에 추가되어 압축하기 전에 8의 배수로 작성합니다. 2진 문자열로 비트별 문자를 명시적으로 지정할 수 있습니다.

다음은 유효한 2진 문자열의 예입니다.

```
'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              /* == ''            */
```

**구현 최대:** 16진수 문자열(공백이 제거된 문자열)의 팩형 길이가 250바이트를 초과할 수 없습니다.

#### 기호:

기호는 다음 문자 세트에서 선택되는 문자의 그룹입니다.

- 영어 영문자(A-Z 및 a-z)

일부 코드 페이지는 소문자 영어 문자 a-z를 포함하지 않습니다.

- 숫자 문자(0-9)

- 문자 . ! ? 및 \_ (밑줄).

느낌표 문자 !의 인코딩은 사용 중인 코드 페이지에 따라 다릅니다.

- 2바이트 문자 세트(DBCS) 문자(X'41'-X'FE'). ETMODE는 이 문자에 대해 적용되어야 기호에서 유효합니다.

사용하기 전에 기호에서 소문자 영문자는 대문자로 변환됩니다(즉, 소문자 a-z에서 대문자 A-Z로).

다음은 유효한 기호의 예입니다.

```
Fred
Albert.Hall
WHERE?
<.H.E.L.L.O>          /* This is DBCS */
```

DBCS(Double-Byte Character Set) 문자에 관해 정보는 [431 페이지의 『제 33 장 2바이트 문자 세트\(DBCS\) 지원』](#)의 내용을 참조하십시오.

기호가 숫자나 마침표로 시작하지 않는 경우 변수로 사용할 수 있고, 값을 지정할 수 있습니다. 값으로 지정하지 않은 경우, 해당 값은 기호 자체의 문자이며 대문자로 변환됩니다(즉 소문자 a-z를 대문자 A-Z로). 숫자나 마침표로 시작하는 기호는 상수 기호이며 지정된 값일 수 없습니다.

다른 양식의 기호는 지수 형식으로 숫자 표시를 지원하도록 허용됩니다. 기호는 숫자(0-9)나 마침표로 시작하지만 순서 E 또는 e로 끝나며, 바로 다음에 선택적인 부호(- 또는 +), 바로 다음에 하나 이상의 숫자(다른 기호 문자가 다음에 올 수 없는)가 옵니다. 이 컨텍스트에서 부호는 기호의 일부이고 연산자는 아닙니다.

다음은 지수 표기법에서 유효한 수의 예입니다.

```
17.3E-12
.03e+9
```

**구현 최대:** 기호는 최대 250자까지 구성될 수 있습니다. 변수인 경우 해당 값은 사용 가능한 스토리지의 양으로만 제한됩니다. 자세한 정보는 [131 페이지의 『제 17 장 REXX 일반 개념』](#)의 노트를 참조하십시오.

#### 숫자:

더하기나 빼기 부호의 선택적 접두부가 있으며 소수점을 표시하는 선택적인 단일 마침표(.)를 포함하여 하나 이상의 10진수 숫자로 구성되는 문자열입니다. 기존 지수 표기법으로 10의 거듭제곱이 접미부로 숫자에 있을 수도 있으며(E, 대문자나 소문자), 다음에 선택적으로 더하기나 빼기 부호가 있으며, 그 다음으로 10의 거듭제곱을 정의하는 하나 이상의 10진수 숫자가 옵니다. 문자열이 수로 사용될 때마다 NUMERIC DIGITS 명령어에서 지정한 자릿수(기본이 9 자릿수)에 반올림이 발생할 수 있습니다. 수의 전제 정의는 [225 페이지의 『제 21 장 숫자와 산술 오퍼레이션』](#)의 내용을 참조하십시오.

숫자에 선행 공백(있는 경우 부호 앞뒤)이 있을 수 있으며 후행 공백이 있을 수 있습니다. 공백은 숫자의 자릿수 간 또는 지수 파트에 임베디드될 수 없습니다. 기호 또는 리터럴 문자열이 숫자일 수 있다는 점에 유의하십시오. 숫자는 변수의 이름일 수 없습니다.

다음은 유효한 숫자에 대한 예입니다.

```
12' -17.9'
127.0650
73e+128
' + 7.9E5 '
'0E000'
```

숫자를 둘러싼 따옴표와 함께 또는 따옴표 없이 숫자를 지정할 수 있습니다. 표현식에서 시퀀스 -17.9(따옴표 표시가 없는)는 단순한 숫자가 아닙니다. 양의 숫자가 다음에 오는 마이너스 연산자입니다(왼쪽에 조건이 없는 경우 접두부 마이너스일 수 있음). 조작의 결과는 숫자입니다.

정수는 0(또는 없음) 10진수 파트가 있고 언어 프로세서가 일반적으로 지수 표기법으로 표현하지 않는 숫자입니다. 즉 NUMERIC DIGITS의 현재 설정(기본값은 9) 앞의 자릿수에 숫자는 없습니다.

**구현 최대:** 지수 표기법으로 표현된 수의 지수가 최대 9 숫자를 가질 수 있습니다.

### 연산자 문자:

문자: + - \ / % \* | & = ~ > < 및 시퀀스 >= <= \> \< \= >< <> == \== // && || \*\* ~> ~< ~ = ~== >> << >>= \<< ~<< \>> ~>> <=< /= /===는 조작을 표시합니다(137 페이지의 『연산자』 참조). 이들 중 몇은 구문 분석 템플릿에서 사용되고 등호가 지정을 표시하는 데도 사용됩니다. 연산자 문자에 근접한 공백이 제거됩니다. 그러므로 다음은 의미에서 동일합니다.

```
345>=123
345 >=123
345 >= 123
345 > = 123
```

이러한 문자 중 일부는 모든 문자 세트에서 사용할 수 없을지도 모릅니다. 이 상황에서, 적절한 변환을 사용할 수 있습니다. 특히, 세로 막대(|) 또는 문자는 분할된 세로 막대로 표시됩니다.

언어에서 *not* 문자, ~는 백슬래시(\)와 동의어입니다. 사용 가능 여부 및 개인 취향에 따라 두 문자를 서로 바꿔 사용할 수 있습니다.

### 특수 문자

리터럴 문자열 밖에서 찾은 경우 연산자로부터 개별 문자와 함께 다음 문자는 특별한 의미가 있습니다.

```
, ; : ) (
```

이러한 문자는 특수 문자의 설정을 구성합니다. 모두 토큰 구분 기호로 동작하며 이것에 인접한 공백이 제거됩니다. 예외는 소괄호의 외부에 인접한 공백으로, 다른 특수 문자에도 근접한 경우에만 삭제됩니다(문자가 소괄호이고 공백이 또한 외부에 있지 않으면). 예를 들면, 언어 프로세서는 A (Z)에서 공백을 제거하지 않습니다. A(Z), 함수 호출에 해당되지 않은 연결입니다. 언어 프로세서는 (A)+(Z)와 동일하기 때문에 (A) + (Z)의 공백을 제거합니다.

다음 예는 절이 토큰으로 구성되는 방법을 표시합니다.

```
'REPEAT' A + 3;
```

6개의 토큰으로 구성됩니다.

- 리터럴 문자열('REPEAT')
- 공백 연산자
- 기호(A, 값을 가질 수 있음)
- 연산자(+)
- 두 번째 기호(3, 숫자와 기호임)
- 절 구분 기호(;)

A와 + 사이 및 +와 3 사이의 공백이 제거됩니다. 그러나 'REPEAT'와 A 사이의 공백 중 하나는 연산자로 남습니다. 그러므로 이 절은 다음과 같이 쓰여진 것처럼 처리됩니다.

```
'REPEAT' A+3;
```

## 내재적 세미콜론

절에서 마지막 요소는 세미콜론 구분 기호입니다. 언어 프로세서는 세미콜론을 뜻합니다. 단일 기호가 다음에 오는 경우 행 끝의 특정 키워드 다음 그리고 콜론 다음입니다.

한 행에 둘 이상의 절이 있을 경우에만 세미콜론을 포함하거나 마지막 문자가 쉼표인 명령어를 종료함을 의미합니다.

행 끝은 보통 절의 끝을 표시하므로 REXX는 최대한 행 끝에서 세미콜론을 표시합니다. 그러나 다음 예외가 있습니다.

- 행은 문자열의 중간에서 종료됩니다.
- 행은 주석의 중간에서 종료됩니다. 절이 다음 행에 계속됩니다.
- 마지막 토큰은 연속 문자(쉼표)이고 행이 주석의 중간에서 종료되지 않습니다.(주석이 토큰이 아니라는 점에 유의하십시오.)

정확한 컨텍스트에 있는 경우 REXX는 자동으로 콜론 다음의 세미콜론을 표시하며(단일 기호가 다음에 오는 경우, 레이블) 특정 키워드 다음에 표시합니다. 이 결과를 가지는 키워드는 ELSE, OTHERWISE 및 THEN입니다. 이 특수 경우는 현저하게 오타 오류를 줄입니다.

**참고:** 올바르게 인식될 수 없었기 때문에 주석 구분 기호, /\* 및 \*/를 형성하는 두 문자는 행 끝으로 분할되지 않아야 합니다(즉, / 및 \*는 다른 행에 표시되지 않아야 합니다). 표시된 세미콜론이 추가됩니다. 문자열 내 리터럴 따옴표를 형성하는 두 개의 연속적 문자는 이 행 끝 규칙도 따릅니다.

## 연속

다음 행 위에 절을 계속 이어가는 한 가지 방법은 쉼표를 사용하는 것이며, 이는 연속 문자로 참조됩니다.

쉼표는 기능상으로 공백으로 대체되므로 세미콜론을 표시하지 않습니다. 하나 이상의 주석이 행 끝 전에 연속 문자 다음에 올 수 있습니다. 연속 문자는 문자열의 가운데에 사용될 수 없거나 문자열 자체의 일부로 처리됩니다. 동일한 상황이 주석에 대해서 true입니다. 쉼표가 실행 추적에 남아 있다는 점에 유의하십시오.

다음 예는 절을 계속하기 위해 연속 문자를 사용하는 방법을 표시합니다.

```
say 'You can use a comma',  
  'to continue this clause.'
```

이 경우 다음이 표시됩니다.

```
You can use a comma to continue this clause.
```

## 표현식 및 연산자

REXX의 표현식은 일반적으로 원래 데이터와 다른 결과를 발생시키기 위해 다양한 방법으로 하나 이상의 데이터를 결합시키기 위한 일반 메커니즘입니다.

### 표현식

표현식은 조건에 대해 실행할 조작을 표시하는 하나 이상의 조작으로 배치된 하나 이상의 조건(리터럴 문자열, 기호, 함수 호출 또는 하위 표현식)으로 구성됩니다.

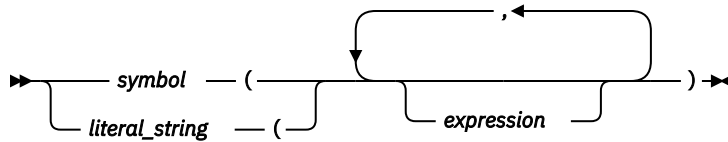
하위 표현식은 왼쪽 및 오른쪽 소괄호로 대괄호 처리된 표현식의 조건입니다.

조건에는 다음이 포함됩니다.

- (따옴표로 구분되는) 리터럴 문자열로, 상수입니다.
- 기호(따옴표 표시 없음)로 대문자로 변환됩니다. 숫자 또는 주기로 시작하지 않는 기호는 변수의 이름일 수 있습니다. 이 경우에 해당 변수의 값이 사용됩니다. 그렇지 않으면 기호는 상수 문자열로 처리됩니다. 기호는 복합일 수도 있습니다.



- 함수 호출(179 페이지의 『제 19 장 함수』 참조)로 다음 형식 중 하나입니다.



표현식의 평가는 왼쪽에서 오른쪽이며, 일반 대수적 방법으로 연산자 우선순위로 소괄호로 수정됩니다(140 페이지의 『소괄호와 연산자 우선순위』 참조). 평가 중 오류가 발생하지 않으면 표현식은 전체 평가됩니다.

모든 데이터는 "유형이 없는" 문자열 형태입니다(데이터는 2진, 16진 또는 배열과 같이 선언된 특정 유형이 아니기 때문). 그러므로 표현식이 평가받을 때, 결과는 문자열입니다. 조건과 결과(산술 및 논리식)은 *null* 문자열(0 길이의 문자열)일 수 있습니다. REXX는 결과의 최대 길이에 대해 제한사항을 부과하지 않습니다. 그러나 언어 프로세서에 사용 가능한 단일 스토리지 요청의 양에는 16MB의 제한이 있습니다. 자세한 정보는 131 페이지의 『제 17 장 REXX 일반 개념』의 노트를 참조하십시오.

## 연산자

연산자는 하나 또는 두 개의 항에 대해 수행될 더하기와 같은 오퍼레이션의 표현입니다.

다음 정보는 각 연산자(접두부 연산자는 제외됨) 두 항에 대해 작동하는 방식에 대해 설명하고 이러한 항은 기호, 문자열, 함수 호출, 중간 결과 또는 하위 표현식일 수 있습니다. 각 접두부 연산자는 그 뒤에 오는 항 또는 하위 표현식에 대해 작동합니다. 연산자 문자에 인접한 공백(및 주석)은 연산자에 영향을 미치지 않으므로 둘 이상의 문자에서 구성된 연산자에는 임베디드 공백 및 주석이 있을 수 있습니다. 또한 표현식에서 발생하지만 다른 연산자에 인접하지 않은 하나 이상의 공백도 연산자 역할을 합니다. 다음과 같이 4가지 유형의 연산자가 있습니다.

- 연결
- 산술
- 비교
- 논리

### 문자열 연결 연산자

연결 연산자는 두 번째 문자열을 첫 번째 문자열의 오른쪽 끝에 추가함으로써 두 개의 문자열을 결합하여 한 개의 문자열을 구성합니다.

연결은 중간 공백이 있거나 없이 발생할 수 있습니다. 연결 연산자는 다음과 같습니다.

#### (공백)

항 사이에 하나의 공백을 삽입하여 연결합니다.

||

중간에 공백을 삽입하지 않고 연결합니다.

#### (abuttal)

중간에 공백을 삽입하지 않고 연결합니다.

|| 연산자를 사용하여 공백 없는 연결을 강제 실행할 수 있습니다.

abuttal 연산자는 다른 연산자로 분리되지 않는 두 개의 항 사이에서 가정됩니다. 이는 두 개의 항이 리터럴 문자열 및 기호와 같이 구문상 구별될 때 또는 주석에 의해서만 분리될 때 발생할 수 있습니다.

## 예제

구문상 구별되는 항의 예제는 다음과 같습니다. Fred의 값이 37.4인 경우 Fred '%'는 37.4%로 평가됩니다.

변수 PETER의 값이 1인 경우 (Fred) (Peter)는 37.41로 평가됩니다.

EBCDIC에서 두 개의 인접한 문자열 중 하나는 16진이고 하나는 리터럴인 경우,

```
'c1 c2'x'CDE'
```

는 ABCDE로 평가됩니다.

:

다음 예제에는 내재된 **abuttal** 연산자가 없고 표현식이 올바르지 않습니다.

```
Fred/* The NOT operator precedes Peter. */¬Peter
```

그러나 다음 예제는 결과로 **abuttal**을 발생시키고 37.40으로 평가됩니다.

```
(Fred)/* The NOT operator precedes Peter. *//(¬Peter)
```

## 산술 연산자

다음 산술 연산자를 사용하여 유효한 수인(132 페이지의 『토큰』 참조) 문자열을 결합할 수 있습니다.

**+**  
추가

**-**  
빼기

**\***  
곱하기

**/**  
나누기

**%**  
정수 나누기(나눈 후 결과의 정수 파트를 리턴)

**//**  
나머지(결과가 음수일 수 있기 때문에 나눈 다음 모듈로가 아닌 나머지를 리턴)

**\*\***  
 거듭제곱(숫자를 정수 제곱함)

**접두부 -**  
빼기와 동일함: 0 - number

**접두부 +**  
더하기와 동일함: 0 + number

자릿수, 유효한 수의 형식 및 산술에 대한 연산 규칙의 세부사항은 225 페이지의 『제 21 장 숫자와 산술 오퍼레이션』의 내용을 참조하십시오. 산술 결과가 지수 표기법으로 표시되면, 반올림이 발생할 수 있습니다.

## 비교 연산자

비교 연산자는 두 항목을 비교하고 비교 결과가 true이면 값 1을 리턴하며 그렇지 않은 경우 0을 리턴합니다.

엄격한 비교 연산자 모두에 두배가 된 연산자를 정의하는 문자 중 하나가 있습니다. ==, \==, /= 및 ¬== 연산자는 두 문자열 사이의 완전 일치 여부를 테스트합니다. 두 문자열은 동일해야 하며(문자 대 문자) 동일 길이가 엄격하게 같은 것으로 간주되어야 합니다. 마찬가지로, >> 또는 <<와 같은 엄격한 비교 연산자는 비교 중인 문자열 중 하나를 채우지 않고 단순한 문자 대 문자 비교를 수행합니다. 두 문자열의 비교는 왼쪽에서 오른쪽까지입니다. 문자열이 더 짧거나 다른 선행 서브 문자열이 있는 경우, 다른 문자열보다 작거나 적습니다. 또한 엄격한 비교 연산자는 두 피연산자에 대한 숫자 비교를 수행하려고 시도하지 않습니다.

기타 모든 비교 연산자의 경우, 포함된 두 항목이 숫자이면 숫자 비교가 발생합니다(선행 0(영)이 무시되며 229 페이지의 『숫자 비교』의 내용 참조). 그렇지 않은 경우, 두 항목은 문자열로 처리됩니다(선행 및 후행 공백은 무시되며 더 짧은 문자열이 오른쪽의 공백으로 채워짐).

문자 비교와 엄격한 비교 조작은 모두 대소문자를 구분하며 정확한 조합 순서 모두는 구현을 위해 사용된 문자 세트에 따라 다릅니다. 예를 들면, EBCDIC 환경에서 소문자 영문자는 대문자에 선행하며 숫자 0-9는 모든 영문자보다 높습니다.

비교 연산자 및 조작은 다음과 같습니다.

**=**  
조건이 일치하면 true(수적으로 또는 채우기 등 실행 시)

**\=, ¬=, /=**  
조건이 일치하지 않으면 true(=와 반대)



>  
 보다 큼  
 <  
 보다 작음  
 ><  
 크거나 작음(같지 않음과 동일)  
 <>  
 크거나 작음(같지 않음과 동일)  
 >=  
 보다 크거나 같음  
 \<, ~<  
 보다 작지 않음  
 <=  
 보다 작거나 같음  
 \>, ~>  
 보다 크지 않음  
 ==  
 조건이 정확하게 일치하면 true(동일)  
 \==, ~==, /==  
 조건이 정확하게 일치하지 않으면 true(==와 반대).  
 >>  
 엄격하게 초과  
 <<  
 엄격하게 미만  
 >>=  
 엄격하게 이상  
 \<<, ~<<  
 엄격하게 보다 작지 않음  
 <<=  
 엄격하게 이하  
 \>>, ~>>  
 엄격하게 보다 크지 않음

**참고:** 언어에서 *not* 문자, ~는 백슬래시(\)와 동의어입니다. 가용성과 개인 환경설정에 따라 두 문자를 서로 바꿔 사용할 수 있습니다. 백슬래시는 다음 연산자에 표시될 수 있습니다. \ (접두부가 아님), \=, \==, \<, \>, \<< 및 \>>.

### 논리(부울) 연산자

문자열은 0인 경우 false 값을 가지고 1인 경우 true 값을 가집니다. 논리 연산자는 이러한 값(0 또는 1 이외의 값은 허용되지 않음)을 1개 또는 2개 사용하여 0 또는 1을 리턴합니다.

**&**

AND

두 항 모두 true이면 1을 리턴합니다.

**|**

포함적 OR

두 항 중 하나라도 true이면 1을 리턴합니다.

**&&**

배타적 OR

두 항 중 하나(그러나 둘 다는 아님)가 true이면 1을 리턴합니다.

접두부 \,~  
논리 NOT

부정; 1은 0이 되고 0은 1이 됩니다.

## 소괄호와 연산자 우선순위

표현식 평가는 왼쪽에서 오른쪽으로 이루어지며 소괄호 및 연산자 우선순위가 이를 수정합니다.

함수 호출을 식별하는 경우 이외에 괄호가 나타나면 항이 필요한 경우 소괄호 사이의 전체 하위 표현식이 즉시 평가됩니다.

다음 시퀀스가 표시되고 operator2가 operator1보다 높은 우선순위를 갖는 경우 하위 표현식(term2 operator2 term3)이 먼저 평가됩니다. 필요에 따라 동일한 규칙이 반복해서 적용됩니다.

```
term1 operator1 term2 operator2 term3
```

그러나 개별 항은 표현식의 왼쪽에서 오른쪽으로 평가됩니다(즉, 표시되는 순서대로). 우선순위 규칙은 *operations*의 순서에만 영향을 미칩니다.

예를 들어, \*(곱하기)는 +(더하기)보다 높은 우선순위를 가지므로 3+2\*5는 13으로 평가됩니다(왼쪽에서 오른쪽 평가가 엄격하게 발생한 경우의 결과인 25가 아님). 곱하기 전에 더하기가 발생하도록 강제 실행하려면 표현식을 (3+2)\*5로 다시 작성할 수 있습니다. 소괄호를 추가하면 처음 세 개 토큰이 하위 표현식이 됩니다. 마찬가지로 표현식 -3\*\*2는 9로 평가됩니다(-9 대신), 접두부 마이너스 연산자가 거듭제곱 연산자보다 높은 우선순위를 가지기 때문입니다.

연산자의 우선순위 순서는 다음과 같습니다.

1. + - ~ \ (접두부 연산자)
2. \*\* (거듭제곱)
3. \* / % //(곱하기 및 나누기)
4. + - (더하기 및 빼기)
5. (blank) || (abuttal) (공백을 사용하거나 사용하지 않고 연결)
6. 비교 연산자:
  - = > <
  - == >> <<
  - \= ~=
  - >< <>
  - \> ~>
  - \< ~<
  - \== ~==
  - \>> ~>>
  - \<< ~<<
  - >= >>=
  - <= <<=
  - /= /==
7. &(and)
8. | &&(or 또는 배타적 or)

## 예제

기호 A는 값이 3인 변수이고 DAY는 값이 Monday인 변수이고 기타 변수는 초기화되지 않습니다. 예제 표현식은 다음 평가가 있습니다.

```
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 */
```

**참고:** 마지막 예제에서 연산자가 >>가 아닌 >인 경우 응답이 다릅니다. '0E0000'이 지수 표기법에 유효한 숫자이므로 숫자 비교가 수행되고 따라서 '0E0000' 및 '000000'은 등가로 평가됩니다. 우선순위의 REXX 순서는 기존의 대수학 및 기타 컴퓨터 언어에서와 같으므로 일반적으로 어렵지 않습니다. 공통 표기법과는 다음 두 가지 차이점이 있습니다.

- 접두부 마이너스 연산자는 항상 거듭제곱 연산자보다 높은 우선순위를 가집니다.
- 거듭제곱 연산자(다른 연산자와 마찬가지로)은 왼쪽에서 오른쪽으로 평가됩니다.

예를 들어, 다음과 같습니다.

```
-3**2    == 9  /* not -9 */
-(2+1)**2 == 9  /* not -9 */
2**2**3  == 64 /* not 256 */
```

## 절 및 명령어

절은 null 절, 레이블, 명령어, 지정, 키워드 명령어와 명령으로 분할될 수 있습니다.

### NULL 절

공백 또는 주석, 또는 둘 다로만 구성되는 절은 *null* 절입니다. 완전히 무시됩니다(주석을 포함하는 경우를 제외하면 적절한 경우 추적됨).

**참고:** null 절은 명령어가 아닙니다. 예를 들어 IF 명령어에서 THEN 또는 ELSE 다음에 추가 세미콜론을 두는 것은 더미 명령어를 사용하는 것과 같지 않습니다(PL/I에서와 같이). NOP 명령어는 이 용도로 제공됩니다.

### 레이블

콜론이 다음에 오는 단일 기호로 구성되는 절은 레이블입니다. 이 컨텍스트에서의 콜론은 세미콜론(절 구분 기호)을 표시하므로 필요한 세미콜론은 없습니다. 레이블은 CALL 명령어, SIGNAL 명령어 및 내부 함수 호출의 대상을 식별합니다. 둘 이상의 레이블은 명령어 앞에 올 수 있습니다. 레이블은 null 절로 처리되고 디버깅을 지원하기 위해 선택적으로 추적될 수 있습니다.

임의 수의 연속적인 절은 레이블일 수 있습니다. 다른 절 전에 여러 레이블을 허용합니다. 복제 레이블이 허용되지만 제어는 프로그램에서 첫 번째 복제에만 전달됩니다. 나중에 발생하는 복제 레이블을 추적할 수 있지만 CALL, SIGNAL 또는 함수 호출의 대상으로 사용될 수 없습니다.

DBCS 문자를 사용할 수 있습니다. [431 페이지의 『제 33 장 2바이트 문자 세트\(DBCS\) 지원』](#)의 내용을 참조하십시오.

## 명령어

명령어는 언어 프로세서가 사용하는 일부 조치 과정을 설명하는 하나 이상의 절로 구성됩니다. 명령어는 지정, 키워드 명령어 또는 명령일 수 있습니다.

## 지정사항

*symbol=expression* 양식의 단일 절은 지정으로 알려진 명령어입니다. 지정은 변수에 (새) 값을 제공합니다. 142 페이지의 『지정 및 기호』의 내용을 참조하십시오.

## 키워드 명령어

키워드 명령어는 첫 번째 절이 명령어를 식별하는 키워드로 시작하는 하나 이상의 절입니다. 키워드 지시사항은 외부 인터페이스와 제어 플로우를 제어합니다. 일부 키워드 명령어는 중첩된 명령어를 포함할 수 있습니다. 다음 예제에서 DO 구성(DO, 다음에 오는 명령어 그룹 및 연관된 END 키워드)은 단일 키워드 명령어로 간주됩니다.

```
DO
  instruction
  instruction
  instruction
END
```

서브 키워드는 특정 일부 명령어의 내 예약된 키워드로, 예를 들어 DO 명령어의 기호 TO 및 WHILE입니다.

## 명령

명령은 표현식으로만 구성되는 절입니다. 표현식이 평가되고 결과가 일부 외부 환경에 명령 문자열로 전달됩니다.

## 지정 및 기호

변수는 값이 REXX 프로그램의 실행 동안 변경될 수 있는 오브젝트입니다. 변수의 값 변경 프로세스는 새 값으로 지정하는 것입니다.

변수의 값은 모든 문자를 포함할 수 있는 길이의 단일 문자열입니다.

새 값을 ARG, PARSE 또는 PULL 명령어, VALUE 내장 함수 또는 변수 풀 인터페이스로 지정할 수 있습니다. 변수의 값을 변경하는 가장 일반적인 방법은 지정 명령어 자체입니다. 다음 양식의 절이 지정에 사용됩니다.

```
symbol=expression;
```

*expression*의 결과가 등호 기호의 왼쪽에 기호에 이름 지정된 변수의 새 값이 됩니다. 현재, VM에서 *expression*을 생략하면 변수는 null 문자열로 설정됩니다. 그러나 null 문자열: *symbol*= ' '에 대해 명시적으로 변수를 설정하는 것이 좋습니다. 예:

```
/* Next line gives FRED the value "Frederic" */
Fred='Frederic'
```

변수의 이름을 지정하는 기호는 (0-9) 또는 마침표(.)로 시작할 수 없습니다.(변수 이름의 첫 번째 문자에 대해 이 제한사항 없이 숫자를 재정의할 수 있습니다. 예를 들어 3=4;는 3 값 4라는 변수를 제공합니다.)

항상 기호가 정의된 값을 가지기 때문에 값으로 지정되지 않더라도 표현식에서 기호를 사용할 수 있습니다. 값이 지정되지 않은 변수는 초기화되지 않습니다. 대문자로 변환된 해당 값이 기호 자체의 문자입니다(즉 소문자 a-z는 대문자 A-Z로). 그러나 복합 기호인 경우 해당 값은 기호의 파생된 이름입니다(143 페이지의 『복합 기호』 참조). 예:

```
/* If Freda has not yet been assigned a value, */
/* then next line gives FRED the value "FRED" */
Fred=Freda
```

REXX의 기호 의미는 해당 컨텍스트에 따라 다릅니다. 표현식에서 용어로(키워드가 아닌) 기호는 네 개 그룹(상수 기호, 단순 기호, 복합 기호 및 스텝) 중 하나의 그룹에 속합니다. 상수 기호는 새 값을 지정받을 수 없습니다. 이름이 단순 값에 해당하는 변수에 단순 기호를 사용할 수 있습니다. 어레이와 목록과 같이 보다 복잡한 변수의 콜렉션에 대해 복합 기호와 스텝을 사용할 수 있습니다.

## 상수 기호

상수 기호는 숫자(0-9) 또는 마침표(.)로 시작됩니다.

상수 기호의 값을 변경할 수 없습니다. 간단하게 기호의 문자로 구성되는 문자열입니다(즉, 소문자 알파벳 문자로 대문자로 변환됩니다).

다음은 상수 기호입니다.

```
77
827.53
.12345
12e5      /* Same as 12E5 */
3D
17E-3
```

## 단순 기호

단순 기호는 마침표를 포함하지 않으며 숫자(0-9)로 시작되지 않습니다.

기본적으로 그 값은 기호의 문자입니다(즉, 대문자로 변환됨). 기호에 값이 지정된 경우 기호는 변수에 이름을 지정하고 그 값은 해당 변수의 값입니다.

다음은 단순 기호입니다.

```
FRED
Whatagoodidea? /* Same as WHATAGOODIDEA? */
?12
<.D.A.T.E>
```

## 복합 기호

복합 기호가 이를 참조할 때 해당 이름 내 변수의 대체를 허용합니다.

복합 기호는 하나 이상의 마침표와 최소 두 개 이상의 다른 문자를 포함합니다. 숫자나 마침표로 시작할 수 없으며 복합 기호에 하나의 마침표만 있는 경우 마지막 문자일 수 없습니다.

이름은 스템(마침표의 그 부분까지 및 첫 번째 마침표 포함)로 시작합니다. 상수 기호, 간단한 기호 또는 null인 이름의 파트(마침표로 구분됨), 후미가 다음에 옵니다. 복합 기호의 파생된 이름은 기호의 스템으로, 대문자로 다음에 후미가 오며, 단순한 모든 기호가 해당 값으로 대체됩니다. 후미 자체는 문자, A-Z, a-z, 0-9 및 @ # \$ ¢ . ! ? 및 대문자로 구성될 수 있습니다. 후미의 값은 null 문자열 및 공백을 포함한 문자열을 포함하여 모든 문자열일 수 있습니다. 예를 들어, 다음과 같습니다.

```
taila='* ('
tailb=''
stem.taila=99
stem.tailb=stem.taila
say stem.tailb      /* Displays: 99 */
/* But the following instruction would cause an error */
/*      say stem.* ( */
```

스템 다음에 임베디드된 기호로 상수 기호를 사용할 수 없습니다(예를 들어, 12.3E+5). 이 경우 전체 기호는 유효한 기호일 수 없습니다.

다음은 복합 기호입니다.

```
FRED.3
Array.I.J
AMESSY..One.2.
<.F.R.E.D>.<.A.B>
```

기호가 사용되기 전에(즉, 참조 시) 언어 프로세서는 후미의 값(예에서 I, J 및 One)을 대체하므로 파생된 새 이름을 생성합니다. 파생된 이름은 단순한 기호처럼 사용됩니다. 즉, 해당 값은 기본적으로 파생된 이름이거나 또는 (지정의 대상으로 사용되었던 경우) 해당 값은 파생된 이름으로 이름 지정된 변수의 값입니다.

발생된 기호로의 대체는 공통 스템이 있는 변수 컬렉션의 임의 색인 작성(등록)을 허용합니다. 대체된 값은 모든 문자(마침표와 공백 포함)를 포함할 수 있습니다. 대체는 한 번만 가능합니다.

요약: 기호로 참조되는 복합 변수의 파생된 이름

```
s0.s1.s2. --- .sn
```

은

```
d0.v1.v2. --- .vn
```

에서 제공되며, 여기서 d0은 대문자 양식의 기호 s0이며 vn에 대한 v1은 상수나 단순한 기호 s1에서 sn까지의 값입니다. 기호 s1-sn 중 하나는 null일 수 있습니다. 값 v1-vn은 null일 수 있고, 모든 문자를 포함할 수도 있습니다(특히, 소문자가 대문자로 변환되지 않고, 공백이 제거되지 않으며 마침표에 특별한 의미는 없음).

일부 예는 REXX 프로그램으로부터 작은 추출의 양식을 따릅니다.

```
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" */
```

복합 기호를 사용하여 아래첨자가 필수 숫자가 아닌 변수의 어레이와 목록을 설정할 수 있으므로 프로그래머에게 큰 범위를 제공합니다. 유용한 애플리케이션은 하나 이상의 변수의 값에서 사용된 아래첨자에서 배열을 설정하여 연관 메모리(주소 지정 가능한 콘텐츠)의 양식에 영향을 줍니다.

**구현 최대:** 대체 전후에, 변수 이름의 길이가 250자를 초과할 수 없습니다.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## 스텝

스텝은 마지막 문자인 한 개의 마침표만 포함하는 기호입니다. 숫자 또는 마침표로 시작할 수 없습니다.

다음은 스텝입니다.

```
FRED.
A.
<.A.B>.
```

기본적으로 스텝의 값은 해당 기호의 문자(즉, 대문자로 변환됨)로 구성되는 문자열입니다. 기호에 값이 지정된 경우 기호는 변수에 이름을 지정하고 그 값은 해당 변수의 값입니다.

스텝이 지정의 대상으로 사용되는 경우 이름이 해당 스텝으로 시작되는 가능한 모든 복합 변수는 이전에 값을 가지고 있었는지 여부에 상관없이 새 값을 받습니다. 지정에 따라 해당 스텝을 포함하는 모든 기호에 대한 참조는 스텝에 또는 개별 변수에 다른 값이 지정될 때까지 새 값을 리턴합니다.

예를 들어, 다음과 같습니다.

```
hole. = "empty"
hole.9 = "full"

say hole.1 hole.mouse hole.9

/* says "empty empty full" */
```

따라서 변수의 전체 컬렉션에 동일한 값을 지정할 수 있습니다. 예를 들어, 다음과 같습니다.

```
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
```

**참고:** 스템을 사용하여 변수의 전체 컬렉션에 지정된 값을 항상 얻을 수 있습니다. 그러나 이는 파생된 이름이 스템과 같은 복합 변수를 사용하는 것과는 같지 않습니다. 예를 들어, 다음과 같습니다.

```
total. = 0
null = ""
total.null = total.null + 5
say total. total.null          /* says "0 5" */
```

DROP 및 PROCEDURE 명령어를 사용하여 해당 스템에 의해 참조되는 변수의 컬렉션을 조작할 수 있습니다. DROP FRED. 는 해당 이름을 가진 모든 변수를 삭제하고(159 페이지의 『DROP』 참조) PROCEDURE EXPOSE FRED. 는 해당 스템을 가진 가능한 모든 변수를 노출시킵니다(168 페이지의 『PROCEDURE』 참조).

#### 참고:

1. ARG, PARSE 또는 PULL 명령어 또는 VALUE 기본 제공 함수나 변수 풀 인터페이스가 변수를 변경하는 경우 그 영향은 지정과 동일합니다. 값을 지정할 수 있는 어느 곳에서나 스템을 사용하여 변수의 전체 컬렉션을 설정합니다.
2. 표현식은 연산자 =를 포함할 수 있고 명령어는 표현식으로만 구성될 수 있으므로(145 페이지의 『외부 환경에 대한 명령』 참조) 모호성은 다음 규칙에 의해 해결됩니다. 기호로 시작되고 두 번째 토큰이 등호(=)인(또는 등호로 시작되는) 절은 표현식(또는 키워드 명령어)이 아니라 지정입니다. 여러가지 방법(예: 널 문자열을 첫 번째 이름 앞에 넣거나 표현식의 첫 번째 파트를 소괄호로 엔클로징하는 방법)으로 절이 명령어로 처리되는지 확인할 수 있으므로 이는 제한사항이 아닙니다.

마찬가지로 의도치 않게 REXX 키워드를 지정에서 변수 이름으로 사용하는 경우 이로 인해 혼란이 발생해서는 안됩니다. 예를 들어, 다음 절은 ADDRESS 명령어가 아니라 지정입니다.

```
Address='10 Downing Street';
```

3. SYMBOL 함수를 사용하여 기호에 값이 지정되었는지 여부를 테스트할 수 있습니다(201 페이지의 『SYMBOL』 참조). 또한 SIGNAL ON NOVALUE를 설정하여 초기화되지 않은 변수의 사용을 트랩할 수 있습니다(복합 변수에 후미가 있는 경우는 제외됨. 233 페이지의 『제 22 장 조건 및 조건 트랩』 참조).

## 외부 환경에 대한 명령

주위 환경에 대한 명령 실행은 REXX의 내부 부분입니다.

### 환경

REXX 프로그램이 실행되는 시스템은 명령을 처리하기 위한 하나 이상의 환경을 포함할 것으로 가정됩니다. 환경은 REXX 프로그램에 대한 항목에 대해 기본적으로 선택됩니다. ADDRESS 명령어를 사용하여 환경을 변경할 수 있습니다. ADDRESS 기본 제공 함수를 사용하여 현재 환경의 이름을 찾을 수 있습니다. 기본 운영 체제는 REXX 프로그램에 외부적인 환경을 정의합니다. REXX/CICS 프로그램을 위한 기본 환경은 REXXCICS입니다.

### 명령

명령을 현재 주소 지정된 환경에 보내려면 양식의 절을 사용하십시오.

```
expression;
```

표현식이 평가되어 결과가 문자열이 되며(null 문자열일 수 있음), 적절하게 준비되어 기본 시스템에 제출됩니다. 평가받지 않을 표현식 파트는 따옴표로 묶어야 합니다.

환경은 명령을 처리하는데 부작용이 있을 수 있습니다. 결국 리턴 코드를 설정한 후 언어 프로세서로 제어를 리턴합니다. 리턴 코드는 문자열, 일반적으로 숫자로, 처리된 명령에 관한 일부 명령을 리턴합니다. 리턴 코드는 명령이 성공적인지 아닌지를 보통 표시하지만 다른 정보를 표시할 수도 있습니다. 언어 프로세서는 이 리턴 코드를 REXX 특수 변수 RC에 배치합니다. 236 페이지의 『특수 변수』의 내용을 참조하십시오.

리턴 코드 설정 외에, 오류나 실패가 발생하면 기본 시스템은 언어 프로세서에 표시할 수도 있습니다.

- 오류는 해당 명령을 사용하는 프로그램이 보통 준비될 것으로 예상하는 명령으로 발생하는 조건입니다. 예를 들어, 편집 시스템에 대한 찾기 명령은 오류로 requested string not found를 보고할 수 있습니다.



- 실패는 해당 명령을 사용하는 프로그램이 보통 복구되지 않을 것으로 예상하는 명령으로(예를 들어 실행할 수 없거나 찾을 수 없는 명령) 발생하는 조건입니다.

명령에서 오류나 실패는 ERROR 또는 FAILURE에 대한 조건 트랩이 ON인 경우 REXX 처리에 영향을 줄 수 있습니다(233 페이지의 『제 22 장 조건 및 조건 트랩』 참조). traced if TRACE E 또는 TRACE F가 설정된 경우 명령을 추적하게 될 수도 있습니다. TRACE Normal은 TRACE F와 동일하며 기본값입니다. 174 페이지의 『TRACE』의 내용을 참조하십시오.

여기에 기본 REXX/CICS 명령 환경에 명령을 제출하는 예가 있습니다. 다음 시퀀스의 결과는 REXX/CICS에 제출된 문자열 EXECIO \* READ TSQUEUE1 MYDATA.입니다.

```
TSQ1 = 'TSQUEUE1'
"EXECIO * READ" TSQ1 "MYDATA."
```

리턴 시, CICS 임시 스토리지 큐 TSQUEUE1이 MYDATA 어레이로 성공적으로 읽힌 경우 RC에 있는 리턴 코드에는 값 0이 있습니다. TSQUEUE1이 비어 있는 경우, 적절한 리턴 코드는 RC에 위치합니다.

**참고:** 해당 환경에 전달되기 전에 표현식이 평가된다는 점을 기억하십시오. 따옴표에서 평가받지 않는 표현식의 파트를 따옴표로 묶으십시오. 예:

```
"EXECIO * READ" /* * does not mean "multiplied by" */
```

## CICS에서 실행되는 REXX의 기본 구조

REXX/CICS 지원은 REXX/CICS 개발 시스템에 대해서는 CICREXD, REXX CICS 런타임 기능에 대해서는 CICREXR라는 이름의 기본 인터페이스 프로그램을 제공하며 이 프로그램은 CICS 리전 내에서 REXX exec를 로드하고 실행하는 데 사용됩니다.

각 REXX exec은 별도의 CICS 태스크에서 실행됩니다. 임의의 중첩된 REXX exec은 상위 exec의 CICS 태스크에서 실행됩니다.

**참고:** exec이 개발 시스템에서 실행되고 있는지 또는 런타임 기능에서 실행되고 있는지 판별하기 위해 GETVERS 명령을 사용할 수 있습니다. 371 페이지의 『GETVERS』의 내용을 참조하십시오.

### REXX exec 호출

- Exec이 터미널에서 시작됨

CICS가 프로그램과 연관된 트랜잭션 ID를 사용하여 실행할 프로그램을 판별합니다. REXX/CICS는 REXX/CICS 명령인 DEFTRNID로 작성된 테이블을 사용하여 CICS 트랜잭션 ID를 특정 REXX exec과 연관시킵니다. REXX/CICS 제공 exec인 CICRXTRY와 연관된 CICS 트랜잭션 ID를 기본 REXX/CICS 트랜잭션 ID라고 합니다. 제공된 기본값은 REXX입니다.

CICS 화면에서 REXX만 입력하면 CICRXTRY exec이 시작됩니다. 다른 피연산자가 지정되는 경우(예: REXX MYEXEC ABC) exec MYEXEC이 시작되고 ABC는 이 exec에 인수로 전달됩니다. REXX/CICS 기본 트랜잭션 ID 외의 CICS 트랜잭션 ID는 연관된 exec이 시작되도록 하고 다른 피연산자는 이 exec에 인수로 전달됩니다. 예를 들어, EDIT TEST.EXEC은 REXX/CICS 편집기 exec인 CICEDIT가 시작되도록 하고 인수인 TEST.EXEC은 편집하거나 작성할 파일의 이름을 지정합니다. 모든 REXX/CICS 트랜잭션 ID에는 REXX/CICS 기본 모듈인 CICREXD와 연관시키는 CICS 정의가 있어야 합니다.

- Exec은 CICS START 명령을 사용하여 시작됩니다.

REXX/CICS exec은 CICS START 명령을 사용하여 시작될 수 있습니다. START 명령은 시작할 CICS 트랜잭션 ID의 이름을 지정하고 REXX/CICS DEFTRNID 명령에 의해 작성된 테이블은 시작할 exec의 이름을 지정합니다.

exec이 CICRXTRY이고 CICS 시작 데이터가 있는 경우 첫 번째 피연산자는 시작할 exec의 이름을 지정하고 그 외 다른 피연산자는 이 exec에 인수로 전달됩니다. 시작 데이터가 없는 경우 CICRXTRY가 시작됩니다.

CICRXTRY를 제외한 exec의 경우 시작 데이터는 인수로 exec에 전달됩니다. 일반적으로 REXX/CICS exec에는 연관된 터미널이 있습니다. 그러나 REXX/CICS exec에 트랜잭션에 연관된 터미널이 없는 경우 모든 터미널 출력은 REXX/CICS SET TERMOUT 명령이 지정한 대로 CICS 임시 스토리지 큐로 경로 지정되거나 버려집니다. 터미널이 연관되지 않은 트랜잭션에 대해 터미널 입력이 요청되는 경우 오류가 생성됩니다.

- Exec이 CICS LINK 또는 XCTL 명령을 사용하여 시작됨



REXX/CICS 인터페이스 프로그램인 CICREXD 또는 CICREXR은 CICS LINK 또는 XCTL 명령을 사용하여 호출할 수 있습니다. 이 방식으로 호출되는 경우 COMMAREA가 인터페이스 프로그램에 전달되어야 합니다.

- CICS LINK가 REXX를 호출하는 데 사용되는 경우 통신 영역의 길이는 최소한 16자 이상이어야 합니다. 처음 4자는 호출 애플리케이션에 완료 코드를 리턴하기 위해 사용됩니다. 이 완료 코드는 호출된 exec의 리턴 코드가 아니라 414 페이지의 『특정 명령과 연관되지 않은 리턴 코드』에 나열된 REXX/CICS 처리의 리턴 코드입니다. 다음 12자는 나중에 사용할 수 있도록 예약됩니다. 16자 접두부 뒤에 오는 데이터는 exec에 전달될 공백 구분된 REXX exec 이름 및 인수를 전달하는 데 사용할 수 있습니다. CICS LINK로 시작된 exec은 대화식 모드로 실행되어야 합니다. 이는 호출 애플리케이션으로 돌아가기 위해 호출자의 스토리지를 보호하는 데 필요합니다.
- CICS XCTL이 REXX를 호출하는 데 사용되는 경우 통신 영역은 MVS SIB 유형 1 제어 블록을 포함할 수 있습니다. SIB이 사용되지 않는 경우 이 영역의 길이는 최소한 16자 이상이어야 합니다. 16자 접두부는 나중에 사용하기 위해 예약됩니다. 16자 접두부 뒤에 오는 데이터를 사용하여 공백 구분 REXX exec 이름 및 이 exec에 전달될 인수를 전달할 수 있습니다.

- MVS SIB 유형 1 제어 블록에서 발행된 exec

REXX exec은 ATD(Application Type Descriptions)를 통해 OfficeVision 또는 MVS에서 호출될 수 있습니다. ATD를 작성할 때 호출에 XCTL 또는 START를 사용하십시오. XCTL을 사용하는 경우 애플리케이션 프로그램으로 CICREXD 또는 CICREXR을 사용하십시오. START를 사용하는 경우 애플리케이션 프로그램 필드에 REXX/CICS의 기본 트랜잭션 ID(예: REXX)를 배치하십시오.

exec 이름 및 임의의 매개변수를 전달하는 두 가지 방법이 있습니다.

- MSG-TEXT 필드 - (MSG-REDEF = '1')
- 매개변수 설명 레코드(PDR) - 'REXXEXEC'

MSG-TEXT 필드는 ATD 이름과 최소 하나 이상의 매개변수가 지정되는 경우 명령행에서 키 입력된 데이터로 채워집니다. PDR REXXEXEC이 사용되는 경우 필요한 매개변수가 추가된 exec 이름은 PDR에 대한 데이터 영역에 있어야 합니다. MSG-TEXT와 PDR REXXEXEC이 둘 다 사용되는 경우 MSG-TEXT의 데이터가 우선합니다. 둘 중 어느 것도 사용되지 않는 경우 REXXTRY 대화식 유틸리티(CICRTRY exec)가 호출됩니다.

**참고:** REXX 명령어 및 REXX/CICS 명령의 대화식 실행을 가능하게 하는 REXXTRY라고 하는 유틸리티(CICRTRY exec)가 REXX/CICS에 제공됩니다. 이 유틸리티를 호출하려면 CICRTRY와 연관된 REXX/CICS 트랜잭션 ID를 피연산자 없이 입력하십시오.

## exec이 실행되는 위치

REXX/CICS exec은 CICS 리전 내에서 해당 exec을 발행하는 CICS 태스크의 일부로 실행됩니다. REXX 해석기는 완전히 재진입 가능하고 16MB 행(AMODE=31, RMODE=ANY) 위에서 실행됩니다.

## Exec 찾기 및 로드

다음 규칙을 사용하여 exec을 찾고 이를 스토리지에 로드함으로써 이 exec을 시작할 수 있습니다.

1. EXECLOAD를 사용하여 스토리지에 로드된 exec이 검색되고 exec을 찾은 경우 EXECLOAD에 의해 로드된 사본이 사용됩니다.
2. REXX/CICS CD 명령에 의해 정의된 사용자의 현재 RFS 디렉토리가 검색됩니다. 찾은 경우 exec은 로드되어 시작됩니다.
3. REXX/CICS PATH 명령에 의해 정의된 사용자의 경로가 검색됩니다. 찾은 경우 exec은 로드되어 시작됩니다.
4. 사용자가 권한 부여된 사용자인 경우 CICS 시동 JCL에서 CICAUTH의 ddname으로 지정된 데이터 세트가 검색됩니다. exec을 찾은 경우 이 exec이 로드되어 시작됩니다.
5. CICEXEC 및 CICUSER의 ddname을 사용하여 지정된 데이터 세트가 해당 순서대로 검색됩니다. exec을 찾을 수 없는 경우 오류 코드가 리턴됩니다.

## exec 편집

REXX/CICS 개발 시스템을 설치한 경우 REXX exec은 제공된 REXX/CICS 편집기를 사용하여 편집할 수 있습니다. 239 페이지의 『제 23 장 REXX/CICS 텍스트 편집기』의 내용을 참조하십시오. 또한 REXX/CICS exec이

MVS PDS에 상주하는 경우 TSO에서 ISPF/PDF 편집기(또는 기타 호환 가능한 편집기)를 사용하여 편집할 수 있습니다.

REXX/CICS exec의 열 73 - 8-에서 순서 번호는 허용되지 않습니다.

## REXX 파일 시스템

Exec은 REXX/CICS와 함께 제공되는 VSAM-based RFS(REXX File System)에 또는 MVS 파티션된 데이터 세트에 멤버로 저장될 수 있습니다. [267 페이지의 『제 24 장 REXX/CICS 파일 시스템』](#)의 내용을 참조하십시오.

**참고:** exec을 호출하기 위해 지정한 파일 ID가 파일 유형 확장자를 포함하지 않는 경우 REXX exec을 찾고 발행하려고 시도할 때 파일 유형이 EXEC인 RFS 파일 ID만 검색됩니다. 파일 ID에 파일 유형 확장자가 포함된 경우 일치하는 파일 유형을 가지는 RFS 파일만 검색됩니다(해당 exec을 찾고 실행하려고 시도할 때). MVS PDS 검색의 경우 파일 이름만 PDS 멤버 이름과 비교됩니다.

## exec 실행 검색 순서의 제어

CD 및 PATH 명령은 exec을 로드하려고 할 때 사용자 REXX 라이브러리의 검색 순서를 정의합니다. 현재 디렉토리(CD 명령에 의해 설정됨)가 검색되고 나면 PATH 명령을 사용하여 지정된 모든 디렉토리가 검색되고 그 뒤에는 CICS 리전의 시동 JCL에서 지정된 시스템 MVS PDS 라이브러리가 옵니다. 검색 순서에 대한 자세한 정보는 [179 페이지의 『함수와 서브루틴』](#)의 내용을 참조하십시오.

## 사용자 작성 명령 추가

REXX/CICS 명령으로 호출될 수 있도록 exec을 정의할 수 있습니다. DEFCMD 명령은 REXX/CICS 사용자 명령을 정의하는 데 사용됩니다. DEFCMD는 표준 CICS 지원 언어뿐 아니라 REXX에서 작성된 명령도 지원합니다. [355 페이지의 『DEFCMD』](#)의 내용을 참조하십시오.

## 표준 REXX 기능의 지원

표준 REXX 기능(예: SAY 및 TRACE 명령문, PULL 및 PARSE EXTERNAL 명령문, REXX 스택화 및 REXX 함수)이 지원됩니다.

### SAY 및 TRACE 명령문

REXX SAY 및 TRACE 터미널 I/O 출력 명령문은 CICS 터미널 제어 지원을 사용하여 시뮬레이션된 행 모드 출력을 제공합니다. 또한 SET TERMOUT 명령은 행 모드 출력을 임시 스토리지 큐로 라우팅하는 데 사용할 수 있습니다. [382 페이지의 『SET』](#)의 내용을 참조하십시오.

### PULL 및 PARSE EXTERNAL 명령문

REXX PULL 및 PARSE EXTERNAL 터미널 I/O 입력 명령문은 CICS 터미널 제어 지원을 사용하여 시뮬레이션된 행 모드 입력을 제공합니다.

1. PULL(또는 PARSE PULL)은 프로그램 스택에서 행을 가져오려고 시도하고 비어 있는 경우에만 터미널에 대한 읽기를 발행합니다.
2. 비터미널 접속 트랜잭션의 일부로 실행되고 있는 REXX exec에서 터미널 행 모드 입력을 수행하려는 시도는 오류이며 이로 인해 이 exec은 오류 메시지와 함께 종료됩니다.

### REXX 스택 지원

각 사용자에게는 REXX exec의 여러 생성 간 공유되는 프로그램 스택이 있습니다. 이 단일 자동 프로그램 스택의 이름은 지정되지 않습니다. 이름 지정된 스택이 필요한 경우 RLS LPUSH, LQUEUE 및 LPULL 명령을 사용하십시오.

### REXX 함수 지원

REXX/CICS는 표준 SAA 레벨 2 기본 제공 함수 세트를 지원하며 다음과 같은 경우는 예외입니다.

- 스트림 I/O 함수는 지원되지 않습니다.

- USERID 함수는 사용자가 사인온한 경우 1 - 8자의 CICS 사용자 ID를 리턴합니다. CICS 사용자가 사인온하지 않았고 CICS 리전에 대해 기본 사용자가 지정된 경우(CICS 시스템 프로그래머가 CICS 시동 매개변수에 DFLTUSER를 지정하여) 해당 값을 사용합니다.

**참고:** 기본 사용자는 REXX 파일 시스템과 REXX 목록 시스템 디렉토리를 공유합니다.

- REXX 사용자가 CICS 리전의 가상 스토리지를 표시하거나 수정할 수 있도록 하는 STORAGE 함수. 이 함수는 권한 부여된 exec에서 또는 권한 부여된 사용자에게 의해서만 성공적으로 호출할 수 있습니다.

## REXX 명령 환경 지원

현재 사용 가능한(REXX ADDRESS 명령과 함께 사용할) REXX 명령 환경은 REXXCICS, CICS, EXECSQL, EDITSVR, FLSTSVR, RFS 및 RLS입니다.

### REXX 호스트 명령 환경 추가

새 REXX/CICS 명령과 명령 환경이 동적으로 정의될 수 있도록 하기 위한 지원이 제공됩니다. 새 명령은 REXX 또는 임의의 REXX/CICS 지원 언어(예: Assembler, COBOL, C, PL/I)로 작성될 수 있습니다. 명령 및 명령 환경이 REXX/CICS에 대해 정의되는 방식에 대한 자세한 정보는 [291 페이지의 『제 26 장 REXX/CICS 명령 정의』](#)의 내용을 참조하십시오.

## 표준 CICS 기능에 대한 지원

표준 CICS 기능에 대한 지원을 설명합니다. 여기에는 CICS �핑된 I/O 지원, 데이터 세트 I/O 서비스, CICS 기능 및 서비스에 대한 인터페이스, exec에서 사용자 애플리케이션 발행, CICS 임시 및 트랜지언트 스토리지 큐에 대한 REXX 인터페이스, 의사 대화식 트랜잭션 지원 및 DBCS 지원이 포함됩니다.

### CICS �핑된 I/O 지원

CICS 기본 �핑 지원(BMS) I/O는 CICS SEND MAP, RECEIVE MAP 및 CONVERSE MAP 명령과 REXX/CICS CONVTMAP 및 COPYS2R 명령에 의해 제공됩니다. [449 페이지의 『제 36 장 BMS\(Basic Mapping Support\) 예』](#)의 내용을 참조하십시오.

**참고:** BMS 맵은 일반적인 CICS 프로시저를 사용하여 사전 정의되어야 합니다.

### 데이터 세트 I/O 서비스

표준 CICS 파일 I/O 명령(예: EXEC CICS READ 및 WRITE)이 지원됩니다. 상위 레벨 I/O는 제공된 RFS 명령을 사용하여 EXEC에서 VSAM 기반 RFS(REXX File System)로 수행될 수 있습니다. 또한 동적 할당은 표준 MVS 파티션된 데이터 세트가 IMPORT 및 EXPORT 명령, 그리고 REXX/CICS 편집기에서 사용할 수 있도록 하기 위해 사용됩니다.

### CICS 기능 및 서비스에 대한 인터페이스

ADDRESS CICS 명령 환경에서는 대부분의 CICS 명령(애플리케이션 개발 참조서에서 정의된 대로)에 대해 지원이 제공됩니다. 지원되는 명령에 대한 자세한 정보는 [331 페이지의 『제 30 장 REXX/CICS 명령』](#)의 내용을 참조하십시오.

### exec에서 사용자 애플리케이션 발행

REXX/CICS는 EXEC CICS START, LINK 및 XCTL 명령을 제공하여 CICS 트랜잭션을 시작하거나 REXX exec 내에서 CICS 프로그램을 호출하는 기능을 제공합니다.

### CICS 스토리지 큐에 대한 REXX 인터페이스

명령 지원은 REXX/CICS에서 CICS 임시 스토리지 및 트랜지언트 데이터 큐를 읽고, 쓰고, 삭제하기 위해 존재합니다.

## 의사 대화식 트랜잭션 지원

REXX exec을 위한 CICS 의사 대화식 지원은 CICS RETURN TRANSID() 명령, REXX/CICS PSEUDO 명령(376 페이지의 『PSEUDO』 참조) 및 SETSYS PSEUDO 명령(385 페이지의 『SETSYS』 참조)의 사용을 통해 지원됩니다.

## 다른 프로그래밍 언어에 대한 인터페이스

REXX/CICS는 CICS LINK 및 CICS XCTL REXX/CICS 명령에 대한 지원을 통해 REXX/CICS 지원 언어로 작성된 CICS 프로그램을 호출하는 기능을 지원합니다. DEFCMD 및 DEFSCMD 명령을 사용하여 정의되는 새 REXX 명령을 구현하는 데 사용되는 프로그램에 대해 동일한 지원을 제공합니다. EXEC CICS START가 REXX exec에서 실행될 수 있도록 하기 위한 지원도 제공됩니다.

## DBCS 지원

SAA 레벨 2 REXX에 포함된 전체 범위의 DBCS 함수와 처리 기술을 REXX/CICS 사용자가 사용할 수 있습니다.

## 기타 기능

- CICS 사용자의 4문자 터미널 ID를 리턴하기 위해 TERMID 명령이 제공되었습니다.
- 검색 PF 키를 지정하여 REXXTRY 대화식 유틸리티(CICRTRY exec)에 있는 동안 행 모드 I/O를 사용하여 입력한 마지막 입력 행을 검색할 수 있습니다. SET RETRIEVE 명령(382 페이지의 『SET』)을 참조하십시오.
- SET TERMOUT 명령은 행 모드 터미널 출력(SAY 또는 TRACE로부터)이 터미널 대신 또는 터미널에 추가하여 CICS 임시 스토리지 큐로 경로 지정되도록 합니다.
- PULL 명령어는 PULL이 터미널에서 데이터를 읽는 경우 AID 키의 이름이 눌린 상태로 REXX 변수 PULLKEY를 설정합니다.
- 터미널에 대한 행 모드 출력은 화면이 가득 찼을 경우 화면의 오른쪽 하단 모서리에 MORE가 표시되도록 합니다. 계속하려면 Clear 또는 Enter를 누르십시오.
- 터미널로부터의 행 모드 입력은 화면의 오른쪽 하단 모서리에 READ가 표시되도록 합니다.

## 제 18 장 키워드 명령어

키워드 지시사항은 첫 번째 절이 지시사항을 식별하는 키워드로 시작하는 하나 이상의 절입니다. 일부 키워드 지시사항이 제어의 플로우에 영향을 주는 반면에, 다른 지시사항은 서비스를 프로그래머에게 제공합니다. DO와 같이, 일부 키워드 명령어는 중첩된 명령어를 포함할 수 있습니다.

구문 다이어그램에서, 대문자로 기호(단어)는 키워드 또는 서브 키워드를 표시합니다. 기타 단어(예: *expression*)는 이전에 정의된 대로 토큰의 컬렉션을 표시합니다. 그러나 키워드와 서브 키워드는 대소문자를 사용하지 않습니다. 기호 `if`, `If` 및 `iF` 모두는 동일 효과를 가집니다. 행의 끝에 표시되기 때문에 일반적으로 절 구분 기호(`;`) 대부분을 생략할 수 있습니다.

**141** 페이지의 『절 및 명령어』에 설명된 대로, 해당 키워드가 절의 첫 번째 토큰인 경우 및 두 번째 토큰이 = 문자(지정 표시) 또는 콜론(레이블 표시)으로 시작되지 않은 경우에만 키워드 명령어가 인식됩니다. 키워드 ELSE, END, OTHERWISE, THEN 및 WHEN은 동일한 상황에서 인식됩니다. REXX에서 정의된 키워드로 시작되는 모든 절이 명령일 수 없습니다. 그러므로 다음은 ARG 키워드 명령어이며 ARG 내장 함수에 대한 호출로 시작하는 명령이 아닙니다.

```
arg(fred) rest
```

키워드가 DO, IF 또는 SELECT 명령어의 올바른 위치에 없는 경우 구문 오류가 발생합니다.(키워드 THEN이 IF 또는 WHEN절의 본문에서도 인식됩니다.) 다른 컨텍스트에서 키워드가 예약되지 않으며 변수의 이름이나 레이블로 사용될 수 없습니다(일반적으로 권장되지 않지만).

서브 키워드로 알려진 특정 다른 키워드는 개별 명령어의 절 내에 예약됩니다. 예를 들어, 기호 VALUE 및 WITH는 ADDRESS 및 PARSE 명령어에서 각각 서브 키워드입니다. 세부사항은 각 명령어의 설명을 참조하십시오. 예약된 키워드에 관한 일반 논의는 445 페이지의 『예약 키워드』의 내용을 참조하십시오.

키워드에 인접한 공백은 후속 토큰으로부터의 키워드 분리 이외의 효과는 없습니다. VALUE 다음 하나 이상의 공백은 다음 예의 서브 키워드에서 *expression*을 분리하는 데 필요합니다.

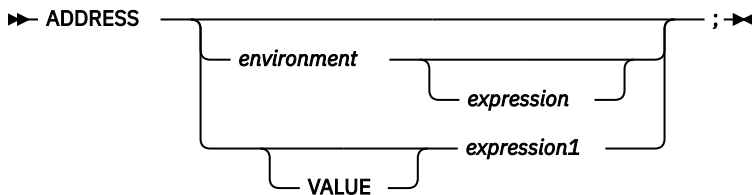
ADDRESS VALUE *expression*

그러나 가독성에 추가하지 않더라도 다음 예에서 VALUE 서브 키워드 다음에 공백이 필요하지 않습니다.

ADDRESS VALUE 'ENVIR' || number

## ADDRESS

ADDRESS는 임시로 또는 영구적으로 명령의 대상을 변경합니다. 명령은 외부 환경에 발송된 문자열입니다. 표현 식으로만 구성되거나 ADDRESS 명령어를 사용하여 명령을 전송할 수 있습니다.



대체 하위 명령 환경의 개념은 86 페이지의 『프로그램에서 명령 실행』에서 설명됩니다.

단일 명령을 지정된 환경에 전송하려면, *environment*, 리터럴 문자열이나 단일 기호를 코딩하십시오. *expression* 다음에 오는 상수가 되도록 사용됩니다. (환경 이름은 명령을 처리할 수 있는 외부 프로시저 또는 프로세스의 이름입니다.) 환경 이름은 8자로 제한됩니다. *expression*이 평가되고 결과 문자열이 *environment*로 라우팅되어 명령으로 처리됩니다. (평가받기 위해 원하지 않는 표현식의 파트를 따옴표로 묶으십시오.) 명령 실행 후, *environment*는 이전의 설정으로 다시 설정되므로 단일 명령에 대해서는 임시로 대상이 변경됩니다. 다른 명령에 대해서처럼 특수 변수 RC가 설정됩니다. (145 페이지의 [『외부 환경에 대한 명령』](#) 참조) 이런 방식으로 처리된 오류와 명령의 실패는 펄스처럼 트래핑되거나 추적됩니다.



## 예제

```
ADDRESS CICS "READQ TSQ QUEUE('QUEUE1') INTO(VAR1)" /* CICS */
```

*environment*만을 지정한 경우, 대상이 지속적으로 변경됩니다. 다음 ADDRESS 명령어가 처리될 때까지 다음에 오는 모든 명령(REXX 명령어나 지정 명령어가 아닌 질)이 지정된 명령 환경에 라우팅됩니다. 이전에 선택된 환경이 저장됩니다.

```
address cics
"READQ TSQ QUEUE('QUEUE1') INTO(VAR1)"
ADDRESS RFS
'COPY PROFILE.EXEC TEMP.EXEC'
```

마찬가지로 VALUE 양식을 사용하여 환경에 대해 지속적으로 변경할 수 있습니다. 여기에서 *expression1*(단순한 변수 이름일 수 있음)이 평가되며 결과는 환경의 이름을 형성합니다. *expression1*이 리터럴 문자열이나 기호로 시작되지 않은 경우(즉, 연산자 문자나 소괄호와 같이 특수 문자로 시작된 경우) 서브 키워드 VALUE를 생략할 수 있습니다.

```
ADDRESS ('ENVIR'||number) /* Same as ADDRESS VALUE 'ENVIR'||number */
```

인수 없이, 명령은 이전에 환경이 지속적으로 변경되기 전에 선택했던 환경으로 다시 라우팅되고 현재 환경 이름이 저장됩니다. 환경을 변경한 후, ADDRESS의 반복된 실행만이 교대로 두 환경 사이 명령 대상을 전환시킵니다.

두 환경 이름이 내부 및 외부 서브루틴과 함수 호출에 자동으로 저장됩니다. [153 페이지의 『CALL』](#)의 내용을 참조하십시오.

주소 지정 설정은 현재 선택된 환경 이름입니다. ADDRESS 기본 제공 함수를 사용하여 현재 주소 설정을 검색할 수 있습니다([182 페이지의 『ADDRESS』](#) 참조).

## ARG

ARG는 프로그램 또는 내부 루틴에 제공된 인수 문자열을 검색하고, 변수에 지정합니다.

```
➡ ARG ┌──────────────────┐ ; ➡
      │ template_list      │
```

ARG는 짧은 형식의 명령어입니다.

```
➡ PARSE UPPER ARG ┌──────────────────┐ ; ➡
                   │ template_list    │
```

*template\_list*는 단일 템플레이트이지만, 쉼표로 구분된 여러 템플리트일 수 있습니다. 지정되면, 각 템플리트는 공백이나 패턴 또는 둘 다로 분리된 기호 목록입니다.

서브루틴이나 내부 함수가 처리되지 않으면, 프로그램에 매개변수로서 전달된 문자열은 구문 분석([211 페이지의 『제 20 장 구문 분석』](#))에 관한 섹션에서 설명된 규칙에 따라 변수로 구문 분석됩니다.

서브루틴 또는 내부 함수가 처리 중인 경우, 사용된 데이터는 호출자가 루틴에 전달한 인수 문자열입니다.

이 경우, 두 경우 모두 언어 프로세서는 처리 전에 전달된 문자열을 대문자(즉, a-z를 소문자 A-Z로)로 변환합니다. 대문자 변환을 원하지 않는 경우 PARSE PULL 명령어를 사용하십시오.

(일반적으로 다른 템플리트로) 동일한 소스 문자열 또는 문자열에 반복해서 ARG 및 PARSE ARG 명령어를 사용할 수 있습니다. 소스 문자열은 변경되지 않습니다. 구문 분석된 데이터의 길이 또는 콘텐츠에 대한 유일한 제한 사항은 호출자가 부과하는 제한사항입니다.

## 예제

```
/* String passed is "Easy Rider" */
Arg adjective noun .
```

```

/* Now:  ADJECTIVE  contains 'EASY'      */
/*       NOUN       contains 'RIDER'     */

```

프로그램이나 루틴에 둘 이상의 문자열을 사용 가능하다고 예상하는 경우, 각 템플릿이 차례로 선택되도록 구문 분석 *template\_list*에서 쉽표를 사용할 수 있습니다.

```

/* 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'           */

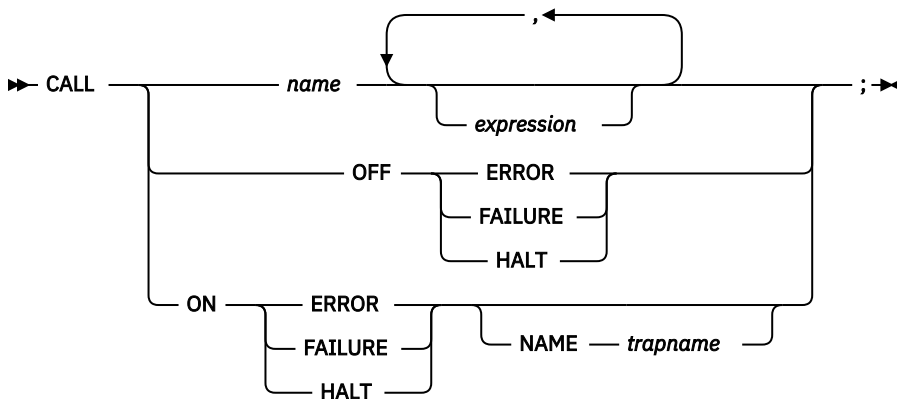
```

#### 참고:

1. ARG 기본 제공 함수는 REXX 프로그램이나 내부 루틴에 대한 인수 문자열을 검색하거나 검사할 수도 있으며 [183 페이지의 『ARG\(인수\)』](#)의 내용을 참조하십시오.
2. 처리 중인 데이터의 소스는 프로그램의 항목에서도 사용 가능합니다. 세부사항은 PARSE 명령어(SOURCE 옵션) [166 페이지의 『PARSE』](#)의 내용을 참조하십시오.

## CALL

CALL은 루틴을 호출하거나(*name*을 지정한 경우) 특정 조건의 트래핑을 제어합니다(ON 또는 OFF를 지정한 경우).



트래핑을 제어하려면 OFF 또는 ON 및 트래핑할 조건을 지정합니다. OFF는 지정된 조건 트랩을 끕니다. ON은 지정된 조건 트랩을 실행합니다. 조건 트랩의 정보는 [233 페이지의 『제 22 장 조건 및 조건 트랩』](#)의 내용을 참조하십시오.

루틴을 호출하려면 상수로 받아들여지는 리터럴 문자열, 기호 또는 *name*을 지정하십시오. *name*은 글자 그대로 처리되는 기호 또는 리터럴 문자열이어야 합니다. 호출된 루틴은 다음과 같을 수 있습니다.

#### 내부 루틴

이를 호출하는 CALL 지시사항 또는 함수 호출과 같은 프로그램에 있는 함수 또는 서브루틴.

#### 내장 루틴

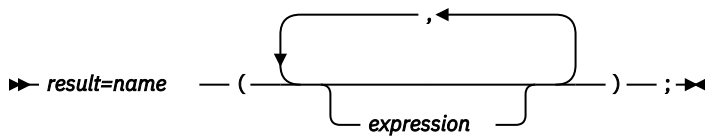
REXX 언어의 일부로 정의되는 함수(서브루틴으로 호출될 수 있음).

#### 외부 루틴

기본 제공되지 않거나 이를 호출하는 CALL 명령어나 함수와 동일한 프로그램에 없는 함수 또는 서브루틴.

*name*이 문자열인 경우(즉, 따옴표에서 *name* 지정), 내부 루틴 검색이 무시되고 기본 제공 함수나 외부 루틴만이 호출됩니다. 기본 제공 함수(및 일반적으로 외부 루틴의 이름)의 이름은 대문자입니다. 그러므로 리터럴 문자열의 이름도 대문자여야 합니다.

호출 루틴이 선택적으로 결과를 리턴할 수 있고, 리턴 시 CALL 명령어는 기능상으로 다음 절과 일치합니다.



호출된 루틴이 결과를 리턴하지 않으면 함수로 호출하는 경우 오류가 발생합니다(이전에 표시된 것과 같이).

REXX/CICS는 심표로 구분되는 최대 20개의 표현식 지정을 지원합니다. 표현식은 왼쪽에서 오른쪽으로 순서대로 평가되고, 루틴 실행 중 인수 문자열을 형성합니다. 호출된 루틴에서의 ARG 또는 PARSE ARG 명령어 또는 ARG 기본 제공 함수는 제어가 CALL 명령어로 리턴될 때까지 이전에 호출 프로그램에서 활성화하는 대신 이 문자열에 액세스합니다. 추가 심표를 포함하여, 적절한 경우, 표현식을 생략할 수 있습니다.

그러면 CALL은 함수 호출과 같은 메커니즘을 정확하게 사용하여 *name*이라는 루틴의 분기를 발생시킵니다. 179 페이지의 『제 19 장 함수』의 내용을 참조하십시오. 검색 순서는 함수의 섹션에 있지만 간단하게 다음과 같습니다.

#### 내부 루틴:

CALL 명령어에서 *name*과 일치하는 레이블로 시작하는 동일한 프로그램 내 명령어 시퀀스가 있습니다. 따옴표에서 루틴 이름을 지정하는 경우, 내부 루틴을 해당 검색 순서에 고려하지 않습니다. SIGNAL 및 CALL 을 함께 사용하여 실행 시 이름을 판별하는 내부 루틴을 호출할 수 있습니다. 이를 다중 방향 호출이라고 합니다 (172 페이지의 『SIGNAL』 참조). RETURN 명령어는 내부 루틴의 실행을 완료합니다.

#### 내장 루틴:

이는 다양한 함수를 제공하기 위한 언어 프로세서에 내장된 루틴입니다. 루틴에 의한 결과인 문자열을 항상 리턴합니다.(181 페이지의 『내장 함수』 참조).

#### 외부 루틴:

사용자는 언어 프로세서와 호출 프로그램에 외부적인 루틴을 쓰거나 사용할 수 있습니다. 외부 루틴은 REXX 에서 코드화되어야 합니다. CALL 명령어가 서브루틴으로서 REXX로 작성된 외부 루틴을 호출하면, ARG 또는 PARSE ARG 명령어 또는 ARG 기본 제공 함수가 있는 인수 문자열을 검색할 수 있습니다.

내부 루틴 실행 중, 이전에 알려진 모든 변수는 일반적으로 액세스 가능합니다. 그러나 PROCEDURE 명령어는 서브루틴과 호출자를 서로 보호하도록 로컬 변수 환경을 설정할 수 있습니다. PROCEDURE 명령어에 대한 EXPOSE 옵션은 선택된 변수를 루틴에 노출시킬 수 있습니다.

외부 프로그램을 서브루틴으로 호출하는 것은 내부 루틴을 호출하는 것과 유사합니다. 그러나 외부 루틴은 모든 호출자의 변수가 항상 숨겨진 내재적 PROCEDURE입니다. 내부 값의 상태(NUMERIC 설정 등)는 기본값(호출자의 값 상속 대신)으로 시작합니다. 또한 루틴에서 리턴하도록 EXIT을 사용할 수 있습니다.

제어가 내부 루틴에 도달하면, CALL 명령어의 행 번호는 변수 SIGL에서 사용할 수 있습니다(호출자의 변수 환경에서). 디버그 지원으로 사용될 수 있으므로 제어가 루틴에 도달하는 방법을 알아낼 수 있습니다. 내부 루틴이 PROCEDURE 명령어를 사용하는 경우, CALL의 행 번호에 액세스하려면 EXPOSE SIGL에 필요합니다.

결국 서브루틴은 RETURN 명령어를 처리해야 하며 그 때 제어는 최초 CALL 다음에 오는 절로 리턴합니다. RETURN 명령어가 표현식을 지정하면 변수 RESULT는 해당 표현식의 값으로 설정됩니다. 그렇지 않으면, 변수 RESULT가 삭제됩니다(초기화되지 않음).

내부 루틴은 그 자체로의 재귀 호출과 마찬가지로 다른 내부 루틴에 대한 호출을 포함할 수 있습니다.

**구현 최대:** 제어 구조의 전체 중첩이 내부 루틴 호출을 포함하며 사용 가능한 스토리지에 따라 다릅니다.

#### 예

```
/* 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
```



내부 서브루틴(및 함수) 실행 중, 중요한 모든 정보가 자동으로 저장되며 루틴에서 리턴 시 복원됩니다. 다음과 같습니다.

- DO 루프 및 다른 구조의 상태.

서브루틴 내 SIGNAL을 실행하면 서브루틴이 호출되었을 때 활성이었던 DO 루틴 등이 종료되지 않기 때문에 안전합니다. (그러나 서브루틴 내 현재 활성인 루틴이 종료됩니다.)

- 추적 조치.

서브루틴이 디버그된 후, TRACE Off를 시작 시 삽입할 수 있으며 호출자의 추적에 영향을 주지 않습니다. 반대로, 간단하게 서브루틴을 디버깅하려고 하는 경우 시작 시 TRACE Results를 삽입하고 리턴되면(예: Off) 추적은 입력 시 조건으로 자동으로 복원됩니다. 마찬가지로, ?(대화식 디버그) 및 !(명령 금지)는 루틴에 저장됩니다.

- NUMERIC 설정.

산술 연산의 DIGITS, FUZZ 및 FORM(164 페이지의 『NUMERIC』 참조)이 저장되며 리턴 시 복원됩니다. 그러므로 서브루틴은 호출자에 영향을 주지 않고 사용할 필요가 있는 자릿수 등을 설정할 수 있습니다.

- ADDRESS 설정.

명령의 현재 및 이전 대상(151 페이지의 『ADDRESS』 참조)이 저장되며 리턴 시 복원됩니다.

- 조건 트랩.

조건 트랩(CALL ON 및 SIGNAL ON)이 저장된 다음 리턴 시 복원됩니다. 이는 CALL ON, CALL OFF, SIGNAL ON 및 SIGNAL OFF는 호출자가 설정한 조건에 영향을 주지 않고 서브루틴에서 사용될 수 있음을 의미합니다.

- 조건 정보.

이 정보는 현재 트래핑 조건의 상태와 기점을 설명합니다. CONDITION 기본 제공 함수는 이 정보를 리턴합니다. 186 페이지의 『CONDITION』의 내용을 참조하십시오.

- 경과 시간 클럭.

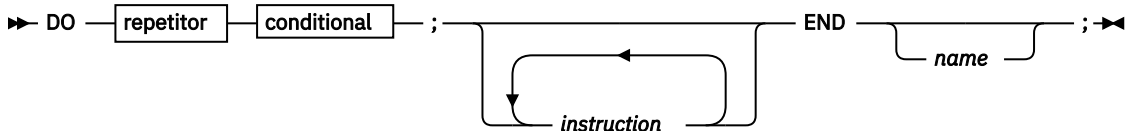
서브루틴은 해당 호출자에서 경과 시간 클럭을 상속받지만(201 페이지의 『TIME』 참조) 시간 클럭이 루틴 호출에 저장되기 때문에 서브루틴이나 내부 함수가 독립적으로 다시 시작되고 해당 호출자에 영향을 주지 않고 클럭을 사용할 수 있습니다. 같은 이유로 내부 루틴 내 시작된 클럭은 호출자에 사용 가능하지 않습니다.

- OPTIONS 설정.

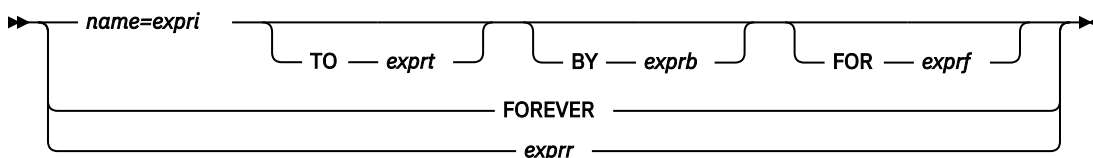
ETMODE 및 EXMODE가 저장되고 리턴 시 복원됩니다. 자세한 정보는 165 페이지의 『OPTIONS』의 내용을 참조하십시오.

## DO

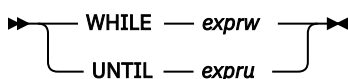
DO 그룹 명령어는 함께 그리고 선택적으로 반복적으로 처리합니다. 반복적 실행 중, 제어 변수(name)는 값의 일부 범위에 걸쳐 단계별로 처리할 수 있습니다.



### repetitor



### conditional



DO 그룹 명령어는 함께 및 선택적으로 반복적으로 처리합니다. 반복적 실행 중 제어 변수(*name*)는 값의 일부 범위에 걸쳐 단계별로 처리될 수 있습니다.

## 구문 노트

- *exprr*, *expri*, *exprb*, *exprt* 및 *exprf* 옵션은(있는 경우) 수로 평가하는 표현식입니다. *exprr* 및 *exprf* 옵션은 결과적으로 모든 양의 정수 또는 0이 되기 위해 추가적으로 제한됩니다. 필요한 경우 수는 NUMERIC DIGITS의 설정에 따라 반올림합니다.
- *exprw* 또는 *expwu* 옵션은(있는 경우) 1 또는 0으로 평가하는 표현식일 수 있습니다.
- TO, BY 및 FOR 구문은, 사용되면, 순서에 있을 수 있고, 작성되는 순서대로 평가됩니다.
- *instruction*은 IF, SELECT 또는 DO 명령어 자체와 같이 복잡한 구성을 포함하여, 지정, 명령 또는 키워드 명령을 포함한 모든 명령일 수 있습니다.
- 표현식에서 기호로 사용될 수 없다는 점에서, 서브키워드 WHILE 및 UNTIL가 DO 명령문 내에 예약됩니다. 마찬가지로 TO, BY 및 FOR는 *expri*, *exprt*, *exprb* 또는 *exprf*에서 사용될 수 없습니다. 바로 키워드 DO 다음에 오고 등호가 다음에 오지 않으면, FOREVER도 예약됩니다.
- 적절한 경우, *exprb* 옵션은 1로 기본 설정됩니다.

## 단순 DO 그룹

*repetitor* 또는 *conditional*을 지정하지 않은 경우 구성은 많은 명령어와 함께 그룹화합니다. 한 번에 처리됩니다. 다음 예에서 명령어는 한 번에 처리됩니다.

```
/* 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
```

## 반복적 DO 루프

DO 명령어에 리피터터 구문, 조건부 구문 또는 모두가 있는 경우, 명령어 그룹은 반복적 DO 루프를 형성합니다. 명령어는 리피터터 구문에 따라 처리되며, 선택적으로 조건부 구문으로 수정됩니다. [157 페이지의 『조건부 구문\(WHILE 및 UNTIL\)』](#)의 내용을 참조하십시오.

- 단순한 반복적 루프

단순한 반복적 루프는 리피터터 구문이 반복의 개수로 평가하는 표현식인 반복적 DO 루프입니다.

*repetitor*가 생략되지만 *conditional*이 있거나 *repetitor*가 FOREVER이면, 명령어 그룹이 명목상 "영구적"으로 처리되며, 즉 조건이 충족되거나 루프(예: LEAVE)를 종료하는 REXX 명령어를 처리할 때까지입니다.

**참고:** 조건부 구문에 관한 토론은 [157 페이지의 『조건부 구문\(WHILE 및 UNTIL\)』](#)의 내용을 참조하십시오.

반복적 루프의 간단한 형식에서 *exprr*가 즉시 평가되며(결과가 모든 양수의 정수나 0(영)이어야 함), 루프는 여러 번 처리됩니다. 예:

```
/* This displays "Hello" five times */
Do 5
    say 'Hello'
end
```

명령과 지정 사이 구별과 마찬가지로, *exprr*의 첫 번째 토큰은 기호이며 두 번째 토큰은 =이며(으로 시작되며), *repetitor*의 제어된 양식이 예상됩니다.

- 제어된 반복적 루프

제어된 양식은 *name*, 초기값으로 지정된 제어 변수(*expri*의 결과로, 0이 추가된 것처럼 형식이 지정됨)가 명령어 목록의 첫 번째 실행 앞에 지정됩니다. 변수는 명령어 목록이 처리되는 두 번째 및 추후 시간 전에 단계별로 처리됩니다(*exprb*의 결과를 추가하여).

종료 조건(*exprt*의 결과로 판별된)이 충족되지 않는 동안 명령어 목록이 처리됩니다. *exprb*가 양수 또는 0인 경우, *name*이 *exprt*보다 크면 루프가 종료됩니다. 음수인 경우, *name*이 *exprt*보다 작으면 루프가 종료됩니다.

*expri*, *exprt* 및 *exprb* 옵션의 결과는 숫자여야 합니다. 루프가 시작되기 전 및 제어 변수가 초기값으로 설정되기 전에 한 번만 평가됩니다. *exprb*의 기본값은 1입니다. *exprt*가 생략되면, 일부 다른 조건이 중지시키지 않는 한 루프는 무기한으로 실행합니다. 예:

```
Do I=3 to -2 by -1      /* Displays: */
  say i                 /*      3      */
end                     /*      2      */
                        /*      1      */
                        /*      0      */
                        /*     -1      */
                        /*     -2      */
```

숫자가 정수일 필요는 없습니다.

```
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    */
```

제어 변수는 루프 내에서 변경될 수 있고 루프의 반복에 영향을 미칠 수 있습니다. 제어 변수의 값을 변경하면 일반적으로 좋은 프로그래밍 연습으로 간주되지 않지만, 특정 상황에서는 적절할 수 있습니다.

종료 조건은 각 반복 시작에서 테스트됩니다(제어 변수가 단계별로 처리된 후 및 두 번째 및 후속 반복 시). 그러므로 종료 조건이 즉시 충족되는 경우 명령어 그룹은 전체적으로 건너뛸 수 있습니다. 제어 변수가 이름별로 참조된다는 점도 주의하십시오. (예를 들어)복합 이름 A.I가 제어 변수에 대해 사용된 경우, 루프 내에서 I를 변경하면 제어 변수가 변경됩니다.

제어된 루프의 실행은 FOR 구문으로 추가적으로 제한될 수 있습니다. 이 경우, *exprf*를 지정해야 하며 모든 양의 정수 또는 0(영)으로 평가해야 합니다. 간단한 반복적 루프에서 반복 카운트처럼 작용하고, 다른 조건이 막지 않으면 루프 주위에 반복의 수에 제한을 설정합니다. TO 및 BY 표현식과 같이 한 번만 평가됩니다. DO 명령어가 처음 처리되며 제어 변수 전에 초기값을 수신합니다. TO 조건과 같이, FOR 조건은 각 반복 초기에 확인됩니다. 예:

```
Do Y=0.3 to 4.3 by 0.7 for 3 /* Displays: */
  say Y                     /*      0.3    */
end                         /*      1.0    */
                           /*      1.7    */
```

제어된 루프에서, 제어 변수를 설명하는 *name*은 END 절에 지정될 수 있습니다. 이 *name*은 DO 절에서 대소문자를 제외하고 모든 면에서 *name*과 일치해야 합니다(복합 변수에 대한 대체가 수행되지 않음에 주의). 그렇지 않은 경우 구문 오류가 발생합니다. 최소 오버헤드로 루프 중첩이 자동으로 확인될 수 있게 합니다. 예:

```
Do K=1 to 10
  ...
  ...
End k /* Checks that this is the END for K loop */
```

**참고:** REXX 산술 규칙이 제어 변수를 단계별 처리 계산에 적용되기 때문에, NUMERIC 설정은 제어 변수의 연속되는 값에 영향을 미칠 수 있습니다.

## 조건부 구문(WHILE 및 UNTIL)

조건부 구문은 반복적 DO 루프의 반복을 수정할 수 있습니다. 루프의 종료를 발생시킬 수 있습니다. *repetitor*의 양식(없음, FOREVER, 단순 또는 제어된)을 따를 수 있습니다. WHILE 또는 UNTIL을 지정하는 경우, *exprw* 또는 *expru*는 개별적으로 모든 변수의 최신 값을 사용하여 루프 주위에 매번 평가되며(0 또는 1로 평가되어야 함), *exprw*가 0으로 평가되거나 *expru*가 1로 평가된 경우 루프가 종료됩니다.

WHILE 루프의 경우 조건은 명령어 그룹의 맨 위에 평가됩니다. UNTIL 루프의 경우 조건은 맨 아래에서 평가됩니다. 제어 변수 전에 단계별로 처리됩니다. 예:

```
Do I=1 to 10 by 2 until i>6
  say i
end
/* Displays: "1" "3" "5" "7" */
```

**참고:** LEAVE 또는 ITERATE 명령어를 사용하면 반복적 루프의 실행을 수정할 수도 있습니다.

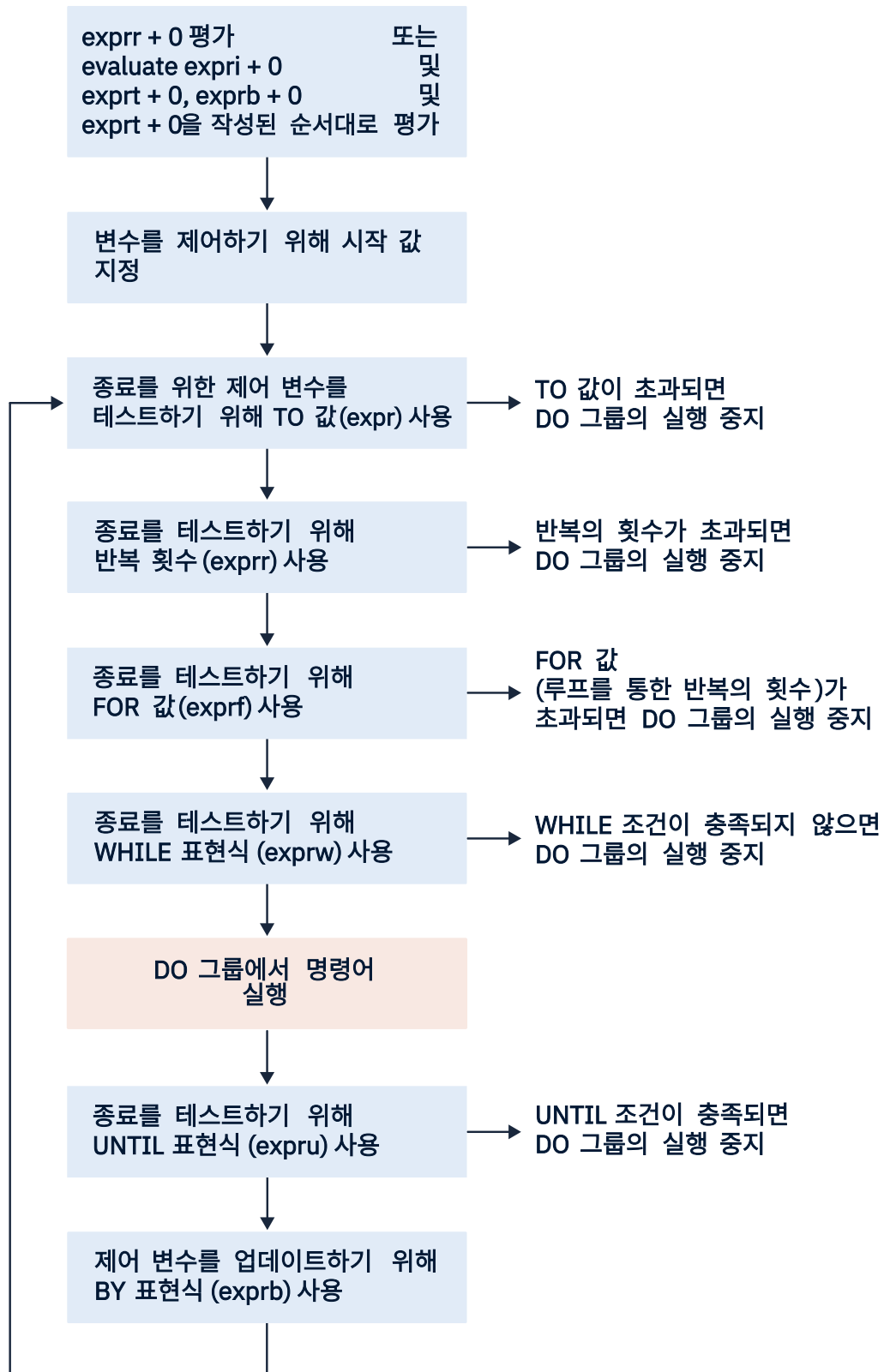
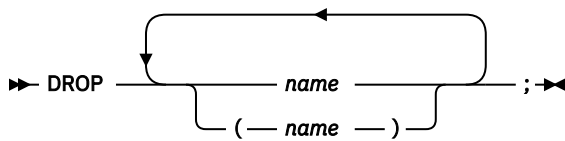


그림 47. DO 루프의 개념

## DROP

DROP은 변수를 지정 해제합니다. 즉 초기화되지 않은 원래 상태로 복원합니다.



*name*이 괄호로 둘러싸여 있지 않으면, 삭제하려는 변수를 식별하고, 하나 이상의 공백 또는 주석으로 다른 *name*에서 구분된 유효한 변수 이름인 기호여야 합니다.

소괄호가 단일 *name*을 둘러싸면, 해당 값은 삭제할 변수의 보조 목록으로 사용됩니다.(공백은 괄호 내부나 외부에서 필요하지 않지만 추가할 수 있습니다.) 이 보조 목록은 괄호가 허용되지 않는다는 점을 제외하고 원래 목록과 동일한 규칙을 따라야 합니다(즉, 공백으로 구분된 유효한 변수 이름).

변수는 왼쪽에서 오른쪽의 순서로 삭제됩니다. 두 번 이상 이름을 지정하거나 알려지지 않는 변수를 삭제하는 것은 오류가 아닙니다. 표시된 변수가 이름 지정되며(168 페이지의 『PROCEDURE』 참조), 이전 생성된 변수는 삭제됩니다.

## 예제

```

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. */

```

여기에서, 소괄호의 변수 이름은 보조 목록으로 사용됩니다.

```

mylist='c d e'
drop (mylist) f
/* Drops the variables C, D, E, and F          */
/* Does not drop MYLIST                       */

```

스탬을 지정하고(즉, 마지막 문자로서 하나의 마침표만 포함하는 기호) 해당 스탬으로 시작하는 모든 변수를 삭제합니다.

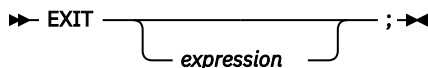
```

Drop z.
/* Drops all variables with names starting with Z. */

```

## EXIT

EXIT는 무조건적으로 프로그램을 중단합니다.



선택적으로 EXIT는 문자열을 호출자로 리턴합니다. 내부 루틴이 현재 실행되고 있더라도, 프로그램은 바로 중지합니다. 내부 루틴이 활성이지 않으면, RETURN(171 페이지의 『RETURN』 참조)과 EXIT는 실행 중인 프로그램에 대한 결과에서 동일합니다.

*expression*을 지정하는 경우 평가되고 프로그램이 중지되면 평가에서 유래하는 문자열이 호출자에 다음에 전달됩니다. 예:

```

j=3
Exit j*4
/* Would exit with the string '12' */

```

*expression*을 지정하지 않은 경우 데이터도 호출자에 다음에 전달되지 않습니다. 프로그램이 외부 함수로 호출된 경우, RETURN이 사용된 경우 즉시, 또는 EXIT가 사용된 경우 호출자로 리턴 시에만 오류로 감지됩니다.

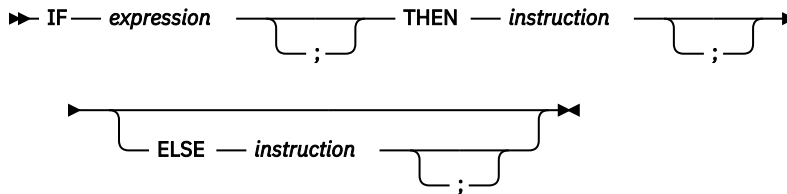
프로그램의 "끝까지 실행"은 명령어 EXIT와 항상 동일하며 전체 프로그램을 중지하고 결과 문자열이 없음을 리턴합니다.

**참고:** 프로그램이 명령 인터페이스를 통하여 호출되면 리턴된 값을 기본 운영 체제에서 허용 가능한 리턴 코드로 변환하도록 작성하려고 합니다. 변환이 실패하면 기본 운영 체제 때문에 실패할 것으로 생각되므로 SIGNAL ON

SYNTAX를 사용한 트래밍이 적용되지 않습니다. 리턴된 문자열은 값이 일반 레지스터에 맞는 정수여야 합니다 (즉, 범위  $-2^{31}$ 에서  $2^{31}-1$ 까지여야 함).

## IF

IF는 *expression*의 평가에 따라 명령어 또는 명령어 그룹을 조건부로 처리합니다. *expression*이 평가되며 결과가 0 또는 1이 되어야 합니다.



결과가 1인 경우(true)에만 THEN 다음의 명령이 처리됩니다. ELSE를 지정하는 경우, 평가 결과가 0인 경우(false)에만 ELSE 다음의 명령이 처리됩니다.

### 예제

```

if answer='YES' then say 'OK!'
    else say 'Why not?'

```

ELSE 절이 THEN 파트의 마지막 절과 같은 행에 있는 경우 ELSE 앞에 세미콜론이 필요하다는 점을 기억하십시오.

```

if answer='YES' then say 'OK!'; else say 'Why not?'

```

ELSE는 동일한 레벨에 가장 가까운 IF로 바인딩합니다. 다음 예와 같이 IF 구성이 중첩되면 NOP 명령어를 사용하여 오류와 가능한 혼동을 줄일 수 있습니다.

```

If answer = 'YES' Then
    If name = 'FRED' Then
        say 'OK, Fred.'
    Else
        nop
Else
    say 'Why not?'

```

### 참고:

1. *instruction*은 DO, SELECT 또는 IF 명령어 자체와 같이 복잡한 구성을 포함하여, 지정, 명령 또는 키워드 명령일 수 있습니다. null 절은 명령어가 아니므로 THEN 또는 ELSE 다음에 세미콜론(또는 레이블)을 두면 더미 명령어를 두는 것과 같지 않습니다(PL/I에서와 같이). NOP 명령어는 이 용도로 제공됩니다.
2. 기호 THEN은 *expression* 내에서 사용될 수 없습니다. 절을 시작할 필요가 없다는 점에서 키워드 THEN이 다르게 처리됩니다. IF 절에서의 표현식은 THEN으로 끝나지 않으며 ;가 필요하지 않습니다. 그렇지 않은 경우 다른 컴퓨터 언어에 익숙한 사람은 상당한 난이도를 경험할 것입니다.

## INTERPRET

INTERPRET는 *expression*을 평가하여 동적으로 빌드된 명령어를 처리합니다.

```

INTERPRET — expression — ;

```

결과 문자열이 프로그램에 삽입된 행인 것처럼(DO; 및 END;로 대괄호 처리된) *expression*이 평가된 다음 처리됩니다(해석됨).

모든 명령어(INTERPRET 명령어 포함)이 허용되지만 DO...END 및 SELECT...END와 같은 구성이 완료되어야 함에 유의하십시오. 예를 들어, 반복적인 DO...END 구성 전체도 포함하지 않는 한 해석되는 명령어의 문자열은 LEAVE 또는 ITERATE 명령어를 포함할 수 없습니다(반복적 DO 루프 내에서만 유효함).

세미콜론이 제공되지 않은 경우 실행 중 표현식 종료 시 세미콜론이 표시됩니다.

## 예제

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!                          */
```

## 참고:

1. 레이블 절은 해석된 문자열에 허용되지 않습니다.
2. INTERPRET 명령어의 개념에 처음이거나 이해하지 못하는 결과를 얻게 되는 경우, 결과에서 TRACE R 또는 TRACE I로 이를 실행하면 도움이 된다는 것을 알 수 있습니다. 예:

```
/* Here is a small REXX program. */
Trace Int
name='Kitty'
indirect='name'
interpret 'say "Hello" indirect'!"!"
```

실행 시 다음 추적을 제공합니다.

```
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!
```

여기서 3행과 4행은 5행에서 사용된 변수를 설정합니다. 5행의 실행은 두 단계로 처리됩니다. 리터럴 문자열, 변수 (INDIRECT) 및 다른 리터럴 문자열을 사용하여, 먼저 해석될 문자열이 빌드됩니다. 원래 프로그램의 실제 일부인 것처럼 완전한 결과 문자열이 해석됩니다. 새 절이기 때문에 그와 같이 처리된 다음(5행 아래 두 번째 \*\* 추적 플래그) 처리됩니다. 다시 리터럴 문자열이 변수(NAME) 및 다른 리터럴의 값으로 연결되며 최종 결과(Hello Kitty!)가 표시됩니다.

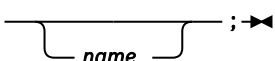
3. 여러 목적으로 INTERPRET 명령어 대신 VALUE 함수를 사용할 수 있습니다(204 페이지의 『VALUE』 참조). 그러므로 다음 행은 마지막 예에서 5행을 대체합니다.

```
say "Hello" value(indirect)!"!"
```

INTERPRET는 일반적으로 둘 이상의 명령문이 함께 해석되는 경우 또는 표현식이 동적으로 평가되는 경우와 같이 특별한 경우에만 필요합니다.

## ITERATE

ITERATE는 반복적 DO 루트 내 플로우를 변경합니다(즉, 단순 DO를 포함한 구조가 아닌 DO 구문).

➡ ITERATE  ; ➡



명령어의 그룹 실행이 중지되며 END 절이 사용된 것처럼 DO 명령어에 제어가 전달됩니다. DO 명령어가 루프로 종료되지 않는 한 제어 변수(있는 경우)가 보통 증가되며 테스트되며 명령어 그룹이 다시 처리됩니다.

*name*은 상수로 사용되는 기호입니다. *name*이 지정되지 않은 경우 ITERATE는 가장 내부 활성 반복적 루프를 단계별 실행합니다. *name*이 지정된 경우, 현재 활성 루프의 제어 변수 이름이어야 하며(가장 내부일 수 있음) 단계별 실행하는 루프입니다. 반복을 위해 선택한 루프 내 활성 루프가 종료됩니다(LEAVE 명령어에 의한 것처럼).

## 예

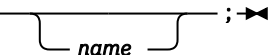
```
do i=1 to 4
  if i=2 then iterate
  say i
end
/* Displays the numbers:  "1" "3" "4" */
```

## 참고:

1. 지정된 경우, *name*은 대소문자를 제외한 모든 면에서 DO 절에서 제어 변수 이름을 지정하는 기호와 일치해야 합니다. 비교가 작성되는 경우 복합 변수의 대체가 실행되지 않습니다.
2. 현재 처리 중인 경우 루프는 활성입니다. 서브루틴이 루프 실행 중 호출되는 경우(또는 INTERPRET 명령어가 처리되는 경우), 서브루틴이 리턴했거나 INTERPRET 명령이 완료했을 때까지 루프는 비활성 상태가 됩니다. ITERATE는 비활성 루프를 단계별 실행하는 데 사용될 수 없습니다.
3. 둘 이상의 활성 루프가 동일한 제어 변수를 사용하면, ITERATE는 가장 내부 루프를 선택합니다.

## LEAVE

LEAVE는 하나 이상의 반복적 DO 루프(즉, 단순 DO가 아닌 모든 DO 구성)를 즉시 종료합니다.

➡ LEAVE  ; ➡

명령어 그룹의 처리가 종료되면 END 절이 발생하고 종료 조건이 충족된 것처럼 END 절 뒤에 오는 명령어에 제어가 전달됩니다. 그러나 종료 시 제어 변수(있는 경우)는 LEAVE 지시사항이 처리될 때 가졌던 값을 포함하게 됩니다.

*name*은 상수로 간주되는 기호입니다. *name*을 지정하지 않으면 LEAVE가 가장 안쪽 활성 반복 루프를 종료합니다. *name*을 지정하면 현재 활성인 루프(가장 안쪽 루프일 수 있음)의 제어 변수 이름이어야 하고 해당 루프(및 그 안에 있는 모든 활성 루프)가 종료됩니다. 그러면 선택한 루프의 DO 절과 일치하며 END 뒤에 오는 절에 전달됩니다.

## 예

```
do i=1 to 5
  say i
  if i=3 then leave
end
/* Displays the numbers:  "1" "2" "3" */
```

## 참고:

1. 지정하는 경우 *name*은 DO 절의 제어 변수에 이름을 지정하는 기호와 대소문자 구분을 제외한 모든 측면에서 일치해야 합니다. 비교가 수행될 때 복합 변수의 대체는 수행되지 않습니다.
2. 현재 처리되고 있는 경우 이 루프는 활성입니다. 루프 실행 중에 서브루틴이 호출되는 경우(또는 INTERPRET 지시사항이 처리되는 경우) 이 서브루틴이 리턴되거나 INTERPRET 지시사항이 완료될 때까지 이 루프는 비활성 상태가 됩니다. LEAVE는 비활성 루프를 종료하는 데 사용할 수 없습니다.
3. 둘 이상의 활성 루프가 동일한 제어 변수를 사용하는 경우 LEAVE는 가장 안쪽 루프를 선택합니다.

## NOP

NOP은 아무 효과가 없는 터미 명령어입니다. THEN 또는 ELSE 절의 대상으로 유용할 수 있습니다.

➡ NOP — ;➡

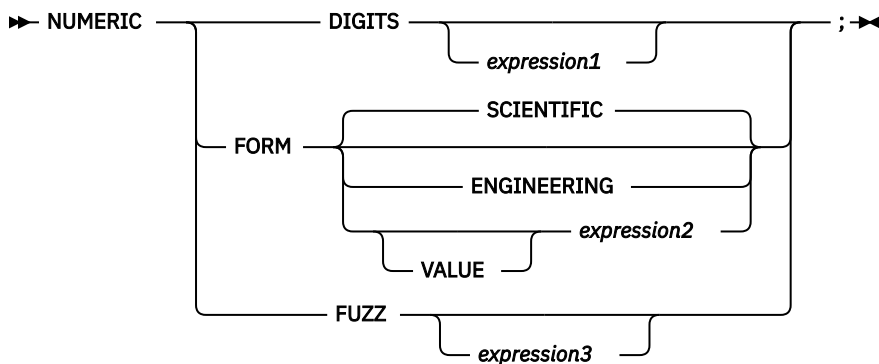
## 예

```
Select
  when a=c then nop          /* Do nothing */
  when a>c then say 'A > C'
  otherwise      say 'A < C'
end
```

**참고:** NOP 대신 추가 세미콜론을 넣으면 널인 절만 삽입하게 되고 이는 무시됩니다. 두 번째 WHEN 절은 THEN 뒤에 예상되는 첫 번째 명령어로 표시되고 따라서 구문 오류로 처리됩니다. NOP은 참인 명령어이고 따라서 THEN 절의 올바른 대상입니다.

## NUMERIC

NUMERIC는 프로그램이 산술 연산을 수행하는 방법을 변경합니다.



이 명령어의 옵션은 225 페이지의 『제 21 장 숫자와 산술 오퍼레이션』 및 232 페이지의 『오류』에서 자세하게 설명되며 다음으로 요약됩니다.

### NUMERIC DIGITS

이 옵션은 산술 연산과 산술 기본 제공 함수가 평가받는 자릿수를 제어합니다. *expression1*이 생략된 경우 자릿수의 기본값은 9자리입니다. 그렇지 않은 경우, *expression1*은 모든 양수의 정수로 평가되며 현재 NUMERIC FUZZ 설정보다 커야 합니다.

DIGITS 값에 제한이 없지만(사용 가능한 스토리지 양 제외, 자세한 정보는 131 페이지의 『제 17 장 REXX 일반 개념』에서 노트 참조) 높은 정밀도는 상당한 처리 시간을 요구할 수 있음에 주의하십시오. 가능한 한 어디서나 기본값을 사용하도록 권장합니다.

DIGITS 기본 제공 함수와 함께 현재 NUMERIC DIGITS 설정을 검색할 수 있습니다. 191 페이지의 『DIGITS』의 내용을 참조하십시오.

### NUMERIC FORM

이 옵션은 산술 연산과 산술 기본 제공 함수의 결과로 사용한 지수 표기법 REXX의 양식을 제어합니다. SCIENTIFIC(이 경우 소수점 앞에 0(영)이 아닌 하나의 숫자만 표시) 또는 ENGINEERING(이 경우 10의 거듭제곱은 항상 3의 배수)일 수 있습니다. 기본값은 SCIENTIFIC입니다. 서브키워드 SCIENTIFIC 또는 ENGINEERING이 직접 FORM을 설정하거나 VALUE 뒤에 오는 표현식(*expression2*) 평가 결과에서 가져옵니다. 이 경우, 결과는 SCIENTIFIC 또는 ENGINEERING이어야 합니다. *expression2*가 기호나 리터럴 문자열로 시작되지 않은 경우(즉, 연산자 문자나 소괄호와 같은 특수 문자로 시작된 경우) 서브키워드 VALUE를 생략할 수 있습니다.

FORM 기본 제공 함수로 현재 NUMERIC FORM 설정을 검색할 수 있습니다. 193 페이지의 『FORM』의 내용을 참조하십시오.

### NUMERIC FUZZ

이 옵션은 숫자 비교 연산 중 전체 자릿수에서 무시되는 자리를 제어합니다. 229 페이지의 『숫자 비교』의 내용을 참조하십시오. *expression3*을 생략하는 경우 기본값은 0자리입니다. 그렇지 않으면, *expression3*은

0 또는 모든 양수의 정수로 평가되어야 하며, 현재 NUMERIC DIGITS 설정에 따라 필요한 경우 반올림되고, 현재 NUMERIC DIGITS 설정보다 작아야 합니다.

모든 숫자 비교 중 NUMERIC FUZZ는 임시로 NUMERIC DIGITS의 값을 NUMERIC FUZZ 값만큼 줄입니다. 해당 숫자를 비교 중 DIGITS에서 FUZZ 자리를 뺀 자릿수 미만으로 뺀 다음 0과 비교합니다.

현재 NUMERIC FUZZ 설정을 FUZZ 기본 제공 함수로 검색할 수 있습니다. [194 페이지의 『FUZZ』](#)의 내용을 참조하십시오.

**참고:** 세 개의 숫자 설정이 내부 및 외부 서브루틴과 함수 호출에서 자동으로 저장됩니다. 세부사항은 CALL 명령어([153 페이지의 『CALL』](#))를 참조하십시오.

## OPTIONS

OPTIONS는 특수 요청이나 매개변수를 언어 프로세서에 전달합니다. 예를 들면, 이는 언어 프로세서 옵션이거나 특수 문자 세트를 정의할 수 있습니다.

►► OPTIONS — *expression* — ;◄◄

*expression*이 평가되고 한 번에 하나의 단어로 결과가 검사됩니다. 언어 프로세서는 단어를 대문자로 변환합니다. 언어 프로세서가 단어를 인식하면 이를 따릅니다. 인식되지 않는 단어는 무시되며 다른 프로세서에 명령어로 가정됩니다.

언어 프로세서는 다음 단어를 인식합니다.

### ETMODE

리터럴 문자열과 기호를 지정하고 DBCS 문자를 포함한 주석이 유효한 DBCS 문자열인지 확인합니다. 이 옵션을 사용하는 경우 프로그램의 첫 번째 명령어여야 합니다.

*expression*이 외부 함수 호출인 경우, 예를 들어 OPTIONS 'GETETMOD' () 이고 프로그램이 DBCS 리터럴 문자열을 포함하는 경우, 함수 이름을 따옴표로 묶어 옵션이 적용되기 전에 전체 프로그램이 스캔되지 않도록 하십시오. 프로그램에서 DBCS 리터럴 문자열 해석의 오류 가능성 때문에 ETMODE를 설정하도록 내부 함수 호출을 사용하지 않는 것이 좋습니다.

### NOETMODE

리터럴 문자열과 기호 및 DBCS 문자를 포함한 주석이 유효한 DBCS 문자열인지 확인하지 않도록 지정합니다. NOETMODE가 기본값입니다. 프로그램에서 첫 번째 지시사항이 아니면 언어 프로세서는 이 옵션을 무시합니다.

### EXMODE

논리 문자를 기반으로 혼용 명령문에서 명령어, 연산자 및 함수가 DBCS 데이터를 처리하도록 지정합니다. DBCS 데이터 무결성은 유지보수됩니다.

### NOEXMODE

문자열에서 모든 데이터를 바이트 기반으로 처리되도록 지정합니다. DBCS 문자의 무결성이 있는 경우, 손실될 수 있습니다. NOEXMODE가 기본값입니다.

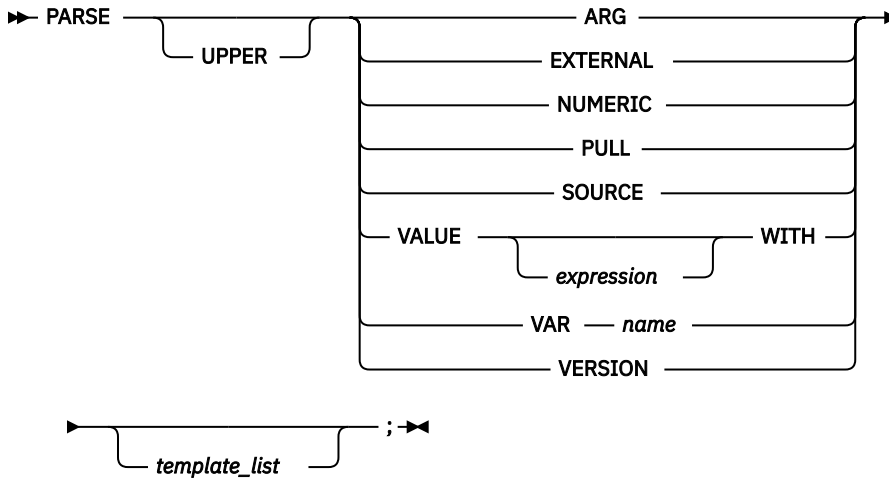
### 참고:

1. 언어 프로세서의 스캐닝 프로시저 때문에, 리터럴 문자열, 기호 또는 주석에 DBCS 문자를 포함한 프로그램에서 첫 번째 명령어로 OPTIONS 'ETMODE' 명령어를 배치해야 합니다. OPTIONS 'ETMODE'를 첫 번째 명령어로 두지 않고 프로그램에서 나중에 사용하는 경우, 오류 메시지 CIREX488E를 수신합니다. 프로그램의 첫 번째 명령어로 배치하는 경우 이후 모든 사용이 무시됩니다. 표현식이 레이블 검색을 시작하는 항목을 포함하는 경우, 레이블 검색 프로세스 중 토큰화된 모든 절은 ETMODE의 현재 설정 내에 토큰화됩니다. 그러므로 프로그램에서 첫 번째 명령문이면 기본값은 NOETMODE입니다.
2. DBCS 리터럴과 DBCS 주석을 포함한 프로그램의 적절한 스캐닝을 확인하려면, 단어 ETMODE, NOETMODE, EXMODE 및 NOEXMODE를 리터럴 문자열(즉, 따옴표 표시로 묶인)을 OPTIONS 명령어에 입력하십시오.
3. EXMODE 설정이 저장되고 서브루틴과 함수 호출 전체에 걸쳐 복원됩니다.
4. DBCS 문자를 1바이트 EBCDIC 문자와 구별하도록, DBCS 문자의 시퀀스는 SO(Shift-Out) 문자 및 SI(Shift-In) 문자로 묶입니다. SO 및 SI 문자의 16진수 값은 각각 X'0E' 및 X'0F'입니다.
5. OPTIONS 'ETMODE'를 지정하면, 리터럴 문자열 내 DBCS 문자는 리터럴 문자열에서 닫힌 따옴표로 검색에서 제외됩니다.

6. 단어 ETMODE, NOETMODE, EXMODE 및 NOEXMODE는 결과에 여러 번 표시될 수 있습니다. 적용되는 단어는 ETMODE-NOETMODE 및 EXMODE-NOEXMODE 쌍 사이에 지정된 유효한 마지막 문자로 판별됩니다.

## PARSE

PARSE는 구문 분석 규칙에 따라 데이터(여러 소스로부터)를 하나 이상의 변수로 지정합니다.



211 페이지의 『제 20 장 구문 분석』의 내용을 참조하십시오.

*template\_list*는 단일 템플릿인 경우가 많지만 쉽표로 구분되는 여러 템플릿일 수 있습니다. 지정되면, 각 템플릿은 공백이나 패턴 또는 둘 다를 분리된 기호 목록입니다.

각 템플릿은 단일 소스 문자열에 적용됩니다. 다중 템플릿을 지정하면 문자열 오류가 아니지만 PARSE ARG 변형만이 둘 이상의 널리 아닌 소스 문자열을 제공할 수 있습니다. 219 페이지의 『여러 문자열 구문 분석』의 내용을 참조하십시오.

템플릿을 지정하지 않은 경우, 변수도 설정되지 않지만 조치가 필요한 경우 구문 분석에 대한 데이터로 준비하기 위해 수행됩니다. 그러므로 PARSE EXTERNAL 및 PARSE PULL의 경우, 데이터 문자열이 큐에서 제거되며, PARSE LINEIN(및 큐가 비어 있는 경우 PARSE PULL)의 경우, 행이 기본 입력 스트림으로부터 사용되고 PARSE VALUE의 경우, *expression*이 평가됩니다. PARSE VAR의 경우, 지정된 변수가 액세스됩니다. 값이 없는 경우 NOVALUE 조건(사용으로 설정된 경우)이 발생합니다.

UPPER 옵션을 지정한 경우, 구문 분석될 데이터는 먼저 대문자로 변환됩니다(즉, 소문자 a-z를 대문자 A-Z로). 그렇지 않은 경우, 구문 분석 중 대문자 변환이 수행되지 않습니다.

다음 목록은 PARSE 명령어의 각 변형에 대한 데이터를 설명합니다.

### PARSE ARG

프로그램이나 내부 루틴에 전달된 문자열을 입력 인수로 구문 분석합니다. (세부사항 및 예는 152 페이지의 『ARG』의 내용을 참조하십시오.)

**참고:** ARG 기본 제공 함수로 REXX 프로그램이나 내부 루틴으로 인수 문자열을 검색하거나 확인할 수도 있습니다(183 페이지의 『ARG(인수)』의 내용을 참조하십시오.)

### PARSE EXTERNAL

REXX/CICS에 제공된 비SAA 서브키워드입니다. 터미널 입력 버퍼로부터의 다음 문자열이 구문 분석됩니다. 이 큐는 사용자 콘솔 입력이나 메시지와 같이 외부 비동기 이벤트의 결과인 데이터를 포함할 수 있습니다. 해당 큐가 비어 있는 경우, 콘솔은 결과를 읽습니다. 이 메커니즘이 프로그램 스택이 방해받을 수 없는 특수 애플리케이션(디버깅과 같이)이 아닌 PULL이 더 일반적인 일반 콘솔 입력에 사용하지 않아야 하는 점에 유의하십시오.

### PARSE NUMERIC

REXX/CICS에 제공된 비SAA 서브키워드입니다. 현재 숫자 제어(NUMERIC 명령어에서 설정된 것처럼)가 사용 가능합니다. 이러한 제어는 DIGITS FUZZ FORM 순입니다. 예:

```
Parse Numeric Var1
```

이 명령어 후, Var1은 9 0 SCIENTIFIC과 동일합니다. 164 페이지의 『NUMERIC』 및 191 페이지의 『DIGITS』, 193 페이지의 『FORM』, 194 페이지의 『FUZZ』에서 기본 제공 함수를 참조하십시오.

## PARSE PULL

외부 데이터 큐에서 다음 문자열을 구문 분석합니다. 외부 데이터 큐가 비어 있는 경우, PARSE PULL은 기본 입력 스트림(사용자의 터미널)에서 행을 읽고 필요한 경우 행이 완료될 때까지 프로그램이 일시정지됩니다. PUSH 및 QUEUE 명령어를 사용하여 큐의 헤드나 후미에 데이터를 추가할 수 있습니다. 현재 QUEUED 기본 제공 함수를 가진 큐에서 행 수를 찾을 수 있습니다. 197 페이지의 『QUEUED』의 내용을 참조하십시오. 시스템에서 다른 프로그램은 큐를 변경하고 REXX로 작성된 프로그램과의 통신 수단으로 이를 사용할 수 있습니다. PULL 명령어(169 페이지의 『PULL』)도 참조하십시오.

**참고:** PULL 및 PARSE PULL은 프로그램 스택에서 읽힙니다. 비어 있는 경우 터미널 입력 버퍼에서 읽습니다. 너무 많이 비어 있는 경우 콘솔에서 읽습니다.

## PARSE SOURCE

실행 중인 프로그램의 소스를 설명하는 데이터를 구문 분석합니다. 프로그램이 실행되는 동안 언어 프로세서는 수정되는(변경되지 않는) 문자열을 리턴합니다. PARSE SOURCE 명령어는 다음 토큰을 포함한 소스 문자열을 리턴합니다.

1. 문자 CICS.
2. 프로그램이 호스트 명령으로 호출되는지 여부에 따라 표현식의 함수 호출로부터, CALL 명령어로, 또는 서버 프로세스로 문자열 COMMAND, FUNCTION 또는 SUBROUTINE.
3. 대문자로 된 exec의 이름입니다. 파일의 이름(RFS) 또는 MVS 파티션된 데이터 세트 또는 exec가 원래 로드된 DDNAME/member입니다. 세 개의 형식은 다음과 같습니다.
  - DDNAME - 멤버
  - RFS 완전한 파일 ID
  - 데이터 세트 이름(멤버)
4. 항상 REXXCICS인 초기(기본값) 호스트 명령 환경.
5. 특정 CICS 환경의 ID로, 이 경우 CICS-TS입니다.

## PARSE VALUE

*expression*을 평가하는 결과인 데이터를 구문 분석합니다. *expression*을 지정하지 않은 경우, null 문자열이 사용됩니다. WITH는 이 컨텍스트에서 서브키워드이며, *expression* 내 기호로 사용될 수 없다는 점에 유의하십시오.

그러므로, 다음 예제는 현재 시간을 가져오며 해당 구성 파트로 분할됩니다.

```
PARSE VALUE time() WITH hours ':' mins ':' secs
```

## PARSE VAR *name*

변수 *name*의 값을 구문 분석합니다. *name*은 변수 이름으로서 유효한 기호여야 합니다(즉, 주기 또는 숫자로 시작할 수 없음). 템플릿에 나타나지 않으면 변수 *name*은 변경되지 않습니다. 예를 들면 다음 명령어는 첫 번째 단어를 *string*에서 제거하고, 변수 *word1*에 두며, 나머지를 *string*에 다시 지정합니다.

```
PARSE VAR string word1 string
```

마찬가지로 구문 분석되기 전에 또한 다음 예제는 *string*부터 대문자까지 데이터를 변환합니다.

```
PARSE UPPER VAR string word1 string
```

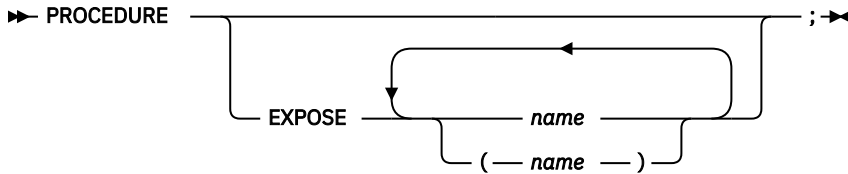
## PARSE VERSION

언어 레벨 및 언어 프로세서의 날짜를 설명하는 정보를 구문 분석합니다. 이 정보는 공백으로 구분된 5개의 단어로 구성됩니다.

1. 문자열 REXX370, 370 구현 표시.
2. 언어 레벨 설명(예: 3.48).
3. 언어 프로세서 릴리스 날짜(예: 05 April 2000).

## PROCEDURE

내부 루틴(서브루틴 또는 함수) 내 PROCEDURE는 그 뒤에 오는 명령어에 대해 변수를 알 수 없도록 설정하여 변수를 보호합니다.



RETURN 명령어가 처리되고 나면 원래 변수 환경이 복원되고 루틴에서 사용된 모든 변수는(노출되지 않음) 삭제됩니다. (노출된 변수는 PROCEDURE 명령어가 노출된 루틴의 호출자에 속한 변수입니다. 루틴이 변수를 참조하거나 변경하는 경우 변수의 원본(호출자의)이 사용됩니다.) 내부 루틴은 PROCEDURE 명령어를 포함하지 않아도 됩니다. 이 경우 조작 중인 변수는 호출자가 "소유"하는 변수입니다. 사용된 경우 PROCEDURE 명령어는 CALL 또는 함수 호출 후 처리되는 첫 번째 명령어여야 합니다. 즉, 레이블 뒤에 오는 첫 번째 명령어여야 합니다.

EXPOSE 옵션을 사용하는 경우 *name*에 의해 지정되는 모든 변수가 노출됩니다. 이에 대한 모든 참조는(설정 및 삭제 포함) 호출자가 소유한 변수 환경을 참조합니다. 따라서 기존 변수의 값에 액세스할 수 있고 모든 변경사항은 루틴에서 RETURN 시에도 지속적입니다. *name*이 소괄호로 묶여 있지 않은 경우 이는 노출시킬 변수를 식별하고, 한 개 이상의 공백으로 다른 *name*에서 분리되는 올바른 변수 이름인 기호여야 합니다.

소괄호가 단일 *name*을 묶은 경우 변수 *name*이 노출된 후 *name*의 값은 즉시 변수의 보조 목록으로 사용됩니다. (공백은 소괄호의 내부 또는 외부에 필요하지 않지만 추가할 수는 있습니다.) 이 보조 목록은 소괄호가 허용되지 않는다는 점을 제외하고 원래 목록(즉, 공백으로 분리된 올바른 변수 이름)과 동일한 규칙을 따라야 합니다.

변수는 왼쪽에서 오른쪽 순서로 노출됩니다. 이름을 한 번 이상 지정하거나 호출자가 변수로 사용하지 않은 이름을 지정하는 것은 오류가 아닙니다.

노출되지 않은 기본 프로그램의 모든 변수는 계속 보호됩니다. 따라서 일부 제한된 호출자의 변수 세트는 액세스 가능한 상태로 설정될 수 있으며 이러한 변수를 변경할 수 있습니다(또는 이 세트의 새 변수를 작성할 수 있음). 이러한 모든 변경사항은 루틴에서 RETURN 시 호출자에게 표시됩니다.

### 예제

```
/* 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 목록의 Z.J가 J 앞에 배치된 경우 호출자의 J 값은 이 때 표시되지 않으므로 Z.1은 노출되지 않습니다. 보조 목록의 변수도 왼쪽에서 오른쪽으로 노출됩니다.

```
/* 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
```

보조 목록을 사용하여 다수의 변수를 한 번에 쉽게 노출시키거나 또는 VALUE 기본 제공 함수를 사용하여 이름 지정된 변수를 동적으로 조작할 수 있습니다.

```
/* 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
```

*stem*을 *name*으로 지정하면 이 스템 및 해당 스템 내 가능한 모든 복합 변수를 노출시킵니다. (144 페이지의 『스템』 참조)

```
/* 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
```

변수가 모든 중간 PROCEDURE 명령어에 포함되었는지 확인하여 여러 번의 루틴 생성을 통해 변수를 노출시킬 수 있습니다.

루틴이 호출되는 방식에 대한 세부사항 및 예제는 CALL 명령어 153 페이지의 『CALL』 및 함수 설명 179 페이지의 『제 19 장 함수』의 내용을 참조하십시오.

## PULL

PULL은 프로그램 스택에서 문자열을 읽습니다. 프로그램 스택이 비어 있는 경우 PULL은 현재 터미널 입력 디바이스로부터 읽기를 시도합니다.

```
➤ PULL ┌──────────────────┐ ; ➤
      │ template_list │
```

PULL은 다음 명령어의 짧은 양식입니다.

```
➤ PARSE UPPER PULL ┌──────────────────┐ ; ➤
                   │ template_list │
```

현재 큐 헤드가 한 개의 문자열로 읽힙니다. *template\_list*를 지정하지 않고는 추가 조치가 수행되지 않습니다(따라서 문자열은 효과적으로 버려짐). 지정하는 경우 *template\_list*는 일반적으로 단일 템플릿이며 이는 공백 또는 패턴 또는 둘 다로 분리되는 기호의 목록입니다. (*template\_list*는 쉼표로 구분된 여러 개의 템플릿일 수 있지만 PULL은 한 개의 소스 문자열만 구문 분석합니다. 여러 개의 쉼표로 구분된 템플릿을 지정하는 경우 첫 번째 변수 외의 템플릿 내 변수에 널 문자열이 지정됩니다.) 이 문자열은 대문자로 변환되고(즉, 소문자 a-z가 대문자 A-Z로 변환됨) 구문 분석에 대한 섹션(211 페이지의 『제 20 장 구문 분석』)에 설명된 규칙에 따라 변수로 구문 분석됩니다. 대문자 변환을 원하지 않는 경우 PARSE PULL 명령어를 사용하십시오.

참고:



1. 외부 데이터 큐의 REXX/CICS 구현은 프로그램 스택입니다. 언어 프로세서가 프로그램 스택으로부터 행을 읽습니다. 프로그램 스택이 비어 있는 경우 터미널 읽기가 발생합니다. 사용자를 위한 프로그램 스택은 \SYSTEM\userid\\*PROGSTACK\*라는 이름의 RLS 큐에 있습니다.
2. PULL로 인해 터미널로부터 읽기가 발생하는 경우 PULL 명령이 완료될 때 변수 PULLKEY가 설정됩니다. 이는 PULL 명령에 대한 응답으로 압축된 AID 키의 이름을 포함합니다(예: ENTER, PFKEY 1, PAKEY 1, MSR, PEN 또는 CLEAR).

이름 지정된 큐에 대한 정보는 REXX 목록 시스템 LPULL 명령 [285 페이지의 『LPULL』](#)의 내용을 참조하십시오.

## 예

```
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.'
```

여기에서는 더미 플레이스홀더인 마침표(.)가 템플릿에서 사용되어 사용자가 입력하는 첫 번째 단어를 격리시킵니다.

외부 데이터 큐가 비어 있는 경우 콘솔 읽기가 발행되고 필요한 경우 행이 완료될 때까지 프로그램이 일시정지됩니다.

QUEUED 기본 제공 함수([197 페이지의 『QUEUED』](#) 참조)는 현재 프로그램 스택에 있는 행의 수를 리턴합니다.

## PUSH

PUSH는 *expression* 후입선출(LIFO)의 평가의 결과 문자열을 외부 데이터 큐에 스택합니다.

```
➡➡ PUSH ┌──────────┐ ;➡➡
          │ expression │
```

*expression*을 지정하지 않으면 널 문자열이 스택됩니다.

**참고:** 외부 데이터 큐의 REXX/CICS 구현은 프로그램 스택입니다. 언어 프로세서가 프로그램 스택으로부터 행을 읽습니다. 프로그램 스택이 비어 있는 경우 터미널 읽기가 발생합니다. 사용자를 위한 프로그램 스택은 \SYSTEM\userid\\*PROGSTACK\*라는 이름의 RLS 큐에 있습니다.

이름 지정된 큐에 대한 정보는 REXX 목록 시스템 LPUSH 명령 [285 페이지의 『LPUSH』](#)의 내용을 참조하십시오.

## 예

```
a='Fred'
push      /* Puts a null line onto the queue */
push a 2  /* Puts "Fred 2" onto the queue */
```

QUEUED 기본 제공 함수([197 페이지의 『QUEUED』](#)에 설명되어 있음)는 현재 외부 데이터 큐에 있는 행의 수를 리턴합니다.

## QUEUE

QUEUE는 *expression*의 결과로 발생하는 문자열을 외부 데이터 큐의 후미에 추가합니다. 즉, 이는 추가된 선입선출(FIFO)입니다.

```
➡➡ QUEUE ┌──────────┐ ;➡➡
          │ expression │
```

*expression*을 지정하지 않으면 널 문자열이 큐 대기됩니다.



**참고:** 외부 데이터 큐의 REXX/CICS 구현은 프로그램 스택입니다. 언어 프로세서가 프로그램 스택으로부터 행을 읽습니다. 프로그램 스택이 비어 있는 경우 터미널 읽기가 발생합니다. 사용자를 위한 프로그램 스택은 \SYSTEM\userid\\*PROGSTACK\*라는 이름의 RLS 큐에 있습니다.


## 예

```
a='Toft'  
queue a 2 /* Enqueues "Toft 2" */  
queue /* Enqueues a null line behind the last */
```

QUEUED 기본 제공 함수(197 페이지의 『QUEUED』에 설명되어 있음)는 현재 외부 데이터 큐에 있는 행의 수를 리턴합니다.

## RETURN

RETURN은 REXX 프로그램 또는 내부 루틴에서 해당 호출 지점으로 제어(및 가능한 경우 결과)를 리턴합니다.

➡ RETURN  ;➡

활성인 내부 루틴(서브루틴 또는 함수)이 없는 경우 RETURN 및 EXIT는 실행되고 있는 프로그램에 대한 효과 측면에서 동일합니다. 160 페이지의 『EXIT』의 내용을 참조하십시오.

서브루틴이 실행되고 있는 경우(CALL 명령어 참조) *expression*(있는 경우)이 평가되고 제어가 다시 호출자에게 전달되고 REXX 특수 변수 RESULT는 *expression*의 값으로 설정됩니다. *expression*이 생략되는 경우 특수 변수 RESULT는 삭제됩니다(초기화되지 않는 상태가 됨). CALL 시에 저장된 다양한 설정(예: 추적 및 주소)도 복원됩니다. 153 페이지의 『CALL』의 내용을 참조하십시오.

함수가 처리되고 있는 경우 RETURN 명령어에 *expression*을 지정해야 하는 점을 제외하고 수행 조치는 동일합니다. 그러면 *expression*의 결과가 함수가 호출된 위치에서 원래 표현식에 사용됩니다. 세부사항은 179 페이지의 『제 19 장 함수』에서 함수에 대한 설명을 참조하십시오.

PROCEDURE 명령어가 루틴(서브루틴 또는 내부 함수) 내에서 처리된 경우 *expression*이 평가된 후 및 결과가 사용되거나 RESULT에 지정되기 전에 현재 생성의 모든 변수가 삭제됩니다(그리고 이전 생성의 모든 변수는 노출 됨).

## SAY

SAY는 사용자가 표시되는 것을 볼 수 있도록 행을 기본 출력 스트림(터미널)에 씁니다.

➡ SAY  ;➡

*expression*의 결과 길이에는 제한이 없습니다. *expression*을 생략하면 널 문자열이 기록됩니다.

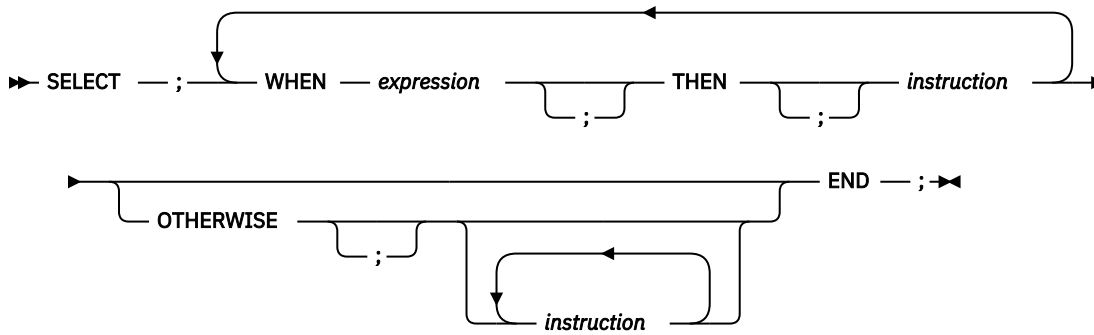
SET TERMOUT 명령을 사용하여 SAY 출력의 경로를 재지정할 수 있습니다.

## 예

```
data=100  
Say data 'divided by 4 =>' data/4  
/* Displays: "100 divided by 4 => 25" */
```

## SELECT

SELECT는 조건부로 여러 대체 명령 중 하나를 호출합니다.



WHEN 다음 *expression*이 차례로 평가되고, 결과적으로 0 또는 1이 되어야 합니다. 결과가 1인 경우 연관된 THEN 다음에 오는 명령어(IF, DO 또는 SELECT와 같은 복합 명령어일 수 있음)이 처리되며 END에 전달합니다. 결과가 0인 경우 제어는 다음 WHEN 절에 전달합니다.

WHEN 표현식이 1로 평가되지 않는 경우, 제어는 있는 경우 OTHERWISE 다음에 명령어에 전달합니다. 이 상황에서, OTHERWISE가 없으면 오류가 발생합니다(그러나 OTHERWISE 다음에 오는 명령어 목록을 생략할 수 있음에 유의하십시오.).

## 예

```

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
  그 외의 경우      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 */

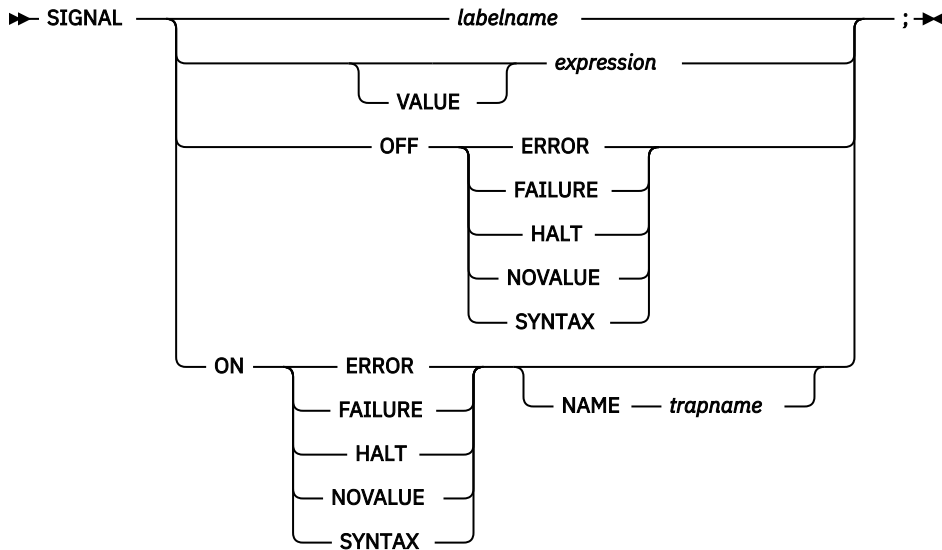
```

## 참고:

1. *instruction*은 DO, IF 또는 SELECT 명령어 자체와 같이 보다 복잡한 구성을 포함하여 지정, 명령 또는 키워드 명령어일 수 있습니다.
2. null 절이 명령어가 아니므로 추가 세미콜론(또는 레이블)을 THEN 절 다음에 두는 것은 더미 명령을 두는 것과 같지 않습니다. NOP 명령어는 이 용도로 제공됩니다.
3. 기호 THEN은 *expression* 내에서 사용될 수 없습니다. 절을 시작할 필요가 없다는 점에서 키워드 THEN이 다르게 처리됩니다. WHEN 절의 표현식을 사용하여 ;(구분 기호) 없이 THEN에서 종료할 수 있습니다.

## SIGNAL

SIGNAL은 제어 플로우에서 비정상적 변경을 야기시키거나(*labelname* 또는 *VALUE expression*), 특정 조건의 트래핑을 제어합니다(ON 또는 OFF를 지정한 경우).



트래핑을 제어하려면 OFF 또는 ON 및 트래핑할 조건을 지정합니다. OFF는 지정된 조건 트랩을 끕니다. ON은 지정된 조건 트랩을 실행합니다. 233 페이지의 『제 22 장 조건 및 조건 트랩』의 내용을 참조하십시오.

제어의 플로우를 변경하기 위해 레이블 이름은 *labelname*에서 파생되거나 VALUE 다음 *expression* 평가 결과에서 가져옵니다. 지정한 *labelname*은 상수로 받아들여지는 리터럴 문자열 또는 기호여야 합니다. *labelname*에 기호를 사용하는 경우, 검색은 알파벳 대소문자와 상관 없습니다. 리터럴 문자열을 사용하는 경우 문자는 대문자여야 합니다. 프로그램에 입력하는 방법과 상관 없이 언어 프로세서가 모든 레이블을 대문자로 변환하기 때문입니다. 마찬가지로 SIGNAL VALUE의 경우 *expression*은 문자열을 대문자로 평가해야 하거나 언어 프로세서가 레이블을 찾지 않습니다. *expression*가 기호 또는 리터럴 문자열로 시작하지 않는 경우(즉, 연산자 문자 또는 소괄호와 같이 특수 문자로 시작된 경우) 서브 키워드 VALUE를 생략할 수 있습니다. 현재 루틴에서 모든 활성 보류 DO, IF, SELECT 및 INTERPRET 명령어가 종료됩니다(즉, 재개할 수 없음). 검색이 프로그램의 맨 위에서 시작했었던 것처럼, 제어는 지정된 이름과 일치하는 프로그램의 첫 번째 레이블에 전달됩니다.

## 예

```
Signal fred; /* Transfer control to label FRED below */
....
Fred: say 'Hi!'
```

검색이 효율적으로 프로그램의 맨 위에서 시작되기 때문에, 복제가 존재하는 경우 제어는 프로그램에서 레이블의 첫 번째 항목에 항상 전달됩니다.

제어가 지정된 레이블에 도달하면, SIGNAL 명령어의 행 번호는 특수 변수 SIGL에 지정됩니다. 제어 전송의 소스를 레이블로 판별하기 위해 SIGL을 사용할 수 있기 때문에 디버깅을 지원할 수 있습니다.

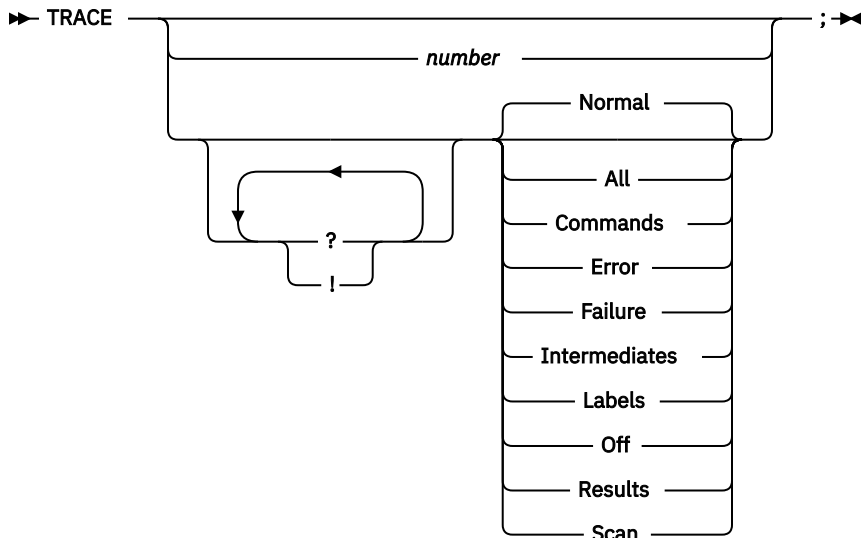
## SIGNAL VALUE 사용

SIGNAL 명령어의 VALUE 양식은 실행 시 이름이 판별되는 레이블로 분기할 수 있습니다. 호출 루틴에서 DO 루트 등이 호출 메커니즘에 의한 종료에 대해 보호되기 때문에 내부 루틴에 대한 다중 방향 호출(또는 함수 호출)에 안전하게 영향을 줄 수 있습니다. 예:

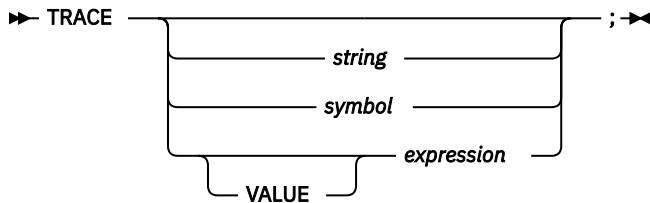
```
fred='PETE'
call multiway fred, 7
....
exit
Multiway: procedure
  arg label . /* One word, uppercase */
  signal value label /* Can add checks for valid labels here */
  /* Transfer control to wherever */
  ....
  Pete: say arg(1) '!' arg(2) /* Displays: "PETE ! 7" */
  return
```

## TRACE

TRACE는 REXX 프로그램의 처리 중 추적 조치를 제어합니다(즉, 사용자에게 표시되는 양).



또는 다음을 수행해야 합니다.



추적은 프로그램에서 일부 또는 전체 설을 설명하며 처리된 것처럼 절의 설명을 생성합니다. TRACE는 주로 디버깅에 사용됩니다. 대화식 디버깅 중 TRACE가 일반적으로 수동으로 입력되기 때문에 해당 구문은 다른 REXX 명령어의 구문보다 간결합니다. (프로그램 실행 중 사용자가 언어 프로세서로 상호 작용할 수 있는 추적의 양식입니다.) 이 사용의 경우, 키 입력을 아끼는 것이 특히 편리합니다.

지정되면, *number*는 정수여야 합니다.

*string* 또는 *expression*은 다음으로 평가됩니다.

- 숫자 옵션
- 나중에 설명된 올바른 접두부나 영문자(단어) 옵션 중 하나
- 널.

*symbol*은 상수로 받아들여지므로 다음과 같습니다.

- 숫자 옵션
- 나중에 설명된 올바른 접두부나 영문자(단어) 옵션 중 하나

*expression*을 평가하는 결과나 TRACE 뒤에 오는 옵션은 추적 조치를 판별합니다. *expression*이 기호나 리터럴 문자열로 시작되지 않는 경우(즉, 연산자나 소괄호와 같이 특수 문자로 시작된 경우) 서브 키워드 VALUE를 생략할 수 있습니다.

### 영문자(단어) 옵션

단어를 전부 입력할 수 있더라도 대문자로 쓰여지고 강조표시된 문자만이 필요합니다. 뒤에 오는 모든 문자가 무시됩니다. 영문자 옵션으로 참조되는 이유입니다. ERROR 및 FAILURE의 정의는 [145 페이지의 『외부 환경에 대한 명령』](#)의 내용을 참조하십시오.

TRACE 조치는 다음과 같이 영문자 옵션에 해당됩니다.

## All

실행 전에 모든 절을 추적합니다(즉, 표시).

## Commands

실행 전에 모든 명령을 추적합니다. 명령의 결과가 오류나 실패인 경우, 추적은 명령으로부터 리턴 코드도 표시합니다.

## Error

명령으로부터의 리턴 코드와 함께 실행 후 결과가 오류나 실패인 명령을 추적합니다.

## Failure

명령으로부터의 리턴 코드와 함께 실행 후 결과가 실패인 명령을 추적합니다. Normal 옵션과 같습니다.

## Intermediates

실행 전에 모든 절을 추적합니다. 표현식 및 대체된 이름 평가 동안 중간 결과도 추적합니다.

## Labels

실행 중 전달된 레이블만 추적합니다. 언어 프로세서가 각 레이블 후 일시정지되면 디버그 모드에서 특히 유용합니다. 사용자가 SIGNAL 명령어 때문에 모든 내부 서브루틴 호출과 제어의 전송을 기록하는 데도 도움이 됩니다.

## 일반

명령으로부터의 리턴 코드와 함께 실행 후 결과가 음수 리턴 코드인 명령을 추적합니다. 이는 기본 설정입니다.

## 끄기

아무 것도 추적하지 않고 특수 접두부 옵션(나중에 설명된)을 OFF로 재설정합니다.

## Results

실행 전에 모든 절을 추적합니다. 표현식을 평가하는(이전의 Intermediates과 대조하여) 마지막 결과를 표시합니다. PULL, ARG 및 PARSE 명령어 중 지정된 값도 표시합니다. 이 설정은 일반 디버깅에 대하여 권장됩니다.

## Scan

처리되지 않고 데이터에 나머지 모든 절을 추적합니다. 기본 검사(누락된 END 등)가 실행되고 추적이 평소처럼 형식 지정됩니다. TRACES 절 자체가 다른 명령문(INTERPRET 또는 대화식 디버그 포함)이나 내부 루틴에서 중첩되지 않은 경우에만 유효합니다.

## 접두부 옵션

접두부 ! 및 ?는 영문자 옵션 중 하나로 또는 단독으로 유효합니다. TRACE 명령어에서 임의의 순서로 두 접두부를 지정할 수 있습니다. 두 번 이상 접두부를 지정할 수 있습니다. 명령어에서 접두부의 각 항목은 이전 접두부의 조치를 되돌립니다. 접두부는 바로 옵션(중간 공백 없이)에 선행해야 합니다.

접두부 ! 및 ?는 다음과 같이 추적과 실행을 수정하십시오.

### ?

대화식 디버그를 제어합니다. 일반적으로 실행 중 ?의 접두부가 있는 TRACE 옵션은 대화식 디버그를 켭니다. 447 페이지의 『프로그램 대화식 디버깅』의 내용을 참조하십시오. 대화식 디버그가 켜져 있는 동안 추적되는 대부분의 절 다음에 해석이 일시정지됩니다. 예를 들면 명령어 TRACE ?E를 사용하면 언어 프로세서가 오류를 리턴하는(즉, 0(영)이 아닌 리턴 코드) 명령을 실행 후 입력을 위해 일시정지합니다.

추적되는 프로그램에 관한 TRACE 명령어가 무시됩니다. (예상치 않게 대화식 디버그에서 종료되지 않도록 합니다.)

여러 방법으로 대화식 디버그를 끌 수 있습니다.

- TRACE 0을 입력하면 모든 추적을 끕니다.
- TRACE를 옵션 없이 입력하면 기본값을 복원합니다. 대화식 디버그를 끄지만 적용 중인 TRACE Normal로 계속 추적합니다(실행 후 실패된 명령 추적)
- TRACE ?를 입력하면 대화식 디버그를 끄고 현재 옵션으로 계속 추적합니다.
- 옵션 앞 ? 접두부로 TRACE 명령어를 입력하면 대화식 디버그를 끄고 새 옵션으로 계속 추적합니다.

? 접두부를 사용하면 대화식 디버그 안 또는 밖으로 교대로 전환합니다. (언어 프로세서가 대화식 디버그 후 사용자 프로그램에서 추가 TRACE 명령문을 무시하기 때문에 CALL TRACE '?'를 사용하여 대화식 디버그를 끄십시오.)

!

호스트 명령 실행을 금합니다. 정규 실행 중 ! 접두가 있는 TRACE 명령어는 모든 후속 호스트 명령의 실행을 일시중단합니다. 예를 들면, TRACE !C를 사용하면 명령을 추적하지만 처리하지 않습니다. 각 명령이 무시되기 때문에 REXX 특수 변수 RC는 0으로 설정됩니다. 잠재적으로 파괴적인 프로그램을 디버깅하기 위해 이 조치를 사용할 수 있습니다. 대화식 디버그에 있는 동안 수동으로 입력된 명령을 금지하지 않습니다. 항상 처리됩니다.

접두부 !가 있는 TRACE 명령어를 실행하여 적용될 때 명령 금지를 끌 수 있습니다. ! 접두부의 반복된 사용은 명령 금지 모드 안 또는 밖으로 교대로 전환합니다. 또는 TRACE 0 또는 옵션이 없는 TRACE를 실행하여 언제든지 명령 금지를 끌 수 있습니다.

## 숫자 옵션

대화식 디버그가 활성화된 경우 및 지정된 옵션이 양의 정수(또는 양의 정수로 평가하는 표현식)인 경우, 해당 숫자는 건너뛰는 디버그 일시정지의 횟수를 표시합니다. (447 페이지의 『프로그램 대화식 디버깅』 참조). 그러나 옵션이 음의 정수(또는 음의 정수로 평가하는 표현식)인 경우, 디버그 일시정지를 포함한 모든 추적이 지정된 절의 개수로 임시로 금지됩니다. 예를 들면, TRACE -100은 일반적으로 추적될 다음 100 절이 사실 표시되지 않음을 의미합니다. 그 후, 이전과 같이 추적이 재개됩니다.

## 추적 팁

1. 루프가 추적 중이면, DO 절 자체는 루프의 모든 반복에 추적됩니다.
2. TRACE 기본 제공 함수를 사용하여 현재 적용 중인 추적 조치를 검색할 수 있습니다(203 페이지의 『TRACE』 참조).
3. 실행 시 사용 가능한 경우, TRACE A, R, I 또는 S를 지정한 경우 null 절에서 주석인 것처럼 추적된 결과 연관된 주석이 추적에 포함됩니다.
4. 실행 전 추적된 명령에 항상 명령의 최종 값이 있으며(즉, 환경에 전달된 문자열) 이를 생성하는 절이 추적된 출력에 제공됩니다.
5. 추적 조치는 서브루틴과 함수 호출에 걸쳐서 자동으로 저장됩니다. 153 페이지의 『CALL』의 내용을 참조하십시오.

## 예

사용할 대부분의 명령 추적 중 하나는 다음과 같습니다.

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/* and tracing Results of expressions begins.      */
```

## TRACE 출력의 형식

중첩의 해당 논리 깊이 등에 따라 자동 형식이 지정(들여쓰기)된 추적된 모든 절이 표시됩니다. 언어 프로세서는 데이터의 인코딩(예를 들어 '40'x 미만인 EBCDIC 값)에서 제어 코드를 (?)로 대체하여 콘솔 간섭을 피합니다. 선행과 후행 공백이 명확하도록 결과(요청된 경우)는 추가 두 공백을 들여쓰기하고 큰따옴표로 묶습니다.

행 번호는 행에 추적된 첫 번째 절에 선행합니다. 행 번호가 99999보다 더 크면 언어 프로세서는 왼쪽에서 자르고 ? 접두부는 잘림을 표시합니다. 예를 들어 행 번호 100354는 ?00354로 표시됩니다. 추적 중 표시된 모든 행에는 추적된 데이터 유형을 식별하기 위해 3자 접두부가 있습니다. 다음과 같을 수 있습니다.

\*\*-\*

프로그램에서 실제로 단일 절 즉, 데이터의 소스를 식별합니다.

+++

추적 메시지를 식별합니다. 대화식 디버그가 시작된 경우 프롬프트 메시지, 명령에서 0이 아닌 리턴 코드일 수 있으며, 대화식 디버그에 있을 때 구문 오류를 표시하거나 프로그램에서 구문 오류 후 절을 역추적할 수 있습니다.

>>>

표현식의 결과(TRACE R에 대해) 또는 구문 분석 중 변수에 지정된 값 또는 서브루틴 호출에서 리턴된 값을 식별합니다.

>.>

구문 분석 중 플레이스홀더에 "지정된" 값을 식별합니다(212 페이지의 『플레이스홀더로 마침표 사용』 참조).

TRACE Intermediates가 적용된 경우에만 다음 접두부가 사용됩니다.

>C>

대체된 변수의 값이 이름에 있는 경우, 추적된 데이터는 사용 전 및 대체 후 추적된 복합 변수의 이름입니다.

>F>

추적된 데이터는 함수 호출의 결과입니다.

>L>

추적된 데이터는 리터럴(문자열, 초기화되지 않은 변수 또는 상수 기호)입니다.

>O>

추적된 데이터는 두 항에 대한 오퍼레이션의 결과입니다.

>P>

추적된 데이터는 접두부 조작의 결과입니다.

>V>

추적된 데이터는 변수의 콘텐츠입니다.

옵션이 TRACE 명령문에 지정되지 않거나 표현식을 평가하는 결과가 null이면, 기본 추적 조치는 복원됩니다. 기본값은 TRACE N, 명령 금지 (!) 끄 및 대화식 디버그 (?) 끄기입니다.

SIGNAL ON SYNTAX가 트래핑하지 않는 구문 오류에 따라 오류에서 절이 항상 추적됩니다. CALL 또는 INTERPRET 또는 활성 중인 함수 호출도 항상 추적됩니다. 찾을 수 없는 레이블에 대한 제어를 전송하려는 시도로 오류가 발생한 경우, 해당 레이블도 추적됩니다. 특수 추적 접두부 +++는 이 역 추적 행을 식별합니다.

## UPPER

UPPER는 한 개 이상 변수의 콘텐츠를 대문자로 변환합니다. 이 변수는 왼쪽에서 오른쪽으로 순서대로 변환됩니다.

이는 REXX/CICS에서 제공되는 비SAA 명령어입니다.

➡ UPPER  variable ; ➡

variable은 하나 이상의 공백 또는 주석으로 다른 variable와 분리되는 기호입니다. 단순 기호와 복합 기호만 지정하십시오. 143 페이지의 『단순 기호』의 내용을 참조하십시오.

이 명령어를 사용하면 TRANSLATE 기본 제공 함수를 반복적으로 호출하는 것보다 편리합니다.

### 예

```
a1='Hello'; b1='there'
Upper a1 b1
say a1 b1      /* Displays "HELLO THERE" */
```

상수 기호 또는 스템이 발생하면 오류 신호가 발생합니다. 초기화되지 않은 변수를 사용하는 것은 오류가 아니고 NOVALUE 조건(SIGNAL ON NOVALUE)가 사용으로 설정된 경우 트랩되는 경우를 제외하고는 영향이 없습니다.





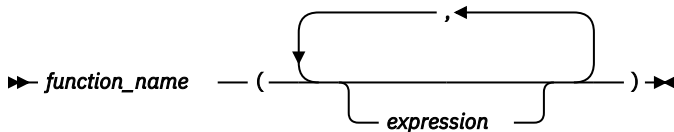
## 제 19 장 함수

함수는 단일 결과 문자열을 리턴하는 내부, 내장 또는 외부 루틴입니다. (서브루틴은 결과를 리턴하거나 리턴하지 않는 내부, 내장 또는 외부 루틴이며 CALL 명령어로 호출되는 함수입니다.)

### 구문

함수 호출은 일부 프로시저를 수행하고 문자열을 리턴하는 루틴을 호출하는 표현식의 한 용어입니다. 이 문자열은 표현식의 계속 평가에서 함수 호출을 대체합니다.

표기법을 사용하여 문자열과 같은 데이터 용어가 올바른 표현식 어느 위치에서나 내부 및 외부 루틴에 함수 호출을 포함시킬 수 있습니다.



`function_name`은 리터럴 문자열이거나 상수로 간주되는 단일 기호입니다.

표현식의 구현 정의된 최대 수가 있을 수 있으며 심표로 구분되고 소괄호 안에 표시됩니다. REXX/CICS에서 구현 최대는 최대 20개의 표현식입니다. 이러한 표현식을 함수에 대한 인수라고 합니다. 각 인수 표현식은 추가 함수 호출을 포함할 수 있습니다.

왼쪽 소괄호는 함수의 이름에 인접해야 하며 그 사이에는 공백이 없어야 합니다. 그렇지 않으면 이 구조는 함수 호출로 인식되지 않습니다. (공백 연산자가 대신 이 위치에서 가정됩니다.) 이름과 왼쪽 소괄호 사이에는 주석만 (효과가 없음) 표시될 수 있습니다.

인수는 왼쪽에서 오른쪽으로 차례대로 평가되고 결과로 발생하는 문자열은 모두 함수에 전달됩니다. 그러면 이 함수는 일부 오퍼레이션을 실행하고(인수가 필수가 아니더라도 보통은 전달된 인수 문자열에 따라) 결국 단일 문자 문자열을 리턴합니다. 그런 다음 이 문자열은 전체 함수 참조가 값이 리턴된 해당 데이터인 변수의 이름으로 대체된 것처럼 원래 표현식에 포함됩니다.

예를 들어 함수 SUBSTR은 언어 프로세스에 기본 제공되고(200 페이지의 『SUBSTR(하위 문자열)』 절 참조) 다음과 같이 사용될 수 있습니다.

```
N1='abcdefghijk'  
Z1='Part of N1 is: 'substr(N1,2,7)  
/* Sets Z1 to 'Part of N1 is: bcdefgh' */
```

함수는 인수의 개수 변수를 가질 수 있습니다. 필요한 것만 지정해야 합니다. 예를 들어, SUBSTR('ABCDEF',4)는 DEF를 리턴합니다.

### 함수와 서브루틴

함수 호출 메커니즘은 서브루틴에 대한 호출 메커니즘과 동일합니다. 함수와 서브루틴 사이의 유일한 차이점은 함수는 데이터를 리턴해야 하지만 서브루틴은 리턴할 필요가 없다는 점입니다.

다음 유형의 루틴은 함수처럼 호출될 수 있습니다.

#### 내부

루틴 이름이 프로그램에서 레이블로 존재하는 경우, 현재 처리 프로세스 상태가 저장되므로 나중에 호출 위치로 돌아가서 실행을 재개할 수 있습니다. 그런 다음 제어가 이름과 일치하는 프로그램에서 첫 번째 레이블에 전달됩니다. CALL 명령어에서 호출된 루틴에서와 같이 다양한 다른 상태 정보(TRACE 및 NUMERIC 설정과 같이)도 저장됩니다. 세부사항은 CALL 명령어(153 페이지의 『CALL』)를 참조하십시오. SIGNAL 및 CALL을 함께 사용하여 실행 시 이름이 판별되는 내부 루틴을 호출할 수 있습니다. 다중 방향 호출이라고 합니다(172 페이지의 『SIGNAL』 참조).

내부 루틴을 함수로 호출하는 경우, 내부 루틴에서 리턴하도록 RETURN 명령어에서 표현식을 지정해야 합니다. 서브루틴으로 호출되는 경우는 지정하지 않아도 됩니다. 예:

```
/* 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
```

내부 레이블을 검색하는 동안 구문 검사가 수행되며 exec가 토큰화됩니다. 자세한 내용은 [성능 고려사항](#)의 내용을 참조하십시오. 그 자체를 호출한다는 점에서 FACTORIAL은 비정상적입니다(이것이 재귀 호출입니다). PROCEDURE 명령어는 새 변수 n이 각 호출에 대해 작성되는지 확인합니다.

**참고:** 루틴에 대한 검색이 있는 경우, 언어 프로세서는 현재 REXX 프로그램에서 해당 명령어를 스캔하여 내부 레이블을 찾습니다. 검색 중 언어 프로세서는 구문 오류가 발생할 수 있습니다. 결과로서 구문 오류는 처리되는 원래 행과 다른 명령문에서 발생할 수 있습니다.

## 기본 제공

이 함수가 항상 사용 가능하며 [181 페이지의 『내장 함수』](#)에서 정의됩니다.

## 외부

프로그램 및 언어 프로세서에 외부적인 함수를 쓰거나 사용할 수 있습니다. 외부 루틴은 REXX에 작성되어야 합니다. REXX 프로그램을 함수로 호출할 수 있으며, 이 경우 둘 이상의 인수 문자열을 전달합니다. ARG 또는 PARSE ARG 명령어나 ARG 기본 제공 함수는 이러한 인수 문자열을 검색할 수 있습니다. 함수로 호출된 경우 프로그램은 데이터를 호출자로 리턴해야 합니다.

## 참고:

1. 외부 REXX 함수는 CICS 제공 언어로 작성된 프로그램에 대한 EXEC CICS LINK를 쉽게 수행할 수 있습니다. 또한, REXX/CICS 명령 루틴은 어셈블러로 작성될 수 있습니다.
2. 외부 REXX 프로그램을 함수로 호출하는 것은 내부 루틴을 호출하는 것과 유사합니다. 그러나 호출자의 변수가 항상 숨겨져 있고 내부 값의 상태(NUMERIC 설정과 같은)는 기본값(호출자의 값 상속 대신)으로 시작한다는 점에서 외부 루틴은 내재적 PROCEDURE입니다.
3. 다른 REXX 프로그램은 함수로 호출될 수 있습니다. 호출된 REXX 프로그램을 중단하기 위해 EXIT 또는 RETURN을 사용할 수 있고 각 경우에 표현식을 지정해야 합니다.
4. 주의하여 변수 함수 이름을 가진 함수를 처리하도록 INTERPRET 명령어를 사용할 수 있습니다. 그러나 프로그램의 명료성을 감소시키기 때문에 가능한 경우 이를 피하십시오.

## 검색 순서

함수에 대한 검색 순서는 내부 루틴, 기본 제공 함수, 외부 함수 순입니다.

함수 이름이 리터럴 문자열(즉, 따옴표에 지정된)로 지정되면 내부 루틴이 사용되지 않습니다. 이 경우 함수는 기본 제공 또는 외부여야 합니다. 기본 제공 함수의 이름을 사용하여 해당 기능을 확장하지만 필요한 경우 기본 제공 함수를 호출할 수 있습니다. 예:

```
/* 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)
```

기본 제공 함수는 대문자로 된 이름을 가지고 있으므로 예에서처럼 검색이 계속되려면 리터럴 문자열의 이름은 대문자여야 합니다. 외부 함수에 대해서도 동일하게 적용됩니다.

외부 함수와 서브루틴은 시스템 정의된 검색 순서가 있습니다.

REXX로 작성된 exec, 명령, 외부 함수 또는 서브루틴이 REXX/CICS(예: CICS 명령행, CALL 명령어, EXEC 명령 또는 exec에서 명령)에서 호출될 때마다 대상 exec를 찾기 위한 검색 순서는 다음과 같습니다.

- DEFCMD 정의 명령 exec 또는 다른 호출된 exec에 대한 검색 순서.

권한 부여된 명령을 실행하려는 시도인 경우, 이는 권한 부여된 사용자이거나 명령이 ddname CICEXEC 또는 CICAUTH에서 로드된 exec에 있는지 여부를 확인합니다. 이들 중 true가 없는 경우, 명령은 -4의 리턴 코드로 실패합니다.

1. 내부 함수나 서브루틴에 대한 현재 exec에서.

이 검색은 해당 이름을 따옴표에 넣음으로써 명시적으로 함수(또는 서브루틴)를 식별하여 이 검색을 무시할 수 있습니다.

2. 스토리지에서 Exec(이전 EXECLOAD에서).

3. 현재 RFS 디렉토리.

4. 현재 PATH. RFS 디렉토리 및 MVS 파티션된 데이터 세트는 실행 시 최신 PATH 명령으로 나열된 순서대로 검색됩니다. 375 페이지의 『PATH』의 내용을 참조하십시오.

5. ddname CICAUTH에 할당된(또는 연결된) PDS 데이터 세트.

사용자가 권한 부여된 사용자이고 현재 exec가 CICEXEC에서 로드되었으며 검색이 권한 부여된 명령의 프로그램에 대한 것인 경우 CICAUTH ddname을 검사하십시오.

6. ddname CICEXEC에 할당된(또는 연결된) PDS 데이터 세트.

7. ddname CICUSER에 할당된(또는 연결된) PDS 데이터 세트.

DEFECMD 명령에 관한 자세한 정보는 355 페이지의 『DEFECMD』의 내용을 참조하십시오.

**참고:**

1. REXX/CICS execs가 MVS Partitioned Organization(DSORG=PO)에서 호출하는 유일한 MVS 데이터세트 유형. MVS 순차 데이터 세트(DSORG=PS)가 지원되지 않습니다.
2. MVS 파티션된 데이터 세트에 거주하는 REXX exec는 73 ~ 80열에 순서 번호가 없을 수 있습니다.
3. 데이터세트가 도달되기 전에 대상 exec를 찾지 못하는 경우(왼쪽에서 오른쪽으로 PATH 목록 처리 시) 마지막 PATH 명령문에 지정된 MVS 파티션된 데이터 세트를 동적으로 할당하려고 합니다(SVC 99로).
4. DEFECMD AUTH 매개변수는 REXX/CICS 명령이 권한 부여된 명령임을 지정하는데 사용됩니다.
5. EXECLOAD 및 EXECDROP 명령은 권한 부여된 명령이며 REXX/CICS 권한 부여된 사용자 또는 CICEXEC에서 로드된 exec만이 이를 실행할 수 있게 합니다.

**실행 중 오류**

외부 또는 기본 제공 함수가 오류를 감지하는 경우 언어 프로세서에게 알리며 구문 오류가 발생합니다. 그러므로 함수 호출을 포함한 절의 실행이 종료됩니다. 마찬가지로 외부 함수가 올바르게 데이터를 리턴하는 데 실패하면 언어 프로세서는 이를 감지하고 오류로 보고합니다.

내부 함수의 실행 중 구문 오류가 발생하면 트래핑될 수 있으며 그런 다음 복구가 가능합니다. 오류가 트래핑되지 않은 경우 프로그램이 종료됩니다.

## 내장 함수

REXX는 문자 처리, 변환 및 정보 함수를 포함하여 풍부한 기본 제공 함수 세트를 제공합니다.

다른 내장 및 외부 함수가 일반적으로 사용 가능합니다. 208 페이지의 『REXX/CICS에서 제공되는 외부 기능』의 내용을 참조하십시오.

기본 제공 함수에 관한 일반 참고사항은 다음과 같습니다.

- 인수도 필요하지 않는 경우에도 함수에서 소괄호는 항상 필수입니다. 첫 번째 소괄호는 중간에 공백 없이 함수의 이름이 다음에 와야 합니다.
- 내장 함수는 내부적으로 NUMERIC DIGITS 9와 함께 동작하며 NUMERIC FUZZ 0은 언급된 곳을 제외하고 NUMERIC 설정에 대한 변경으로 영향을 받지 않습니다. 필요한 경우 NUMERIC DIGITS의 현재 설정에 따라 *number*로서 이름 지정된 인수가 반올림되며(수가 0에 추가된 것처럼) 사용 전 유효성을 검사합니다. DATATYPE의 특정 옵션의 경우 ABS, FORMAT, MAX, MIN, SIGN 및 TRUNC 함수에서 발행합니다. RANDOM에 대해 true가 아닙니다.
- *string*으로서 이름 지정된 인수는 null 문자열일 수 있습니다.

- 인수가 *length*를 지정하는 경우 모든 양의 정수이거나 0(영)이어야 합니다. 문자열에서 시작 문자나 단어를 지정하는 경우, 다르게 언급되지 않는 한 모든 양의 정수여야 합니다.
- 마지막 인수가 선택사항인 경우, 항상 쉼표를 포함하여 생략했음을 표시합니다. 예를 들어, `DATATYPE(1)`과 같이 `DATATYPE(1,)`은 `NUM`을 리턴합니다. 후행 쉼표의 수를 포함할 수 있습니다. 이는 무시됩니다. (실제 매개변수가 있는 곳에서, 기본값이 적용됩니다.)
- *pad* 문자를 지정하는 경우 정확하게 문자 하나의 길이어야 합니다. (채움 문자는 보통 오른쪽에서 문자열을 확장합니다. 예를 들어, `LEFT` 기본 제공 함수 [195 페이지](#)의 『`LEFT`』의 내용을 참조하십시오.)
- 함수에 *option*이 있는 경우 문자열의 첫 번째 문자를 지정하여 선택할 수 있습니다. 해당 문자는 대문자나 소문자일 수 있습니다.
- 일부 내장 함수는 `DBCS`를 지원합니다. 이 함수의 전체 목록과 설명은 [431 페이지](#)의 『제 33 장 2바이트 문자 세트(`DBCS`) 지원』의 내용을 참조하십시오.

## ABBREV(약어)

*info*이 *information*의 선행 문자와 같으며 **그리고** *info*의 길이가 *length* 미만이지 않은 경우 ABBREV 함수는 1을 리턴합니다. 이들 조건 중 하나가 충족되지 않은 경우 ABBREV는 0입니다.

**► ABBREV( — information — , — info — ) ◄**

*length*를 지정하는 경우 모든 양의 정수이거나 0(영)이어야 합니다. *length*의 기본값은 *info*의 문자 수입니다.

## 예제

```

ABBRV('Print','Pri')      -> 1
ABBRV('PRINT','Pri')      -> 0
ABBRV('PRINT','PRI',4)    -> 0
ABBRV('PRINT','PRY')      -> 0
ABBRV('PRINT','')         -> 1
ABBRV('PRINT','',1)       -> 0

```

**참고:** 0(또는 기본값)의 길이가 사용된 경우 널 문자열이 항상 일치됩니다. 이렇게 하면 기본 키워드가 자동으로 선택될 수 있게 합니다. 예:

```
say 'Enter option:'; pull option .
select /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  ...
otherwise nop;
end;
```

## ABS(절대값)

ABS 함수는 *number*의 절대값을 리턴합니다. 결과에 부호가 없으며, 현재 NUMERIC 설정에 따라 형식이 지정됩니다.

➡ **ABS( — *number* — )** ➡

## 예제

```
ABS('12.3')      -> 12.3
ABS(' -0.307')   -> 0.307
```

## ADDRESS

ADDRESS 함수는 명령이 현재 제출되는 환경의 이름을 리턴합니다. 환경은 하위 명령 환경의 이름일 수 있습니다.

➡ ADDRESS( — ) ➡

자세한 정보는 ADDRESS 명령어(151 페이지의 『ADDRESS』)를 참조하십시오. 후행 공백은 결과에서 제거됩니다.

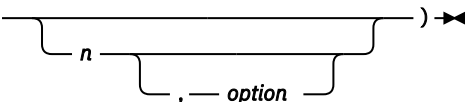
### 예제

```
ADDRESS() -> 'CICS'      /* default under CICS */
ADDRESS() -> 'EDITSVR'   /* default under CICS editor */
```

## ARG(인수)

ARG 함수는 인수 문자열이나 인수 문자열에 관한 정보를 프로그램이나 내부 루틴으로 리턴합니다.

➡ ARG( — ) ➡



*n*을 지정하지 않은 경우, 프로그램이나 내부 루틴에 전달된 인수의 개수가 리턴됩니다.

*n*만 지정하는 경우 *n*번째 인수 문자열이 리턴됩니다. 인수 문자열이 없는 경우 null 문자열이 리턴됩니다. *n*은 모든 양의 정수여야 합니다.

*option*을 지정하는 경우, ARG는 *n*번째 인수 문자열의 존재를 테스트합니다. 다음은 올바른 옵션입니다.(대문자로 처리되고 강조표시된 문자만이 필요합니다. 다음에 오는 모든 문자가 무시됩니다.)

### Exists

*n*번째 인수가 존재하는 경우 1을 리턴합니다. 즉 루틴이 호출되면 명시적으로 지정된 경우입니다. 그렇지 않은 경우 0을 리턴합니다.

### Omitted

*n*번째 인수가 생략된 경우 1을 리턴합니다. 즉 루틴이 호출되면 명시적으로 지정되지 않은 경우입니다. 그렇지 않은 경우 0을 리턴합니다.

### 예제

```
/* following "Call name;" (no arguments) */
ARG()      -> 0
ARG(1)     -> ''
ARG(2)     -> ''
ARG(1,'e') -> 0
ARG(1,'O') -> 1

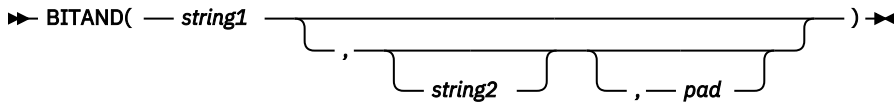
/* following "Call name 'a','b';" */
ARG()      -> 3
ARG(1)     -> 'a'
ARG(2)     -> ''
ARG(3)     -> 'b'
ARG(n)     -> '' /* for n>=4 */
ARG(1,'e') -> 1
ARG(2,'E') -> 0
ARG(2,'O') -> 1
ARG(3,'o') -> 0
ARG(4,'o') -> 1
```

### 참고:

1. 명시적 인수 문자열이 없는 경우, 인수 문자열의 수는 ARG(*n*, 'e')가 1 또는 0을 리턴하는 최대값 *n*입니다. 즉, 마지막 명시적으로 지정된 인수 문자열의 위치입니다.
2. 명령이라는 프로그램은 0 또는 1 인수 문자열만 가질 수 있습니다. 이름으로만 호출된 경우 프로그램에는 0개의 인수 문자열이 있으며 공백을 포함한 다른 것이 명령과 함께 포함되는 경우 1개의 인수 문자열이 있습니다.
3. ARG 또는 PARSE ARG 명령어로 프로그램이나 내부 루틴으로 인수 문자열을 검색하고 직접 구문 분석할 수 있습니다.(152 페이지의 『ARG』, 166 페이지의 『PARSE』 및 211 페이지의 『제 20 장 구문 분석』 참조).

## BITAND (Bit by Bit AND)

BITAND 함수는 두 개의 입력 문자열이 논리적으로 함께 AND된 비트 단위로 구성된 문자열을 리턴합니다.



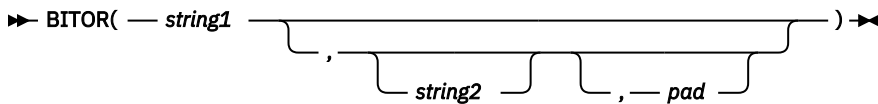
문자열의 인코딩은 논리적 조작으로 사용됩니다. 결과의 길이는 두 문자열에서 더 긴 길이입니다. *pad* 문자가 제공되지 않은 경우, 두 문자열 중에서 더 짧은 문자열이 사용된 경우 AND 연산이 중지되며 더 긴 문자열에서 처리되지 않은 부분은 부분 결과에 추가됩니다. *pad*가 제공된 경우, 논리적 연산을 수행하기 전에 오른쪽의 두 문자열에서 더 짧은 문자열을 확장합니다. *string2*의 기본값은 영 길이(null) 문자열입니다.

### 예제

```
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 (Bit by Bit OR)

BITOR 함수는 두 개의 입력 문자열이 논리적으로 함께 inclusive-OR된 비트 단위로 구성된 문자열을 리턴합니다.



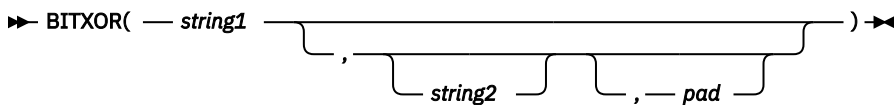
문자열의 인코딩은 논리적 조작으로 사용됩니다. 결과의 길이는 두 문자열에서 더 긴 길이입니다. *pad* 문자가 제공되지 않으면, OR 조작은 두 개의 문자열이 부족한 것 지쳐버리는 때를 중지하고 더 긴 문자열의 미처리된 부분이 부분적인 결과에 추가됩니다. *pad*가 제공된 경우, 논리적 연산을 수행하기 전에 오른쪽의 두 문자열에서 더 짧은 문자열을 확장합니다. *string2*의 기본값은 영 길이(null) 문자열입니다.

### 예제

```
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(Bit by Bit Exclusive OR)

BITXOR 함수는 두 입력 문자열이 논리적으로 함께 eXclusive-OR된 비트 단위로 구성된 문자열을 리턴합니다.



문자열의 인코딩은 논리적 조작으로 사용됩니다. 결과의 길이는 두 문자열에서 더 긴 길이입니다. *pad* 문자가 제공되지 않은 경우, 두 문자열 중에서 더 짧은 문자열이 사용된 경우 XOR 연산이 중지되며 더 긴 문자열에서 처리되지 않은 부분은 부분 결과에 추가됩니다. *pad*가 제공된 경우, 논리적 연산을 수행하기 전에 오른쪽의 두 문자열에서 더 짧은 문자열을 확장합니다. *string2*의 기본값은 영 길이(null) 문자열입니다.

### 예제

```
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, '40'x)       -> '5C5C'x
BITXOR('C711'x, '222222'x, ' ') -> 'E53362'x /* EBCDIC */

```

## B2X(2진을 16진으로)

B2X 함수는 16진으로 변환된 *binary\_string*을 나타내는 문자 포맷으로 문자열을 리턴합니다.

➡ B2X( — *binary\_string* — ) ➡

*binary\_string*은 2진(0 또는 1) 숫자의 문자열입니다. 어떤 길이라도 가능합니다. 가독성을 지원하기 위해 선택적으로 *binary\_string*(선행이나 후행이 아닌 4자리 경계로만)에 공백을 포함할 수 있습니다. 이는 무시됩니다.

리턴된 문자열은 값 A-F에 대해 대문자 알파벳 문자를 사용하며 공백을 포함하지 않습니다.

*binary\_string*이 null 문자열이면, B2X는 null 문자열을 리턴합니다. *binary\_string*에서 2진 숫자의 수가 4의 배수가 아닌 경우, 최대 세 개의 0 숫자가 4의 배수인 총계를 만들기 위해 변환 전에 왼쪽에 추가됩니다.

### 예제

```

B2X('11000011') -> 'C3'
B2X('10111')    -> '17'
B2X('101')       -> '5'
B2X('1 1111 0000') -> '1F0'

```

2진수를 다른 양식으로 변환하기 위해 B2X를 함수 X2D 및 X2C와 결합할 수 있습니다. 예를 들어, 다음과 같습니다.

```

X2D(B2X('10111')) -> '23' /* decimal 23 */

```

## CENTER/CENTRE

*string*을 중앙에 두고 길이를 구성하는 데 필요한 만큼 *pad* 문자를 추가하여 길이의 문자열 *length*을 리턴합니다.

➡ CENTER( — *string* — , — *length* — ) ➡  
 CENTRE( — *string* — , — *length* — , — *pad* — ) ➡

*length*는 모든 양의 정수이거나 0(영)이어야 합니다. 기본 *pad* 문자는 공백입니다. 문자열이 *length*보다 더 길면, 양쪽 끝이 잘립니다. 홀수의 문자가 절단되거나 추가되면, 오른쪽 끝은 왼쪽 끝보다 하나가 많은 문자를 잃거나 얻습니다.

### 예제

```

CENTER(abc, 7)      -> '  abc  '
CENTER(abc, 8, '-') -> '---abc---'
CENTRE('The blue sky', 8) -> 'e blue s'
CENTRE('The blue sky', 7) -> 'e blue '

```

**참고:** 영국과 미국의 철자 차이로 인한 오류를 피하기 위해, 이 함수를 CENTRE 또는 CENTER라고 합니다.

## COMPARE

문자열 *string1* 및 *string2*가 동일하면 COMPARE 함수는 0을 리턴합니다. 그렇지 않으면, 일치하지 않는 첫 번째 문자의 위치를 리턴합니다.

➡ COMPARE( — *string1* — , — *string2* — ) ➡  
 , — *pad* —

더 짧은 문자열은 필요한 경우 오른쪽에 *pad*로 채웁니다. 기본 *pad* 문자는 공백입니다.

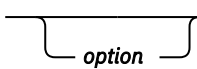


## 예제

```
COMPARE('abc','abc')      -> 0
COMPARE('abc','ak')       -> 2
COMPARE('ab','ab')        -> 0
COMPARE('ab','ab',' ')    -> 0
COMPARE('ab','ab','x')    -> 3
COMPARE('ab--','ab','-')  -> 5
```

## CONDITION

CONDITION 함수는 현재 트래핑된 조건과 연관된 조건 정보를 리턴합니다.

➡ CONDITION(  ) ➡

조건 트랩의 설명은 233 페이지의 『제 22 장 조건 및 조건 트랩』의 내용을 참조하십시오. 다음 정보를 요청할 수 있습니다.

- 현재 트래핑된 조건의 이름
- 해당 조건과 연관된 설명 문자열
- 조건 트랩(CALL 또는 SIGNAL)의 결과로서 처리된 명령어
- 트래핑된 조건의 상태.

리턴할 정보를 선택하려면 다음 옵션을 사용하십시오.(대문자로 처리되고 강조표시된 문자만이 필요합니다. 다음에 오는 모든 문자가 무시됩니다.)

### 조건 이름

현재 트래핑된 조건의 이름을 리턴합니다.

### 설명

현재 트래핑된 조건과 연관된 설명 문자열을 리턴합니다. 사용 가능한 설명이 없는 경우, null 문자열을 리턴하십시오.

### 명령어

CALL 또는 SIGNAL, 현재 조건이 트래핑되면 처리되는 명령어의 키워드 중 하나를 리턴합니다. *option*을 생략하면 이것이 기본값이 됩니다.

### Status

현재 트래핑된 조건의 상태를 리턴합니다. 처리 중 이를 변경할 수 있으며 다음과 같을 수 있습니다.

ON - 조건이 사용 가능함

OFF - 조건이 사용 불가능함

DELAY - 조건의 새 항목이 지연되거나 무시됩니다.

조건이 트래핑되지 않은 경우 CONDITION 함수는 네 개의 모든 경우에 대해 null 문자열을 리턴합니다.

## 예제

```
CONDITION()      -> 'CALL'      /* perhaps */
CONDITION('C')   -> 'FAILURE'
CONDITION('I')   -> 'CALL'
CONDITION('D')   -> 'FailureTest'
CONDITION('S')   -> 'OFF'       /* perhaps */
```

**참고:** CONDITION 함수는 서브 루틴 호출에서 저장되고 복원된 조건 정보를 리턴합니다(CALL ON 조건 트래핑이 발생하는 정보 포함). 그러므로 CALL ON *trapname*으로 호출된 서브 루틴이 리턴된 후, 현재 트래핑된 조건이 CALL이 발생하기 전에 최신이었던 조건으로 되돌립니다(없음일 수 있음). CONDITION은 조건이 트래핑되기 전에 리턴된 값을 리턴합니다.

## COPIES

COPIES 함수는 *string*의 *n* 연결 사본을 리턴합니다. *n*은 모든 양의 정수 또는 0이어야 합니다.



➡ COPIES( — *string* — , — *n* — ) ➡

## 예제

```
COPIES('abc',3)    -> 'abcabcabc'
COPIES('abc',0)    -> ''
```

## C2D(문자를 10진수로)

C2D 함수는 *string*에 대한 2진 표현의 10진수 값을 리턴합니다.

➡ C2D( — *string* — , — *n* — ) ➡

결과를 정수로 표현될 수 없는 경우 오류가 발생합니다. 즉, 결과는 NUMERIC DIGITS의 현재 설정보다 더 많은 숫자를 가지고 있지 않아야 합니다. *n*을 지정하는 경우 리턴된 결과의 길이입니다. *n*을 지정하지 않은 경우, *string*은 부호 없는 2진 숫자로 처리됩니다.

*string*이 null인 경우, 0을 리턴합니다.

**구현 최대:** 입력 문자열이 최종 결과를 형성함에 있어 중요한 250 이상 문자를 가질 수 없습니다. 선행 기호 문자 ('00'x 및 'FF'x)는 이 총계로 계산되지 않습니다.

## 예제

```
C2D('09'X)    -> 9
C2D('81'X)    -> 129
C2D('FF81'X)   -> 65409
C2D('')       -> 0
C2D('a')      -> 129 /* EBCDIC */
```

*n*을 지정하는 경우 문자열은 *n* 문자로 표시된 부호 있는 숫자로 사용됩니다. 맨 왼쪽 비트가 꺼진 경우 수는 양수이며 2보수 표기법에서 맨 왼쪽 비트가 켜진 경우 음수입니다. 두 경우, 정수로 변환되므로 음수일 수 있습니다. *string*은 '00'x 문자로 왼쪽에서 채워지거나("부호 확장되지" 않음에 유의) 왼쪽을 *n* 문자로 자릅니다. 이 채우기 또는 잘림은 RIGHT(*string*, *n*, '00'x)가 처리된 것과 같습니다. *n*이 0인 경우, C2D는 항상 0을 리턴합니다.

```
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(문자를 16진수로)

C2X 함수는 16진으로 변환된 *string*을 나타내는 문자 포맷으로 문자열을 리턴합니다.

➡ C2X( — *string* — ) ➡

리턴된 문자열은 입력 문자열보다 두 배 많은 바이트를 포함합니다. 예를 들면, EBCDIC 시스템에서 문자 1의 EBCDIC 표시는 'F1'X이기 때문에 C2X(1)은 F1을 리턴합니다.

리턴된 문자열은 값 A-F에 대해 대문자로 된 알파벳을 사용하고 공백을 포함하지 않습니다. *string*은 어떤 길이라도 가능합니다. *string*이 null이면 null 문자열을 리턴합니다.

## 예제

```
C2X('72s')    -> 'F7F2A2' /* 'C6F7C6F2C1F2'X in EBCDIC */
C2X('0123'X)   -> '0123' /* 'F0F1F2F3'X in EBCDIC */
```

## DATATYPE

*string*만 지정한 경우 및 *string*이 오류 없이 0에 추가될 수 있는 유효한 REXX 수인 경우, DATATYPE 함수는 NUM을 리턴합니다. *string*이 유효한 수가 아닌 경우 DATATYPE은 CHAR을 리턴합니다.

➡ DATATYPE( — *string* — , — *type* — ) ➡

*type*을 지정한 경우, *string*이 유형과 일치하면 1을 리턴합니다. 그렇지 않은 경우 0을 리턴합니다. *string*이 null인 경우, 해당 함수는 0을 리턴합니다(*type*이 X인 경우 제외, null 문자열에 대해 1을 리턴). 다음은 유효한 *type*입니다. 대문자만이 필요합니다. 다음에 오는 모든 문자가 무시됩니다. hexadecimal 옵션의 경우 h 대신에 x로 옵션의 이름을 지정하는 사용자의 문자열을 시작해야 합니다.)

### Alphanumeric

*string*이 a-z, A-Z 및 0-9 범위의 문자만 포함하는 경우 1을 리턴합니다.

### Binary

*string*이 문자 0이나 1 또는 둘 다를 포함하는 경우에만 1을 리턴합니다.

### C

*string*이 혼용 SBCS/DBCS 문자열인 경우 1을 리턴합니다.

### Dbcs

*string*이 SO와 SI 바이트에 의해 둘러싸인 DBCS-전용 문자열인 경우 1을 리턴합니다.

### Lowercase

*string*이 a-z 범위에서 문자만을 포함하는 경우 1을 리턴합니다.

### 대소문자 혼용

*string*이 a-z 및 A-Z 범위에서 문자만을 포함하는 경우 1을 리턴합니다.

### 숫자

*string*이 유효한 REXX 수인 경우 1을 리턴합니다.

### 기호

*string*이 REXX 기호에서 유효한 문자만 포함하는 경우 1을 리턴합니다([132 페이지의 『토큰』](#) 참조). 대문자와 소문자 알파벳 모두가 허용된다는 점에 유의하십시오.

### Uppercase

*string*이 A-Z 범위에서 문자만을 포함하는 경우 1을 리턴합니다.

### Whole number

*string*이 현재 NUMERIC DIGITS 설정에서 REXX 정수인 경우 1을 리턴합니다.

### heXadecimal

*string*이 a-f, A-F, 0-9 및 공백을 포함하는 경우(16진 문자 쌍 사이에만 공백이 표시되는 경우에 한해) 1을 리턴합니다. 또한 *string*이 null 문자열인 경우, 1을 리턴하며, 이는 올바른 16진 문자열입니다.

**참고:** DATATYPE 함수는 이들 문자의 인코딩과 상관 없이(예를 들어, ASCII 또는 EBCDIC) 문자열에서 문자의 의미나 유형을 테스트합니다.

## 예제

```
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 함수는 기본적으로 해당 날짜에 대해 선행 0(영)이나 공백이 없는 *dd mon yyyy* (일 월 년, 예를 들어 13 Mar 1992) 형식의 로컬 날짜를 리턴합니다.

➡ DATE( option ) ➡

활성 언어에 월 이름의 축약형이 있는 경우 사용됩니다(예: Jan 또는 Feb).

다음 옵션을 사용하여 특정 형식을 얻을 수 있습니다. 대문자만이 필요합니다. 다음에 오는 모든 문자가 무시됩니다.

### Base

기본 날짜, 1 January 0001 이후 완전한 하루의 수(즉, 당일 포함하지 않음)를 *dddddd* 형식으로(선행 0(영)이나 공백이 없음) 리턴합니다. 표현식 DATE('B')//7은 현재 요일에 해당되는 0-6 범위의 수를 리턴하며, 여기서 0은 월요일이며 6은 일요일입니다.

그러므로, 이 함수는 작동 중인 자국어와 상관없는 요일을 판별할 때 사용될 수 있습니다.

**참고:** 1 January 0001의 기본 날짜는 현재 그레고리력(400으로 나눌 수 없는 세기를 제외하고 4로 나눌 수 있는 매년 남는 날로, 매년 365일)을 뒤로 확장하여 판별됩니다. 원래 그레고리력으로 작성된 달력 시스템에서 오류를 고려하지 않습니다.

### Century

*dddddd* 형식에서(선행 0(영)이 없음) 100의 배수인 마지막 해의 1월 1일 이후로 당일을 포함한 일의 수를 리턴합니다. 예: 1992년 3월 13일 DATE(C)에 대한 호출은 33675를 리턴하며 1900년 1월 1일에서부터 1992년 3월 13일까지의 일 수입니다. 마찬가지로, 2000년 1월 2일에서 DATE(C)에 대한 호출은 2를 리턴하며, 2000년 1월 1일에서부터 2000년 1월 2일까지의 일 수입니다.

### Days

다음 형식으로 올해까지 당일을 포함한 일 수를 포함합니다. *ddd*(선행 0(영)이나 공백이 없음).

### European

*dd/mm/yy* 형식의 날짜를 리턴합니다.

### Julian

*yyddd* 형식으로 날짜를 리턴합니다.

### Month

이번 달의 전체 영어 이름을 리턴합니다(예: August).

### 일반

*dd mon yyyy* 형식으로 날짜를 리턴합니다. 이는 기본입니다.

### Ordered

*yy/mm/dd* 형식으로 날짜를 리턴합니다(정렬 등에 적합).

### 표준

*yyyymmdd* 형식으로 날짜를 리턴합니다(정렬 등에 적합).

### Usa

*mm/dd/yy* 형식으로 날짜를 리턴합니다.

### Weekday

혼용 문자로 요일에 대한 영어 이름을 리턴합니다. 예: Tuesday.

**참고:** 하나의 절에서 DATE 또는 TIME에 대한 첫 번째 호출로 인해 해당 절에서 이들 함수에 대한 모든 호출에 대해 사용된 시간소인이 작성됩니다. 그러므로 단일 표현식이나 절에서 DATE 또는 TIME 함수 또는 둘 다에 대한 다중 호출이 서로 일치한다고 보장합니다.

### 예제

다음 예에서는 오늘이 1992년 3월 13일이라고 가정합니다.

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(Double-Byte Character Set Functions)

DBCS 처리 함수의 일부인 함수가 나열됩니다.

- DBADJUST
- DBBRACKET
- DBCENTER
- DBCJUSTIFY
- DBLEFT
- DBRIGHT
- DBRLEFT
- DBRRIGHT
- DBTODBCS
- DBTOSBCS
- DBUNBRACKET
- DBVALIDATE
- DBWIDTH

431 페이지의 『제 33 장 2바이트 문자 세트(DBCS) 지원』의 내용을 참조하십시오.

## DELSTR(문자열 삭제)

DELSTR 함수는  $n$ 번째 문자에서 시작하고,  $length$  문자인 하위 문자열을 삭제한 후  $string$ 을 리턴합니다.

➡ DELSTR( —  $string$  — , —  $n$  —  $length$  ) ➡

$length$ 을 생략하거나  $length$ 가  $n$ 에서  $string$ 의 마지막까지 문자 수보다 더 크면, 함수는  $string$ 의 나머지를 삭제합니다( $n$ 번째 문자 포함).  $length$ 는 모든 양의 정수이거나 0(영)이어야 합니다.  $n$ 은 모든 양의 정수여야 합니다.  $n$ 이  $string$ 의 길이보다 더 크면, 함수는 변경되지 않은  $string$ 을 리턴합니다.

### 예제

```

DELSTR('abcd',3)    -> 'ab'
DELSTR('abcde',3,2) -> 'abe'
DELSTR('abcde',6)   -> 'abcde'

```

## DELWORD(단어 삭제)

DELWORD 함수는  $n$ 번째 단어에서 시작하고  $length$  공백으로 구분된 단어인 하위 문자열 삭제 후  $string$ 을 리턴합니다.

➡ DELWORD( —  $string$  — , —  $n$  —  $length$  ) ➡

$length$ 를 생략하거나  $length$ 가  $n$ 에서  $string$ 의 마지막까지의 단어 수보다 더 크면, 함수는  $string$ 의 나머지 단어를 삭제합니다( $n$ 번째 단어 포함).  $length$ 는 모든 양의 정수이거나 0(영)이어야 합니다.  $n$ 은 모든 양의 정수여야 합니다.

다.  $n$ 가 *string*에서 단어 수보다 크면, 함수는 변경되지 않은 *string*를 리턴합니다. 삭제된 문자열은 포함된 마지막 단어 다음의 공백을 포함하지만 포함된 첫 번째 단어에 선행하는 공백이 없습니다.

## 예제

```
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 함수는 NUMERIC DIGITS의 현재 설정을 리턴합니다.

➡ DIGITS( — ) ➡

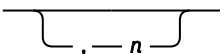
자세한 정보는 NUMERIC 명령어 [164 페이지](#)의 『NUMERIC』의 내용을 참조하십시오.

## 예

```
DIGITS() -> 9 /* by default */
```

## D2C(10진수를 문자로)

D2C 함수는 2진으로 변환된 *wholenumber*, 10진수를 표시하는 문자 형식으로 문자열을 리턴합니다.

➡ D2C( — *wholenumber*  ) ➡

$n$ 을 지정하는 경우, 문자에서 최종 결과의 길이입니다. 변환 후 입력 문자열은 요구되는 길이로 부호 확장됩니다. 수가 너무 커서  $n$  문자에 맞지 않는다면 결과는 왼쪽에서 잘립니다.  $n$ 은 모든 정수 또는 0(영)이어야 합니다.

$n$ 을 생략하면, *wholenumber*는 모든 양의 정수 또는 0(영)이어야 하고 결과 길이는 필요한 만큼입니다. 그러므로 리턴된 결과는 앞에 '00'x 문자가 없습니다.

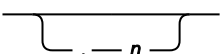
**구현 최대:** 출력 문자열에 250 이상 중요한 문자가 없을 수 있습니다. 그러나 추가 선행 기호 문자('00'x 및 'FF'x)가 있는 경우 더 긴 결과가 가능합니다.

## 예제

```
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(10진수를 16진으로)

D2X 함수는 16진으로 변환된 *wholenumber*, 10진수를 나타내는 문자 포맷으로 문자열을 리턴합니다.

➡ D2X( — *wholenumber*  ) ➡

리턴된 문자열은 값 A-F에 대해 대문자 알파벳을 사용하며 공백을 포함하지 않습니다.

$n$ 을 지정하는 경우, 문자에 최종 결과의 길이입니다. 변환 후 입력 문자열은 요구되는 길이로 부호 확장됩니다. 수가 너무 커서  $n$  문자에 맞지 않는다면 왼쪽에서 잘립니다.  $n$ 은 모든 정수 또는 0(영)이어야 합니다.

$n$ 을 생략하면, *wholenumber*는 모든 양의 정수 또는 0(영)이어야 하고 리턴된 결과 앞에 0(영)이 없습니다.

**구현 최대:** 출력 문자열에 500 이상 중요한 16진 문자가 없을 수 있습니다. 그러나 추가 선행 기호 문자(0과 F)가 있는 경우 더 긴 결과가 가능합니다.

#### 예제

```
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 함수는 오류 번호 *n*과 연관된 REXX 오류 메시지를 리턴합니다.

➡ ERRORTXT( — *n* — ) ➡

*n*은 범위 0-99에 있어야 하며 기타 값은 오류입니다. *n*이 허용된 범위에 있지만 정의된 REXX 오류 번호가 아니면 null 문자열을 리턴합니다. 오류 번호 및 메시지에 관한 완벽한 설명은 [391 페이지의 『제 31 장 오류 번호 및 메시지』](#)의 내용을 참조하십시오.

#### 예제

```
ERRORTXT(16)  -> 'Label not found'
ERRORTXT(60)  -> ''
```

## EXTERNALS

EXTERNALS 함수는 터미널 입력 버퍼(시스템 외부 이벤트 큐)에서 요소의 수를 리턴합니다.

➡ EXTERNALS( — ) ➡

CICS에 동등한 버퍼가 없습니다. 그러므로 REXX의 CICS 구현에서 EXTERNALS 함수는 항상 0을 리턴합니다.  
예:

```
EXTERNALS()  -> 0 /* Always */
```

## FIND

FIND 함수는 *string*에서 발견된 *phrase*의 첫 번째 단어의 단어 수를 리턴하거나 또는 *phrase*를 찾을 수 없거나 *phrase*에 단어가 없는 경우 0을 리턴합니다.

WORDPOS는 이 유형의 단어 검색을 위해 선호되는 기본 제공 함수입니다. [206 페이지의 『WORDPOS\(단어 위치\)』](#)의 내용을 참조하십시오.

➡ FIND( — *string* — , — *phrase* — ) ➡

*phrase*은 공백으로 구분된 단어의 시퀀스입니다. *string* 또는 *phrase*의 단어 사이의 다중 공백은 비교를 위해 단일 공백으로 처리됩니다.

#### 예제

```
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 함수는 NUMERIC FORM의 현재 설정을 리턴합니다.

➡ FORM( — ) ➡

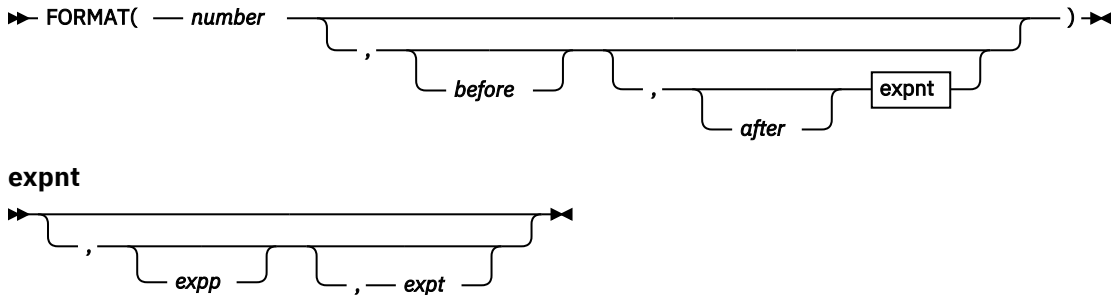
NUMERIC 명령어 [164 페이지](#)의 『NUMERIC』의 내용을 참조하십시오.

예

```
FORM() -> 'SCIENTIFIC' /* by default */
```

## FORMAT

FORMAT 함수는 반올림되고 형식화된 *number*를 리턴합니다.



*number*는 표준 REXX 규칙에 따라 반올림된 첫 번째 수이며, 연산 *number*+0이 수행되는 것과 같습니다. 결과는 정확하게 *number*만 지정한 경우 이 연산의 결과입니다. 다른 옵션을 지정한 경우, *number*는 다음과 같이 형식이 지정됩니다.

*before* 및 *after* 옵션은 결과의 정수와 소수 파트에 사용되는 문자 수를 설명합니다. 이들 수 중 하나 또는 모두를 생략한 경우, 해당 파트에 사용된 문자 수는 필요에 따라 다릅니다.

*before*가 해당 숫자(및 음수의 부호)의 정수 파트를 포함하는 데 충분하지 않는 경우 오류가 발생합니다. *before*가 해당 파트에 필요한 것보다 큰 경우, 해당 숫자는 왼쪽을 공백으로 채웁니다. *after*가 해당 숫자의 소수 파트와 동일한 크기가 아닌 경우 적합하도록 해당 숫자가 반올림됩니다(또는 0(영)으로 확장됨). 0을 지정하면 해당 숫자가 정수로 반올림됩니다.

예제

```
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'
```

처음 3개의 인수는 이전에 설명된 대로입니다. 또한 *expp* 및 *expt*는 결과의 지수 파트를 제외하며, 기본적으로 DIGITS 및 FORM의 현재 NUMERIC 설정에 따라 형식이 지정됩니다. *expp*는 지수 파트의 자릿수를 설정합니다. 기본값은 필요한 만큼 사용됩니다(0(영)일 수 있음). *expt*는 지수 표기법 사용을 위한 트리거 포인트를 설정합니다. 기본값은 NUMERIC DIGITS의 현재 설정입니다.

*expp*가 0이면, 지수가 제공되지 않으며 숫자는 필요에 따라 0(영)이 추가된 단순한 양식으로 표시됩니다. *expp*가 지수를 포함하기에 충분하지 않으면 오류가 발생합니다.

정수나 소수 파트에 필요한 자릿수가 각각 *expt*를 초과하거나 *expt*의 두 배인 경우, 지수 표기법이 사용됩니다. *expt*가 0인 경우 지수가 0이 아니면 지수 표기법이 항상 사용됩니다. (*expp*가 0인 경우 이는 *expt*의 0 값을 대체합니다.) 0이 아닌 *expp*가 지정될 때 지수가 0인 경우, *expp*+2 공백은 결과의 지수 파트로 제공됩니다. 지수가 0이고 *expp*가 지정되지 않은 경우 간단한 양식이 사용됩니다. *expp*는 10 미만이어야 하지만 다른 인수에 제한이 없습니다.

예:

```
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 함수는 NUMERIC FUZZ의 현재 설정을 리턴합니다.

➡ FUZZ( — ) ➡

NUMERIC 함수 [164 페이지](#)의 『NUMERIC』의 내용을 참조하십시오.

예

```
FUZZ()    ->    0    /* by default */
```

## INDEX

INDEX 함수는 한 문자열 *needle*의 문자 위치를 다른 문자열 *haystack*에 리턴하거나 문자열 *needle*을 찾을 수 없거나 널 문자열인 경우 0을 리턴합니다.

POS는 한 문자열 위치를 다른 문자열에서 얻기 위해 선호되는 기본 제공 함수입니다. [197 페이지](#)의 『POS(위치)』의 내용을 참조하십시오.

➡ INDEX( — *haystack* — , — *needle* — , — *start* — ) ➡

기본적으로 검색은 *haystack*(*start*의 값은 1임)의 첫 번째 문자에서 시작됩니다. 이는 다른 *start* 지점(양수여야 함)을 지정하여 대체할 수 있습니다.

예제

```
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 함수는 길이 *length*로 채우거나 잘린 문자열 *new*를 *n*번째 문자 다음의 문자열 *target*으로 삽입합니다.

➡ INSERT( — *new* — , — *target* — ➡

➡ , — *n* — , — *length* — , — *pad* — ) ➡

*n*의 기본값은 0이며, 문자열 시작 전에 삽입됨을 의미합니다. 지정되면, *n* 및 *length*는 모든 양의 정수 또는 0(영)이어야 합니다. *n*이 대상 문자열의 길이보다 큰 경우, 또한 *string* 앞에 채우기가 추가됩니다. *length*의 기본값은 *new*의 길이입니다. *length*가 문자열 *new*의 길이 미만이면, INSERT는 길이 *length*로 *new*를 자릅니다. 기본 *pad* 문자는 공백입니다.



## 예제

```
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, '-', 1) -> '123--abc'
```

## JUSTIFY

JUSTIFY 함수는 공백으로 구분된 단어 사이에 *pad* 문자를 추가하여 양쪽 여백에 맞도록 형식이 지정된 *string*을 리턴합니다.

➡ JUSTIFY( — *string* — , — *length* — , — *pad* — ) ➡

너비 *length*에 수행됩니다(*length*는 모든 양의 정수이거나 0(영)이어야 함). 기본 *pad* 문자는 공백입니다.

첫 번째 단계는 SPACE(*string*)이 실행된 것처럼 추가 공백을 제거하는 것입니다(즉, 다중 공백은 단일 공백으로 변환되며 선행 및 후행 공백이 제거됨). *length*가 변경된 문자열의 너비 미만이면 문자열이 오른쪽에서 잘려지며 후행 공백이 제거됩니다. 추가 *pad* 문자가 왼쪽에서 오른쪽으로 고르게 추가되어 필요한 길이를 제공하며 *pad* 문자는 단어 사이의 공백을 대체합니다.

## 예제

```
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(마지막 위치)

LASTPOS 함수는 한 문자열 *needle*의 마지막 발생 위치를 다른 문자열 *haystack*에 리턴합니다.

➡ LASTPOS( — *needle* — , — *haystack* — , — *start* — ) ➡

LASTPOS는 *needle*이 널 문자열이거나 이 문자열을 찾을 수 없는 경우 0을 리턴합니다. 기본적으로 검색은 *haystack*의 마지막 문자에서 시작되어 뒤로 스캔합니다. 뒤로 스캔이 시작되는 점을 *start*로 지정하여 이를 대체할 수 있습니다. *start*는 양의 정수여야 하며 해당 값보다 크거나 생략되는 경우 기본값 LENGTH(*haystack*)로 설정됩니다. POS 함수 [197 페이지의 『POS\(위치\)』](#)의 내용도 참조하십시오.

## 예제

```
LASTPOS(' ', 'abc def ghi') -> 8
LASTPOS(' ', 'abcdefghi')   -> 0
LASTPOS('xy', 'efgxyz')     -> 4
LASTPOS(' ', 'abc def ghi', 7) -> 4
```

## LEFT

LEFT 함수는 *string*의 가장 왼쪽 *length*자를 포함하여 *length*자 길이의 문자열을 리턴합니다.

➡ LEFT( — *string* — , — *length* — , — *pad* — ) ➡

리턴되는 문자열은 필요에 따라 오른쪽에 *pad* 문자로 채워집니다(또는 잘림). 기본 *pad* 문자는 공백입니다. *length*는 양의 정수이거나 0이어야 합니다. LEFT 함수는 정확히 다음과 같습니다.

➡ SUBSTR — ( — *string* — , — 1 — , — *length* — , — *pad* — ) ➡

## 예제

```
LEFT('abc d',8)      -> 'abc d '
LEFT('abc d',8,'.')  -> 'abc d...'
LEFT('abc def',7)    -> 'abc de'
```

## LENGTH

LENGTH 함수는 *string*의 길이를 리턴합니다.

➡ LENGTH( — *string* — ) ➡

## 예제

```
LENGTH('abcdefgh')  -> 8
LENGTH('abc defg')   -> 8
LENGTH('')           -> 0
```

## LINESIZE

LINESIZE 함수는 현재 터미널 행 너비(언어 프로세서가 SAY 명령어를 사용하여 표시된 행을 구분하는 점)를 리턴합니다.

➡ LINESIZE( — ) ➡

LINESIZE는 다음과 같은 경우 기본값 80을 리턴합니다.

- 접속된 터미널이 없고 출력의 경로가 재지정되고 있습니다.

**참고:** 터미널이 접속되어 있는지 여부를 판별하려면 REXX/CICS 명령 [SCRNINFO](#), 382 페이지의 『[SCRNINFO](#)』를 지정하십시오. 접속된 터미널이 없는 경우 화면 높이 및 화면 너비의 값은 0입니다.

## MAX(최대)

MAX 함수는 지정된 목록에서 현재 NUMERIC 설정에 따라 형식화된 가장 큰 숫자를 리턴합니다.

➡ MAX( — *number* — ) ➡

**구현 최대:** 최대 20개의 숫자를 지정할 수 있으며 추가 인수가 필요한 경우 MAX에 대한 호출을 중첩시킬 수 있습니다.

## 예제

```
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(최소)

MIN 함수는 지정된 목록에서 현재 NUMERIC 설정에 따라 형식화된 가장 작은 숫자를 리턴합니다.

➡ MIN( — *number* — ) ➡

**구현 최대:** 최대 20개의 숫자를 지정할 수 있으며 추가 인수가 필요한 경우에는 MIN에 대한 호출을 중첩시킬 수 있습니다.

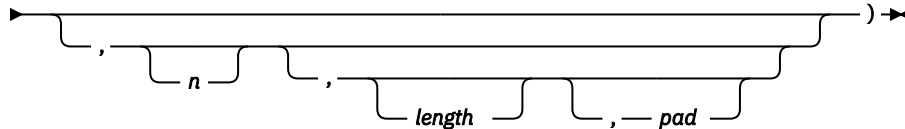
## 예제

```
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 함수는  $n$ 번째 문자에서 시작되고 문자열 *new*로 오버레이되며 길이 *length*로 채워지거나 잘리는 문자열 *target*을 리턴합니다.

➡ OVERLAY( — *new* — , — *target* →



오버레이는 원래 *target* 문자열의 끝을 넘어 확장될 수 있습니다. *length*를 지정하는 경우 이는 양의 정수이거나 0이어야 합니다. *length*의 기본값은 *new*의 길이입니다.  $n$ 이 대상 문자열의 길이보다 큰 경우 *new* 문자열 앞에 채우기가 추가됩니다. 기본 *pad* 문자는 0이고  $n$ 의 기본값은 1입니다.  $n$ 을 지정하는 경우 이는 양의 정수여야 합니다.

## 예제

```
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(위치)

POS 함수는 한 문자열 *needle*의 위치를 다른 문자열 *haystack*에 리턴합니다.

➡ POS( — *needle* — , — *haystack* — , — *start* — ) ➡

*needle*이 널 문자열이거나 이 문자열을 찾을 수 없는 경우 또는 *start*가 *haystack*의 길이보다 큰 경우 POS는 0을 리턴합니다. 기본적으로 검색은 *haystack*의 첫 번째 문자에서 시작됩니다(즉, *start*의 값은 1임). 검색이 시작되는 위치인 *start*(양의 정수여야 함)를 지정하여 이를 대체할 수 있습니다. INDEX 및 LASTPOS 함수인 [194 페이지](#)의 『INDEX』 및 [195 페이지](#)의 『LASTPOS(마지막 위치)』의 내용도 참조하십시오.

## 예제

```
POS('day', 'Saturday')        -> 6
POS('x', 'abc def ghi')       -> 0
POS(' ', 'abc def ghi')       -> 4
POS(' ', 'abc def ghi', 5)    -> 8
```

## QUEUED

QUEUED 함수는 함수가 호출될 때 외부 데이터 큐에 남아 있는 행의 수를 리턴합니다. 남아 있는 행이 없는 경우 PULL 또는 PARSE PULL이 터미널 입력 버퍼로부터 읽습니다. 대기 중인 터미널 입력이 없는 경우 이로 인해 콘솔 읽기가 발생합니다.

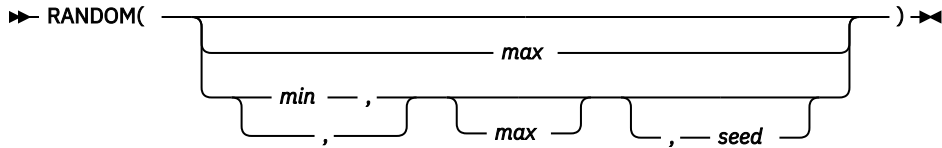
➡ QUEUED( — ) ➡

## 예

```
QUEUED() -> 5 /* Perhaps */
```

## RANDOM

RANDOM 함수는 *min* - *max* 범위(범위 값 포함)의 음이 아닌 의사 난수를 리턴합니다.



*max* 또는 *min* 또는 둘 다를 지정하는 경우 *max*에서 *min*을 뺀 값은 100000을 초과할 수 없습니다. *min* 및 *max*의 기본값은 각각 0 및 999로 설정됩니다. 결과의 반복 가능한 시퀀스를 시작하려면 참고 198 페이지의 『1』에 설명된 대로 특정 *seed*를 세 번째 인수로 사용하십시오. 이 *seed*는 0 - 999999999 범위의 양의 정수여야 합니다.

## 예제

```
RANDOM() -> 305
RANDOM(5,8) -> 7
RANDOM(2) -> 0 /* 0 to 2 */
RANDOM(,1983) -> 123 /* reproducible */
```

## 참고:

1. 의사 난수의 예측 가능한 시퀀스를 얻으려면 RANDOM를 여러 번 사용하십시오. 단, *seed*는 처음에만 지정하십시오. 예를 들어, 6개의 변수를 가진 기울어짐이 없는 금형의 40개 스로우(throw)를 시뮬레이션하려면 다음을 수행하십시오.

```
sequence = RANDOM(1,6,12345) /* any number would */
/* do for a seed */
do 39
  sequence = sequence RANDOM(1,6)
end
say sequence
```

랜덤으로 표시될 수 있도록 초기 *seed*를 사용하여 숫자가 수학적으로 생성됩니다. 이 프로그램을 다시 실행하면 동일한 시퀀스가 다시 생성됩니다. 다른 초기 *seed*를 사용하면 거의 확실하게 다른 시퀀스가 생성됩니다. *seed*를 제공하지 않는 경우 처음 RANDOM이 호출되면 시각 계시기의 마이크로초 필드가 *seed*로 사용되므로 프로그램은 실행될 때마다 거의 항상 다른 결과를 제공합니다.

2. 난수 생성기는 전체 프로그램에 대해 글로벌이며 현재 시드는 내부 루틴 호출 전체에 걸쳐 저장되지 않습니다.

## REVERSE

REVERSE 함수는 역으로 바꾼 *string*을 리턴합니다.

```
➡ REVERSE( — string — ) ➡
```

## 예제

```
REVERSE('ABc.') -> '.cBA'
REVERSE('XYZ ') -> ' ZYX'
```

## RIGHT

RIGHT 함수는 *string*의 가장 오른쪽 *length*자를 포함하여 *length*자 길이의 문자열을 리턴합니다.

➡ RIGHT( — *string* — , — *length* — ) ➡

└──────────┘  
          , — *pad* —

리턴되는 문자열은 필요에 따라 왼쪽에 *pad* 문자로 채워집니다(또는 잘림). 기본 *pad* 문자는 공백입니다. *length* 는 양의 정수이거나 0이어야 합니다.

#### 예제

```
RIGHT('abc d',8)    -> ' abc d'
RIGHT('abc def',5)  -> 'c def'
RIGHT('12',5,'0')   -> '00012'
```

## SIGN

이 SIGN 함수는 *number*이 부호를 표시하는 숫자를 리턴합니다. *number*는 오퍼레이션 *number*+0이 수행된 것처럼 먼저 표준 REXX 규칙에 따라 반올림됩니다.

➡ SIGN( — *number* — ) ➡

SIGN은 *number*가 0보다 적은 경우 -1을 리턴하고 0인 경우에는 0을 리턴하고 0보다 큰 경우에는 1을 리턴합니다.

#### 예제

```
SIGN('12.3')        -> 1
SIGN(' -0.307')     -> -1
SIGN(0.0)            -> 0
```

## SOURCELINE

SOURCELINE 함수는 *n*을 생략하는 경우 프로그램에서 최종 행의 행 번호를 리턴하거나, *n*을 지정하는 경우 프로그램에서 *n*번째 행을 리턴합니다.

➡ SOURCELINE( — *n* — ) ➡

지정된 경우, *n*은 모든 양의 정수여야 하고, 프로그램에서 최종 행의 수를 초과하지 않아야 합니다.

#### 예제

```
SOURCELINE()        -> 10
SOURCELINE(1)        -> '/* This is a 10-line REXX program */'
```

## SPACE

SPACE 함수는 각 단어 사이에 *n pad* 문자가 있는 *string*에 공백으로 구분된 단어를 리턴합니다.

➡ SPACE( — *string* — , — *n* — , — *pad* — ) ➡

*n*을 지정하는 경우 이는 양의 정수이거나 0이어야 합니다. 0인 경우에는 모든 공백이 제거됩니다. 선행 및 후미 공백은 항상 제거됩니다. *n*의 기본값은 1이고 기본값 *pad* 문자는 공백입니다.

#### 예제

```
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

208 페이지의 『REXX/CICS에서 제공되는 외부 기능』의 내용을 참조하십시오.

## STRIP

STRIP 함수는 사용자가 지정하는 *option*에 따라 선행 또는 후미 문자를 포함하거나 둘 다 제거된 *string*을 리턴합니다.

➡ STRIP( — *string* — , — *option* — , — *char* — ) ➡

다음은 올바른 옵션입니다. 대문자만 필요하며 뒤에 오는 모든 문자는 무시됩니다.

### 둘 다

*string*에서 선행 및 후미 문자를 둘 다 제거합니다. 이는 기본값입니다.

### Leading

*string*에서 선행 문자를 제거합니다.

### Trailing

*string*에서 후미 문자를 제거합니다.

세 번째 인수인 *char*은 제거될 문자를 지정하고 기본값은 공백입니다. *char*을 지정하는 경우 정확히 한 문자 길이어야 합니다.

### 예제

```
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(하위 문자열)

SUBSTR 함수는 *n*번째 문자에서 시작되고 길이는 *length*이며 필요한 경우 *pad*로 채워지는 *string*의 하위 문자열을 리턴합니다.

➡ SUBSTR( — *string* — , — *n* — , — *length* — , — *pad* — ) ➡

*n*은 양의 정수여야 합니다. *n*이 LENGTH(*string*)보다 큰 경우 채움 문자만 리턴됩니다.

*length*를 생략하는 경우 문자열의 나머지가 리턴됩니다. 기본 *pad* 문자는 공백입니다.

### 예제

```
SUBSTR('abc',2)      -> 'bc'
SUBSTR('abc',2,4)     -> 'bc '
SUBSTR('abc',2,6,'.') -> 'bc...'
```

**참고:** 일부 상황에서 구문 분석 템플릿의 위치(숫자) 패턴이 하위 문자열을 선택하는 데 있어 더 편리합니다(특히 한 문자열에서 두 개 이상의 하위 문자열을 추출해야 하는 경우). LEFT 및 RIGHT 함수도 참조하십시오.

## SUBWORD

SUBWORD 함수는 *n*번째 문자에서 시작되고 최대 *length* 공백 구분 문자 수인 *string*의 하위 문자열을 리턴합니다.



## 시간

자정 이후 시간의 수를 제공하는 최대 2개 문자를 리턴하며 형식은 hh(결과가 0인 경우를 제외하고 선행 0 또는 공백이 없음)입니다.

## Long

hh:mm:ss.uuuuuu(uuuuuu는 초의 분수이며 마이크로초 단위임) 형식으로 된 시간을 리턴합니다. 결과의 처음 8개 문자는 보통 양식에 대한 규칙과 동일한 규칙을 준수하고 분수 파트는 항상 6자리 숫자입니다.

## 분

자정 이후 분 수를 제공하는 최대 4개의 문자를 리턴하며 형식은 mmmm(결과가 0인 경우를 제외하고 선행 0 또는 공백이 없음)입니다.

## 일반

이전에 설명한 대로 시간을 기본 형식인 hh:mm:ss로 리턴합니다. 시간의 값은 00 - 23이고 분 및 초는 00 - 59입니다. 이들 모두는 항상 두 자리 숫자입니다. 초의 모든 분수는 무시됩니다(시간은 반올림되지 않음). 기본값입니다.

## Reset

경과 시간 시계(나중에 설명됨)가 시작되거나 재설정된 후의 초 수.마이크로초인 ssssssss.uuuuuu를 리턴하고 경과 시간 시계도 0으로 재설정합니다. 이 숫자에는 선행 0 또는 공백이 없으며 NUMERIC DIGITS의 설정은 이 숫자에 영향을 주지 않습니다. 분수 부분에는 항상 6자리 숫자가 있습니다.

## 초

자정 후 초 수를 제공하는 최대 5개 문자를 리턴하며 형식은 sssss(결과가 0인 경우를 제외하고 선행 0 또는 공백이 없음)입니다.

**참고:** 한 절에서 DATE 또는 TIME에 대한 첫 번째 호출을 하면 시간소인이 작성된 다음 해당 절의 이들 함수에 대한 모든 호출에 사용됩니다. 따라서 단일 표현식 또는 절 둘 다 또는 DATE나 TIME 함수에 대한 다중 호출은 서로 일관성을 가지도록 보장됩니다.

**구현 최대:** 경과 시간의 초 수가 9자리를 초과하면(31.6년 경과와 같음) 결과로 오류가 발생합니다.

## 예제

다음 예제에서는 시간을 오후 4:54로 가정합니다.

```
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) */
```

## 경과 시간 시계

TIME 함수를 사용하여 실제(경과된) 시간 간격을 측정할 수 있습니다. 프로그램에서 TIME('E') 또는 TIME('R')에 대한 첫 번째 호출 시 경과 시간 시계가 시작되며 둘 중 한 호출은 0을 리턴합니다. 그런 다음 TIME('E') 및 TIME('R')에 대한 호출은 그 첫 번째 호출 이후 또는 TIME('R')에 대한 마지막 호출 이후 경과된 시간을 리턴합니다.

이 시계는 내부 루틴 호출 전체에 저장됩니다. 이는 내부 루틴이 해당 호출자가 시작한 시간 시계를 상속받음을 의미합니다. 내부 루틴이 시계를 재설정해도 호출자가 수행하는 시간은 영향을 받지 않습니다. 경과 시간 시계의 예제:

```
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 */
```

**참고:** 단일 절에서의 시간 일관성에 대한 이전 노트를 참조하십시오. 경과 시간 시계는 TIME과 DATE에 대한 기타 호출에 동기화되어 단일 절에서의 경과 시간 시계에 대한 다중 호출은 항상 동일한 결과를 리턴합니다. 같은



이유로 두 개의 일반적인 TIME/DATE 결과 사이의 간격은 경과 시간 시계를 사용하여 정확히 계산할 수 있습니다.

## TRACE

TRACE 함수는 현재 유효한 추적 조치를 리턴하고 선택적으로 설정을 변경합니다.

➡ TRACE(                      ) ➡

*option*

*option*을 지정하면 추적 설정을 선택합니다. 이는 다음과 같은 올바른 접두부 중 하나여야 합니다. ? 또는 ! 또는 TRACE 명령어와 연관된 영문자 옵션 중 하나(즉, A, C, E, F, I, L, N, O, R 또는 S로 시작됨) 또는 둘 다.

TRACE 명령어와는 달리 TRACE 함수는 대화식 디버그가 활성화인 경우라도 추적 조치를 변경합니다. 또한 TRACE 명령어와는 다르게 *option*이 숫자가 아닙니다.

### 예제

```
TRACE()      ->  '?R' /* maybe */
TRACE('O')   ->  '?R' /* also sets tracing off */
TRACE('?I')   ->  'O'  /* now in interactive debug */
```

## TRANSLATE

TRANSLATE 함수는 각 문자가 다른 문자로 변환되거나 변경되지 않은 상태인 *string*을 리턴합니다. 이 함수를 사용하여 *string*에서 문자를 다시 정렬할 수도 있습니다.

➡ TRANSLATE( — *string* ➡

,                      ,                      , —                      ) ➡

*tableo*                      *tablei*                      *pad*

출력 테이블은 *tableo*이고 입력 변환 테이블은 *tablei*입니다. TRANSLATE는 *string*의 각 문자에 대해 *tablei*를 검색합니다. 문자를 찾은 경우 *tableo*의 해당 문자가 결과 문자열에 사용되고 *tablei*에 중복이 있는 경우 첫 번째(가장 왼쪽) 발생이 사용됩니다. 문자를 찾을 수 없는 경우 *string*의 원래 문자가 사용됩니다. 결과 문자열은 항상 *string*의 길이와 같은 길이입니다.

테이블의 길이에는 제한이 없습니다. 변환 테이블을 둘 다 지정하지 않고 *pad*를 생략하면 *string*은 단순히 대문자로 변환됩니다(즉, 소문자 a - z이 대문자 A - Z로 변환됨). 그러나 *pad*를 포함시킬 경우 언어 프로세서는 전체 문자열을 *pad* 문자로 변환합니다. *tablei*는 XRANGE('00'x, 'FF'x)로 기본 설정되고 *tableo*는 널 문자열로 기본 설정되며 필요에 따라 *pad*로 채워지거나 잘립니다. 기본 *pad*는 공백입니다.

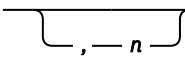
### 예제

```
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', XRANGE('00'x, 'Q')) ->  'APQ '
TRANSLATE('4123', 'abcd', '1234') ->  'dabc'
```

**참고:** 마지막 예제는 문자열에서 문자를 다시 정렬하기 위해 TRANSLATE 함수를 사용하는 방법을 보여줍니다. 이 예제에서 두 번째 인수로 지정된 4자 문자열의 마지막 문자는 문자열의 시작 부분으로 이동합니다.

## TRUNC(자르기)

TRUNC 함수는 *number*의 정수 파트와 *n*개의 소수점 이하 자리 수를 리턴합니다.

➡ TRUNC( — *number* — ) ➡  


기본  $n$ 은 0이고 소수점이 없는 정수를 리턴합니다.  $n$ 을 지정하는 경우 이는 양의 정수이거나 0이어야 합니다.  $number$ 는 오퍼레이션  $number+0$ 이 수행된 것처럼 먼저 표준 REXX 규칙에 따라 반올림됩니다. 그러면 숫자가  $n$ 개의 소수점 이하 자리 수로 잘립니다(또는 필요한 경우 지정된 길이를 만들기 위해 후미 0이 추가됨). 결과가 지수 양식이어서는 안됩니다.

**참고:**  $number$ 는 함수가 처리하기 전에 필요한 경우 NUMERIC DIGITS의 현재 설정에 따라 반올림됩니다.

## 예제

```
TRUNC(12.3)      -> 12
TRUNC(127.09782,3) -> 127.097
TRUNC(127.1,3)   -> 127.100
TRUNC(127,2)     -> 127.00
```

## USERID

USERID 함수는 사용자가 CICS에 사인온한 경우 CICS 사인온사용자 ID를 리턴하거나 CICS 리전 기본 사용자 ID를 리턴합니다(CICS 시스템 프로그래머에 의해 한 명이 지정된 경우).

➡ USERID( — ) ➡

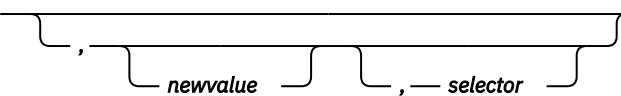
리턴된 값이 항상 8바이트 길이가 되도록 사용자 ID는 오른쪽이 공백으로 채워집니다.

## 예

```
USERID() -> 'ARTHUR' /* Maybe */
```

## VALUE

VALUE 함수는  $name$ (종종 동적으로 구성됨)이 나타내는 기호의 값을 리턴하고 선택적으로 이 값에 새 값을 지정합니다.

➡ VALUE( — *name* — ) ➡  


기본적으로 VALUE는 현재 REXX 변수 환경을 참조하지만  $selector$ 를 지정하려는 경우 값은 RLS여야 합니다. RLS의 선택기를 지정하면 작동하는 변수는 REXX 변수가 아니라 RK RLS(REXX List System) 변수입니다. 이 함수를 사용하여 REXX 변수를 참조하는 경우  $name$ 은 올바른 REXX 기호여야 합니다. (SYMBOL 함수를 사용하여 이를 확인할 수 있습니다.)  $name$ 의 소문자는 대문자로 변환됩니다. 가능한 경우 복합 이름의 대체(143 페이지의 『복합 기호』 참조)가 발생합니다.

$newvalue$ 를 지정하는 경우 이름 지정된 변수에 이 새 값이 지정됩니다. 이는 리턴되는 결과에 영향을 주지 않습니다. 즉, 이 함수는 새 지정 전과 마찬가지로  $name$ 의 값을 리턴합니다.

## 예제

```
/* 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 */
```

다음 예제는 RLS 변수에 저장된 REXX 변수 FRED의 VALUE을 리턴합니다.

```
/* REXX EXEC - ASSIGN FIND VALUE OF FRED */
FRED = 7
'RLS VARPUT FRED \USERS\userid\'
X = VALUE(FRED,,RLS)
SAY X
/* X now = 7 */
```

#### 참고:

1. VALUE 함수가 초기화되지 않은 REXX 변수를 참조하는 경우 이 변수의 기본값이 항상 리턴됩니다. NOVALUE 조건은 발생하지 않습니다. RLS 변수에 대한 참조는 NOVALUE을 발생시키지 않습니다.
2. 단일 리터럴 문자열로 *name*을 지정하고 *newvalue* 및 *selector*를 생략하는 경우 기호는 상수이므로 일반적으로 따옴표 안의 문자열이 전체 함수 호출을 대체할 수 있습니다. 예를 들어, `fred=VALUE('k');`는 NOVALUE 조건이 트랩되지 않는 한 지정 `fred=k;`와 같습니다. [233 페이지의 『제 22 장 조건 및 조건 트랩』](#)의 내용을 참조하십시오.

## VERIFY

VERIFY 함수는 기본적으로 *string*이 *reference*의 문자로만 구성되었는지 여부를 표시하는 숫자를 리턴합니다.

➡ VERIFY( — *string* — , — *reference* — , — *option* — , — *start* — ) ➡

VERIFY는 *string*의 모든 문자가 *reference*에 있는 경우 0을 리턴하고 그렇지 않은 경우에는 *reference*에 없는 *string*의 첫 번째 문자 위치를 리턴합니다.

*option*은 Nomatch(기본값) 또는 Match일 수 있습니다. (대문자만 필요합니다. 그 뒤에 오는 모든 문자는 무시되고 평소대로 대문자나 소문자일 수 있습니다.) Match를 지정하는 경우 이 함수는 *reference*에 있는 *string*의 첫 번째 문자 위치를 리턴하고 문자를 찾을 수 없는 경우에는 0을 리턴합니다.

*start*의 기본값은 1이므로 검색은 *string*의 첫 번째 문자에서 시작됩니다. 다른 *start* 지점(양수여야 함)을 지정하여 이를 대체할 수 있습니다.

*string*이 널인 경우 이 함수는 세 번째 인수의 값과 상관없이 0을 리턴합니다. 마찬가지로 *start*가 `iLENGTH(string)`보다 큰 경우 이 함수는 0을 리턴합니다. *reference*가 널인 경우 Match를 지정하면 이 함수는 0을 리턴합니다. 그렇지 않으면 이 함수는 *start* 값을 리턴합니다.

#### 예제

VERIFY('123','1234567890')	->	0
VERIFY('1Z3','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 함수는 *string*에서 *n*번째 공백으로 분리된 단어를 리턴하거나, *n*보다 작은 단어가 *string*에 있는 경우, 널 문자열을 리턴합니다.

➡ WORD( — *string* — , — *n* — ) ➡

*n*은 모든 양의 정수여야 합니다. 이 함수는 SUBWORD(*string*,*n*,1)과 정확하게 일치합니다.

## 예제

```
WORD('Now is the time',3)    ->  'the'
WORD('Now is the time',5)    ->  ''
```

## WORDINDEX

*string*에서 *n*번째 공백으로 분리된 단어에서 첫 번째 문자의 위치를 리턴하거나, *n*보다 작은 단어가 *string*에 있는 경우, WORDINDEX 함수는 0을 리턴합니다.

➡ WORDINDEX( — *string* — , — *n* — ) ➡

*n*은 모든 양의 정수여야 합니다.

## 예제

```
WORDINDEX('Now is the time',3)    ->  8
WORDINDEX('Now is the time',6)    ->  0
```

## WORDLENGTH

*string*에서 *n*번째 공백으로 분리된 단어의 길이를 리턴하거나, *n*보다 작은 단어가 *string*에 있는 경우, WORDINDEX 함수는 0을 리턴합니다.

➡ WORDLENGTH( — *string* — , — *n* — ) ➡

*n*은 모든 양의 정수여야 합니다.

## 예제

```
WORDLENGTH('Now is the time',2)    ->  2
WORDLENGTH('Now comes the time',2) ->  5
WORDLENGTH('Now is the time',6)    ->  0
```

## WORDPOS(단어 위치)

WORDPOS 함수는 *string*에서 찾은 *phrase*의 첫 번째 단어의 단어 수를 리턴하고 *phrase*가 단어를 포함하지 않거나 *phrase*를 찾을 수 없는 경우 0을 리턴합니다.

➡ WORDPOS( — *phrase* — , — *string* — , — *start* — ) ➡

*phrase* 또는 *string*에서 단어 사이의 다중 공백은 비교를 위해 단일 공백으로 처리되지만 그렇지 않은 경우에는 단어가 정확히 일치해야 합니다.

기본적으로 검색은 *string*의 첫 번째 단어에서 시작됩니다. 검색을 시작할 단어인 *start*(양의 정수여야 함)를 지정하여 이를 대체할 수 있습니다.

## 예제

```
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 함수는 *string*에서 공백으로 구분된 단어의 수를 리턴합니다.

➡ WORDS( — *string* — ) ➡

## 예제

```
WORDS('Now is the time') -> 4
WORDS(' ') -> 0
```

## XRANGE(16진 범위)

XRANGE 함수는 *start*와 *end* 간(범위 값 포함) 올바른 모든 1바이트 인코딩 문자열을 오름차순으로 리턴합니다.

➡ XRANGE( — *start* — , — *end* — ) ➡

*start*의 기본값은 '00'x이고 *end*의 기본값은 'FF'x입니다. *start*가 *end*보다 큰 경우 값은 'FF'x에서 '00'x로 줄 바꿈이됩니다. 지정된 경우 *start* 및 *end*가 단일 문자여야 합니다.

## 예제

```
XRANGE('a', 'f') -> 'abcdef'
XRANGE('03'x, '07'x) -> '0304050607'x
XRANGE(, '04'x) -> '0001020304'x
XRANGE('i', 'j') -> '898A8B8C8D8E8F9091'x /* EBCDIC */
XRANGE('FE'x, '02'x) -> 'FEFF000102'x
```

## X2B(16진을 2진으로)

이 함수는 2진으로 변환된 *hexstring*을 나타내는 문자 형식의 문자열을 리턴합니다.

➡ X2B( — *hexstring* — ) ➡

*hexstring*은 16진 문자의 문자열입니다. 길이에 제한이 없습니다. 각 16진 문자는 4자리 2진 숫자의 문자열로 변환됩니다. 가독성을 높이기 위해 *hexstring*에 공백을 선택적으로 포함시킬 수 있습니다(선행 또는 후미가 아닌 바이트 경계에만). 공백은 무시됩니다.

리턴된 문자열의 길이는 4의 배수이며 어떠한 공백도 포함하지 않습니다.

*hexstring*이 널인 경우 이 함수는 널 문자열을 리턴합니다.

## 예제

```
X2B('C3') -> '11000011'
X2B('7') -> '0111'
X2B('1 C1') -> '000111000001'
```

You can combine X2B with the functions D2X and C2X to convert numbers or character strings into binary form. 예:

```
X2B(C2X('C3'x)) -> '11000011'
X2B(D2X('129')) -> '10000001'
X2B(D2X('12')) -> '1100'
```

## X2C(16진을 문자로)

X2C 함수는 문자로 변환된 *hexstring*을 나타내는 문자 형식의 문자열을 리턴합니다.

➡ X2C( — *hexstring* — ) ➡

리턴된 문자열은 원래 *hexstring*의 바이트 수의 반입니다. *hexstring*의 길이에 제한이 없습니다. 필요한 경우 선행 0으로 채워서 16진수의 짝수로 만듭니다.

가독성을 높이기 위해 *hexstring*에 공백을 선택적으로 포함시킬 수 있습니다(선행 또는 후미가 아닌 바이트 경계에만). 공백은 무시됩니다.

*hexstring*이 널인 경우 함수는 널 문자열을 리턴합니다.

### 예제

```
X2C('F7F2 A2')    ->  '72s'      /* EBCDIC */
X2C('F7f2a2')    ->  '72s'      /* EBCDIC */
X2C('F')          ->  ' '        /* '0F' is unprintable EBCDIC */
```

## X2D(16진을 10진수로)

X2D 함수는 *hexstring*의 10진수 표현을 리턴합니다.

➡ X2D( — *hexstring* — , — *n* — ) ➡

*hexstring*은 16진 문자의 문자열입니다. 결과를 정수로 표시할 수 없는 경우 오류가 결과로 발생합니다. 즉, 결과에는 NUMERIC DIGITS의 현재 설정보다 많은 숫자를 포함해서는 안됩니다.

가독성을 지원하기 위해 *hexstring*에 선택적으로 공백을 포함시킬 수 있습니다(바이트 경계에만 포함시킬 수 있고 선행 또는 후미에는 해당되지 않음). 공백은 무시됩니다.

*hexstring*이 널인 경우 함수는 0을 리턴합니다.

*n*을 지정하지 않으면 *hexstring*이 부호 없는 2진수로 처리됩니다.

**구현 최대:** 입력 문자열에는 500자를 초과하는 16진 문자가 있을 수 없으며 이는 최종 결과 구성에 있어 중요한 사항입니다. 선행 기호 문자(0과 F)는 이 총계에 계수되지 않습니다.

### 예제

```
X2D('0E')        ->  14
X2D('81')        ->  129
X2D('F81')       ->  3969
X2D('FF81')      ->  65409
X2D('c6 f0'X)    ->  240      /* EBCDIC */
```

*n*을 지정하면 문자열을 *n*개의 16진 숫자로 표시되는 부호 있는 숫자로 간주됩니다. 가장 왼쪽 비트가 떨어져 있는 경우 이 숫자는 양수입니다. 그렇지 않은 경우에는 2보수 표기법에서 음수입니다. 두 경우 모두에서 정수로 변환되며 따라서 음수일 수 있습니다. *n*이 0인 경우 함수는 0을 리턴합니다.

필요한 경우 *hexstring*은 왼쪽에 0 문자("부호 확장된" 문자가 아님을 참고)로 채워지거나 왼쪽으로 *n*자까지 잘립니다. 예:

```
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/CICS에서 제공되는 외부 기능

추가 외부 기능이 REXX/CICS 환경에서 제공됩니다.

### STORAGE

STORAGE 함수는 *address*에서 시작하는 사용자 메모리로부터 *length* 바이트를 리턴합니다.

**중요사항:** 권한 부여된 함수입니다. REXX 사용자가 CICS 리전의 31비트 가상 스토리지를 표시 및/또는 수정하도록 허용하는 STORAGE 함수는 권한 부여된 exec 또는 권한 부여된 사용자에게 의해서만 성공적으로 호출될 수 있습니다.

➡ STORAGE( — *address* — , — *length* — , — *data* — ) ➡

*length*는 10진수입니다. 기본값은 1바이트입니다. *address*는 16진 수입니다. *address*의 상위 비트가 무시됩니다. 31비트 주소를 지정하십시오. 그렇지 않으면 다른 결과는 예측 불가능합니다. 64비트 주소를 지정할 수 없습니다.

이전 값이 검색된 후 *data*를 지정하는 경우, *address*에서 시작하는 스토리지는 *data*로 겹쳐집니다(*length* 인수가 이것에는 영향을 주지 않음).

**참고:** STORAGE 함수는 31비트 주소에 작용할 수 있습니다. STORAGE 함수는 64비트 주소에 작용할 수 없습니다.

**예**

```
/* 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

함수

➡ SYSSBA( — *row* — , — *col* — ) ➡

SYSSBA는 화면 *row,col*을 SBA(Set Buffer Address)로 변환합니다.

**row**

화면의 맨 위부터 계수하는 행 번호를 지정합니다.

**col**

열 번호를 지정합니다(화면의 왼쪽부터 계수함).

**참고:** SYSSBA 함수는 호출 시마다 터미널 모델을 조회하고 이를 사용하여 SBA 계산을 터미널 유형으로 조정합니다.

**예**

```
x = SYSSBA(10,20)
```

이 예제는 화면의 행 10, 열 20에 대해 설정된 3바이트 버퍼 주소를 REXX 변수 x로 리턴합니다.





## 제 20 장 구문 분석

구문 분석은 소스 문자열의 데이터를 분할하고 그 조각을 템플릿에서 이름 지정된 변수에 지정합니다.

구문 분석 명령어는 ARG, PARSE 및 PULL입니다. [152 페이지의 『ARG』](#), [166 페이지의 『PARSE』](#) 및 [169 페이지의 『PULL』](#)의 내용을 참조하십시오.

구문 분석할 데이터는 소스 문자열입니다. 템플릿은 소스 문자열을 분할하는 방법을 지정하는 모델입니다. 템플릿 중 가장 단순한 종류의 템플릿은 변수 이름 목록으로만 구성됩니다. 예를 들어, 다음과 같습니다.

```
variable1 variable2 variable3
```

이 종류의 템플릿은 소수 문자열을 공백 구분 단어로 구문 분석합니다. 보다 복잡한 템플릿은 변수 이름에 추가하여 패턴을 포함합니다.

### 문자열 패턴

분할할 위치를 지정하기 위한 소스 문자열의 일치 문자입니다. [213 페이지의 『문자열 패턴을 포함하는 템플릿』](#)의 내용을 참조하십시오.

### 위치 패턴

소스 문자열을 분할할 문자 위치를 표시합니다. [213 페이지의 『위치\(숫자\) 패턴을 포함하는 템플릿』](#)의 내용을 참조하십시오.

구문 분석은 기본적으로 2단계 프로세스입니다.

1. 패턴을 사용하여 소스 문자열을 적절한 하위 문자열로 구문 분석합니다.
2. 각 하위 문자열을 단어로 구문 분석합니다.

## 단어로 구문 분석하기 위한 단순 템플릿

다음은 구문 분석 명령어입니다.

```
parse value 'time and tide' with var1 var2 var3
```

이 명령어의 템플릿은 var1 var2 var3입니다. 구문 분석할 데이터는 키워드 PARSE VALUE와 키워드 WITH, 소스 문자열 time and tide 사이에 있습니다. 구문 분석은 다음과 같이 소스 문자열을 공백 구분 단어로 나누고 이를 템플릿에서 이름 지정된 변수에 지정합니다.

```
var1='time'  
var2='and'  
var3='tide'
```

이 예제에서 구문 분석할 소스 문자열은 리터럴 문자열 time and tide입니다. 다음 예제에서 소스 문자열은 변수입니다.

```
/* 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 VALUE는 소스 문자열의 소문자 a-z를 대문자 A-Z로 변환하지 않습니다. 문자를 대문자로 변환하려는 경우 PARSE UPPER VALUE를 사용하십시오. 대소문자에 대한 구문 분석 명령어의 효과의 요약은 [217 페이지의 『UPPER 사용』](#)의 내용을 참조하십시오.

모든 구문 분석 명령어는 소스 문자열의 파트를 템플릿의 이름 지정된 변수에 지정합니다. 소스 문자열의 특성 또는 기점의 차이점으로 인해 다양한 구문 분석 명령어가 있습니다. [218 페이지의 『구문 분석 명령어 요약』](#)의 내용을 참조하십시오.

PARSE VAR 명령어는 구문 분석할 소스 문자열이 항상 변수라는 점을 제외하고 PARSE VALUE와 비슷합니다. PARSE VAR에서는 소스 문자열을 포함하는 변수의 이름이 키워드 PARSE VAR을 따릅니다. 다음 예제에서는 변수 stars가 소스 문자열을 포함합니다. 템플리트는 star1 star2 star3입니다.

```
/* PARSE VAR example */
stars='Sirius Polaris Rigil'
parse var stars star1 star2 star3          /* star1='Sirius' */
                                           /* star2='Polaris' */
                                           /* star3='Rigil' */
```

템플리트의 모든 변수는 새 값을 받습니다. 템플리트에 소스 문자열의 단어보다 더 많은 수의 변수가 있는 경우 나머지 변수는 널(비어 있는) 값을 받습니다. 이는 단순 템플리트를 사용하여 단어로 구문 분석하는 경우 및 패턴을 포함하는 템플리트를 사용한 구문 분석 등 모든 구문 분석에 대해 true입니다. 다음은 단어로의 구문 분석을 사용하는 예제입니다.

```
/* More variables in template than (words in) the source string */
satellite='moon'
parse var satellite Earth Mercury          /* Earth='moon' */
                                           /* Mercury='' */
```

소스 문자열에 템플리트의 변수보다 많은 수의 단어가 있는 경우 템플리트의 마지막 변수는 나머지 데이터를 모두 받습니다. 다음은 예입니다.

```
/* 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...'*/
```

단어로의 구문 분석은 변수에 지정되기 전에 각 단어에서 선행 및 후미 공백을 제거합니다. 이에 대한 예외는 마지막 변수에 지정되는 단어 또는 단어 그룹입니다. 템플리트의 마지막 변수는 추가 선행 및 후미 공백을 유지하면서 나머지 데이터를 받습니다. 다음은 예입니다.

```
/* 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 ' */
```

소스 문자열 Earth에는 두 개의 선행 공백이 있습니다. 구문 분석에서는 var3='Earth'를 지정하기 전에 이들 공백 둘 다를(단어 분리자 공백 및 추가 공백) 제거합니다. Mars에는 세 개의 선행 공백이 있습니다. 구문 분석에서는 한 개의 단어 분리자 공백을 제거하고 다른 두 개의 선행 공백은 보존합니다. 또한 Mars와 Jupiter 사이의 5개 공백 모두와 Jupiter 뒤 후미 공백을 둘 다 보존합니다.

구문 분석에서는 템플리트가 한 개의 변수만 포함하는 경우 공백을 제거하지 않습니다. 예를 들어, 다음과 같습니다.

```
parse value ' Pluto ' with var1          /* var1=' Pluto '*/
```

## 플레이스홀더로 마침표 사용

템플리트에서 마침표는 플레이스홀더입니다. 변수 이름 대신 사용되지만 데이터를 받지 않습니다. 플레이스홀더는 불필요한 변수의 오버헤드를 저장합니다. 이는 다음의 경우 유용합니다.

- 변수 목록에서 "더미 변수"로서
- 문자열의 끝에서 원하지 않는 정보를 수집하는 경우.

첫 번째 예제의 마침표는 플레이스홀더입니다. 인접한 마침표는 공백으로 구분해야 합니다. 그렇지 않으면 오류가 결과로 발생합니다.

```
/* Period as a placeholder */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars . . brightest .          /* brightest='Sirius' */

/* Alternative to period as placeholder */
```

```
stars='Arcturus Betelgeuse Sirius Rigel'
parse var stars drop junk brightest rest /* brightest='Sirius' */
```

## 문자열 패턴을 포함하는 템플리트

문자열 패턴은 분할할 위치를 표시하는 소스 문자열의 문자와 일치합니다.

문자열 패턴은 리터럴이거나 변수일 수 있습니다.

### 리터럴 문자열 패턴

따옴표 안에 한 개 이상의 문자가 있습니다.

### 변수 문자열 패턴

왼쪽 소괄호 앞에 더하기(+) 또는 빼기(-) 또는 등호(=)가 없는 소괄호 내의 변수입니다. 세부사항은 [216 페이지의 『변수 패턴을 사용한 구문 분석』](#)의 내용을 참조하십시오.

두 가지 템플리트(단순 템플리트와 리터럴 문자열 패턴을 포함하는 템플리트)가 있습니다.

```
var1 var2 /* simple template */
var1 ' , ' var2 /* template with literal string pattern */
```

리터럴 문자열 패턴은 ' , '입니다. 이 템플리트는 다음을 수행합니다.

- 소스 문자열의 시작부터 일치의 첫 번째 문자(덱스트) (그러나 이 문자는 포함하지 않음) 사이에 있는 문자를 `var1`에 넣습니다.
- 일치의 마지막 문자(덱스트 뒤에 오는 공백 뒤의 문자) 뒤에 있는 문자로 시작하여 문자열의 끝에서 종료되는 문자를 `var2`에 넣습니다.

문자열 패턴이 포함된 템플리트는 데이터를 변수에 지정할 때 소스 문자열에서 일부 데이터를 생략할 수 있습니다. 다음 두 예제에서는 단순 템플리트를 리터럴 문자열 패턴을 포함하는 템플리트와 대조합니다.

```
/* Simple template */
name='Smith, John'
parse var name ln fn /* Assigns: ln='Smith, ' */
/* fn='John' */
```

덱스트는 남는다는 점에(변수 `ln`은 'Smith, '를 포함함) 유의하십시오. 다음 예제에서는 템플리트가 `ln ' , ' fn`입니다. 이는 덱스트를 제거합니다.

```
/* Template with literal string pattern */
name='Smith, John'
parse var name ln ' , ' fn /* Assigns: ln='Smith' */
/* fn='John' */
```

먼저 언어 프로세서가 ' , '에 대해 소스 문자열을 스캔합니다. 해당 지점에서 소스 문자열을 분할합니다. 변수 `ln`은 소스 문자열의 첫 번째 문자로 시작되고 일치 앞 마지막 문자로 끝나는 데이터를 수신합니다. 변수 `fn`은 일치 뒤 첫 번째 문자로 시작되고 문자열의 끝에서 종료되는 데이터를 수신합니다.

문자열 패턴이 포함된 템플리트는 패턴과 일치하는 소스 문자열의 데이터를 생략합니다. 문자열 패턴을 포함하는 템플리트가 소스 문자열에서 일치하는 데이터를 생략하지 않는 특수한 경우가 있습니다. [220 페이지의 『문자열과 위치 패턴 결합: 특수한 경우』](#)의 내용을 참조하십시오. 패턴에 공백이 없는 경우 변수 `fn`이 'John'(공백을 포함함)을 수신하므로 ' , '(공백 없음) 대신 ' , ' 패턴(공백 포함)을 사용했습니다.

소스 문자열에 문자열 패턴에 대한 일치가 포함되지 않은 경우 일치하지 않는 문자열 패턴 앞에 오는 모든 변수는 질문의 모든 데이터를 가져옵니다. 해당 패턴 뒤에 오는 모든 변수는 널 문자열을 수신합니다.

널 문자열을 찾을 수 없습니다. 널 문자열은 항상 소스 문자열의 끝과 일치합니다.

## 위치(숫자) 패턴을 포함하는 템플리트

위치 패턴은 소스 문자열에서 데이터를 분할하는 문자 위치를 식별하는 수입니다. 숫자는 정수여야 합니다.

절대적 위치 패턴은 다음과 같습니다.

- 더하기(+) 또는 빼기(-) 부호가 앞에 없거나 등호(=)가 앞에 있는 숫자.

- 왼쪽 괄호 앞에 등호가 있는 소괄호 안의 변수. 변수 위치 패턴에 관한 세부사항은 [216 페이지의 『변수 패턴을 사용한 구문 분석』](#)의 내용을 참조하십시오.

해당 숫자는 소스 문자열을 분할하는 절대 문자 위치를 지정합니다.

절대적 위치 패턴을 가진 템플리트는 다음과 같습니다.

```
variable1 11 variable2 21 variable3
```

숫자 11과 21은 절대적 위치 패턴입니다. 숫자 11은 입력 문자열에서 11번째 위치를 참조하며 21은 21번째 위치를 참조합니다. 이 템플리트의

- 소스 문자열에서 1 ~ 10 문자를 variable1에 넣으며
- 11 ~ 20 문자를 variable2에 넣으며
- 21에서 끝까지의 문자를 variable3에 넣습니다.

위치 패턴은 다음과 같이 레코드의 파일과 작업할 경우 대부분 유용할 수 있습니다.

```

character positions:
      1      11      21      40
+-----+-----+-----+
FIELDS: |LASTNAME |FIRST  |PSEUDONYM |end of
+-----+-----+-----+

```

다음 예는 이 레코드 구조를 사용합니다.

```

/* 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 */

```

소스 문자열은 먼저 문자 위치 11과 위치 21에서 분할됩니다. 언어 프로세서는 문자 1 ~ 10을 lastname으로, 문자 11 ~ 20을 firstname으로, 문자 21 ~ 40은 pseudonym으로 지정합니다.

템플리트는 다음일 수 있습니다.

```
1 lastname 11 firstname 21 pseudonym
```

(다음이 아닌:

```
lastname 11 firstname 21 pseudonym
```

) 1 지정은 옵션입니다.

선택적으로 템플리트에서 숫자 앞에 등호를 넣을 수 있습니다. 등호는 템플리트에서 숫자 앞의 등호가 없음과 동일합니다. 숫자는 소스 문자열의 특정 문자 위치를 참조합니다. 이 두 템플리트는 동일하게 작동합니다.

```

lastname 11 first 21 pseudonym
lastname =11 first =21 pseudonym

```

상대적인 위치 패턴은 더하기(+) 또는 빼기(-) 부호가 앞에 있는 숫자입니다. (왼쪽 소괄호가 앞에 있고 더하기(+) 또는 빼기(-) 부호가 있는 소괄호 내 변수일 수도 있습니다. 세부사항은 [216 페이지의 『변수 패턴을 사용한 구문 분석』](#)의 내용을 참조하십시오.)

숫자는 소스 문자열을 분할하는 상대적인 문자 위치를 지정합니다. 더하기 또는 빼기는 문자열의 시작에서(첫 번째 패턴의 경우) 또는 마지막 항목의 위치에서부터 각각 오른쪽 또는 왼쪽으로 이동함을 표시합니다. 마지막 일치 항목의 위치는 마지막 일치 항목의 첫 번째 문자입니다. 상대적인 위치 패턴으로 처리된 절대적 위치 패턴에 관한 동일한 예는 다음과 같습니다.

```

/* 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 */
```

부호와 숫자 사이의 공백은 중요하지 않습니다. 그러므로 +10 및 + 10은 의미가 동일합니다. +0이 유효한 상대적인 위치 패턴이라는 점을 유의하십시오.

절대 및 상대 위치 패턴은 교환 가능합니다(문자열 패턴이 변수 이름에 선행하고 위치 패턴이 변수 이름 다음에 오는 특별한 경우를 제외하고 220 페이지의 『문자열과 위치 패턴 결합: 특수한 경우』의 내용 참조). 절대 및 상대 위치 패턴의 예로부터의 템플리트는 동일한 결과를 제공합니다.

	lastname 11  lastname +10	firstname 21  firstname + 10	pseudonym   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.

위치 패턴만으로 일치되는 조작은 소스 문자열에서 이전 위치로 백업할 수 있습니다. 다음은 절대 위치 패턴을 사용한 예입니다.

```
/* 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" */
```

절대 위치 패턴 1은 소스 문자열에서 첫 번째 문자로 백업됩니다.

상대 위치 패턴으로, 빼기 부호 다음의 숫자는 이전 위치로 백업합니다. 다음은 상대 위치 패턴을 사용한 동일한 예입니다.

```
/* 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 */
```

이전 예에서, 상대 위치 패턴 -3은 소스 문자열에서 첫 번째 문자로 백업합니다.

마지막 두 개 예에서 템플리트는 동일합니다.

	var1 4   var1 +2	1   -3	var2 2   var2 +1	4 var3 5   +2 var3 +1	11 var4   +6 var4
Start at 2.	Non-inclusive stopping point is 4 (2+2=4).	Go to 1. (4-3=1)	Non-inclusive stopping point is 2 (1+1=2).	Go to 4 (2+2=4). Non-inclusive stopping point is 5 (4+1=5).	Go to 11 (5+6=11).

위치 패턴이 있는 템플리트를 사용하여 여러 번 지정할 수 있습니다.

```
/* 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. */
```

## 패턴 결합 및 단어로 구문 분석

템플리트가 다중 단어를 포함한 섹션으로 소스 문자열을 나눈 패턴을 포함하는 경우 어떤 일이 발생합니까? 문자열 및 위치 패턴은 소스 문자열을 서브 문자열로 나눕니다. 그런 다음 언어 프로세서는 템플리트의 섹션을 각 서브 문자열로 적용하고 단어로 구문 분석하기 위한 규칙을 적용합니다.

```

/* Combining string pattern and parsing into words          */
name='    John      Q.   Public'
parse var name fn init '.' ln      /* Assigns: fn='John'   */
                                   /*      init='      Q'   */
                                   /*      ln='      Public' */

```

패턴은 템플리트를 두 섹션으로 나눕니다.

- fn init
- ln

일치 패턴은 소스 문자열을 두 개의 하위 문자열로 분할합니다.

- ' John Q'
- ' Public'

언어 프로세서는 적절한 템플레이트 섹션을 기반으로 이 하위 문자열을 단어로 구문 분석합니다.

John은 3개의 선행 공백이 있습니다. 단어로의 구문 분석이 마지막 변수로부터를 제외하고 선행과 후행 공백을 제거하기 때문에 모든 것이 제거됩니다.

Q은 6개의 선행 공백이 있습니다. init이 템플리트의 해당 섹션에서 마지막 변수이기 때문에 구문 분석하면 한 단어 구분 기호 공백을 제거하며 나머지를 유지합니다.

하위 문자열 ' Public'의 경우, 구문 분석은 공백을 제거하지 않고 전체 문자열을 ln으로 지정합니다. ln이 템플리트의 이 섹션에서 유일한 변수이기 때문입니다.(공백 처리에 관한 세부사항은 [211 페이지의 『단어로 구문 분석하기 위한 단순 템플리트』](#)의 내용을 참조하십시오.)

```

/* 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'  */

```

패턴은 템플리트를 세 개의 섹션으로 나눕니다.

- var1 var2
- var3
- var4

일치 패턴은 개별적으로 단어로 구문 분석되는 세 개의 하위 문자열로 소스 문자열을 분할합니다.

- 'R E'
- ' X'
- ' X'

변수 var1은 'R'을 수신하며 var2는 'E'를 수신합니다. 각각이 템플리트의 이 섹션에서 유일한 변수이기 때문에 var3 및 var4 모두는 ' X'를 수신합니다(X 앞에 공백이 있는). (공백 처리에 관한 세부사항은 [211 페이지의 『단어로 구문 분석하기 위한 단순 템플리트』](#)의 내용을 참조하십시오.)

## 변수 패턴을 사용한 구문 분석

고정된 문자열 또는 숫자 대신 변수의 값을 사용하여 패턴을 지정하려고 합니다. 이러한 작업은 소괄호 안에 변수의 이름을 배치하여 수행합니다. 이는 변수 참조입니다.

### 이 태스크 정보

소괄호 내부 또는 외부에 공백은 필요하지 않습니다. 그러나 필요한 경우 추가할 수 있습니다.

다음 구문 분석 명령어의 템플리트는 다음 리터럴 문자열 패턴을 포함합니다. ' . '.

```

parse var name fn init ' . ' ln

```

다음은 해당 패턴을 변수 문자열 패턴으로 지정하는 방법입니다.

```
stringptrn='.'
parse var name fn init (stringptrn) ln
```

변수 이름 앞 소괄호에 등호, 더하기 또는 빼기 부호가 선행하지 않는 경우 이 변수의 값은 문자열 패턴으로 처리됩니다. 이 변수는 동일한 템플릿에서 이전에 설정된 변수일 수 있습니다. 예를 들어, 다음과 같습니다.

```
/* 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' */
```

등호, 더하기 또는 빼기 부호가 왼쪽 소괄호 앞에 있는 경우 이 변수의 값은 절대 또는 상대 위치 패턴으로 처리됩니다. 이 변수의 값은 양의 정수이거나 0이어야 합니다.

이 변수는 동일한 템플릿에서 이전에 설정된 변수일 수 있습니다. 다음 예제에서 처음 두 개 필드는 마지막 두 개 필드의 시작 문자 위치를 지정합니다.

```
/* 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' */
```

템플릿에서 위치 패턴 6이 필요한 이유는? 단어 구문 분석은 언어 프로세서가 패턴을 사용하여 소스 문자열을 하위 문자열로 나눈 후에 발생하는 점을 기억해 두십시오. 따라서 위치 패턴 =(pos1)은 언어 프로세서가 열 6의 문자열을 분할하여 공백 구분 단어 12와 26을 각각 pos1과 pos2로 지정할 후에만 =12로 올바르게 해석될 수 있습니다.

## UPPER 사용

PARSE 명령어에서 UPPER를 지정하면 구문 분석 후 문자를 대문자로 변환합니다(소문자 a-z를 대문자 A-Z로).

### 이 태스크 정보

다음 테이블은 대소문자에 관한 구문 분석 명령의 결과를 요약합니다.

구문 분석 전에 영문자를 대문자로 변환함	입력된 대소문자로 영문자를 유지보수함
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 명령어는 단순한 짧은 형식의 PARSE UPPER ARG입니다. PULL 명령어는 단순한 짧은 형식의 PARSE UPPER PULL입니다. 대문자 변환을 원하지 않는 경우, PARSE ARG(ARG 또는 PARSE UPPER ARG 대신)를 사용하고 PARSE PULL(PULL 또는 PARSE UPPER PULL 대신)을 사용하십시오.

## 구문 분석 명령어 요약

모든 구문 분석 명령어는 소스 문자열의 파트를 템플릿에서 이름 지정된 변수에 지정합니다.

### 이 태스크 정보

다음 템플릿은 소스 문자열이 시작되는 위치에 대해 요약합니다.

명령어	소스 문자열이 시작되는 위치
ARG PARSE ARG	서브루틴 또는 함수에 대한 호출에서 프로그램 또는 인수를 호출할 때 나열하는 인수입니다.
PARSE EXTERNAL	터미널 입력 버퍼로부터의 다음 행
PARSE NUMERIC	숫자 제어 정보(NUMERIC 명령어로부터).
PULL PARSE PULL	외부 데이터 큐의 헤드에 있는 문자열입니다. (큐가 비어 있는 경우 기본 입력을 사용합니다. 일반적으로 기본 입력은 터미널입니다.)
PARSE SOURCE	실행되는 프로그램에 대한 정보를 제공하는 시스템 제공 문자열입니다.
PARSE VALUE	명령어에서 키워드 VALUE와 키워드 WITH 사이의 표현식입니다.
PARSE VAR <i>name</i>	<i>name</i> 의 값을 구문 분석합니다.
PARSE VERSION	언어, 언어 레벨 및 (3단어) 날짜를 지정하는 시스템 제공 문자열입니다.

## 구문 분석 명령어 예제

소스 문자열을 단어로 구문 분석하는 예제가 표시됩니다.

### 이 태스크 정보

#### 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         */
/* Assigns: var1='RED'; var2='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
END
Say new; exit /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit
```

ARG는 구문 분석을 하기 전에 영문자를 대문자로 변환합니다. 서브루틴에 대한 호출의 인수가 포함된 ARG의 예제는 219 페이지의 [『여러 문자열 구문 분석』](#)에 있습니다.

#### PARSE ARG

PARSE ARG가 구문 분석 전에 영문자를 대문자로 변환하지 않는다는 점을 제외하고 ARG와 동일하게 작동합니다.

#### 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

211 페이지의 [『단어로 구문 분석하기 위한 단순 템플릿』](#)의 예제를 참조하십시오.

## PARSE VAR

211 페이지의 [『단어로 구문 분석하기 위한 단순 템플릿』](#)에서 시작하여 구문 분석 정보 전체에 대한 예제를 참조하십시오.

## PARSE VERSION

```
parse version . level .
say level                    /* Displays: "3.48" */
```

## PULL

PULL이 구문 분석 전에 영문자를 대문자로 변환하지 않는다는 점을 제외하고 PARSE PULL과 동일하게 작동합니다.

## 고급 구문 분석 정보

고급 구문 분석에는 다중 문자열 구문 분석, DBCS 문자를 사용한 구문 분석 및 특수한 경우가 포함됩니다.

구문 분석의 개념적 보기를 묘사하는 플로우 차트가 제공됩니다.

## 여러 문자열 구문 분석

ARG 및 PARSE ARG만 둘 이상의 소스 문자열을 보유할 수 있습니다. 다중 문자열을 구문 분석하기 위해 쉼표로 구분된 여러 템플릿을 지정할 수 있습니다.

## 이 태스크 정보

다음은 예입니다.

```
parse arg template1, template2, template3
```

이 명령어는 키워드 PARSE ARG와 세 개의 쉼표로 구분된 템플릿으로 구성됩니다. (ARG 명령어의 경우 구문 분석할 소스 문자열은 프로그램을 호출하거나 서브루틴 또는 함수를 호출할 때 지정하는 인수로부터 생성됩니다.) 각 쉼표는 다음 문자열로 이동하기 위한 구문 분석기에 대한 명령어입니다.

예제:

```
/* 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 프로그램이 명령으로 시작될 때 한 개의 인수 문자열만 인식됩니다. 다음의 경우 구문 분석을 위해 여러 인수 문자열을 전달할 수 있습니다.

- 한 개의 REXX 프로그램이 CALL 명령어 또는 함수 호출을 사용하여 다른 REXX 프로그램을 호출하는 경우.
- 다른 언어로 작성된 프로그램이 REXX 프로그램을 시작하는 경우.

소스 문자열보다 많은 템플릿이 있는 경우 나머지 템플릿의 각 변수는 널 문자열을 받습니다. 템플릿보다 많은 수의 소스 문자열이 있는 경우 언어 프로세서는 나머지 소스 문자열을 무시합니다. 템플릿이 비어 있거나 (한 행에 두 개의 쉼표) 또는 변수 이름을 포함하지 않은 경우 다음 템플릿 및 소스 문자열로 구문 분석이 진행됩니다.

## 문자열과 위치 패턴 결합: 특수한 경우

절대 및 상대 위치 패턴이 동일하게 작동하지 않는 특수한 경우가 있습니다.

### 이 태스크 정보

문자열 패턴을 포함하는 템플릿을 사용한 구문 분석이 패턴과 일치하는 소스 문자열의 데이터를 건너뛰는 방식을 보여줍니다(213 페이지의 『문자열 패턴을 포함하는 템플릿』 참조). 그러나 다음 시퀀스를 포함하는 템플릿은 일치하는 데이터를 건너뛰지 않습니다.

- 문자열 패턴
- 변수 이름
- 상대적 위치 패턴

상대적 위치 패턴은 문자열 패턴과 일치하는 첫 번째 문자에 상대적으로 이동합니다. 따라서 지정에는 해당 문자열 패턴과 일치하는 소스 문자열의 데이터가 포함됩니다.

```
/* 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" */
```

다음은 이 템플릿이 작동하는 방식입니다.

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 문자 구문 분석

DBCS 문자 구문 분석은 SBCS 문자 구문 분석을 할 때와 같은 규칙을 따릅니다.

### 이 태스크 정보

리터럴 문자열과 기호는 DBCS 문자를 포함할 수 있지만 숫자는 SBCS 문자에 있어야 합니다. DBCS 구문 분석의 예제는 433 페이지의 『PARSE』의 내용을 참조하십시오.

## 구문 분석 단계 세부사항

구문 분석의 개념적 보기를 제공하는 그림이 제공됩니다.

다음 세 개의 그림은 사용자가 구문 분석의 개념을 이해할 수 있도록 돕습니다. 그림에는 오류 케이스가 포함되지 않음을 참고하십시오.

그림에는 정의가 다음과 같은 용어가 포함됩니다.

#### 문자열 시작

소스 문자열(또는 하위 문자열)의 시작입니다.

#### 문자열 끝

소스 문자열(또는 하위 문자열)의 끝입니다.

#### 길이

소스 문자열의 길이입니다.

#### 일치 시작

소스 문자열에 있으며 일치의 첫 번째 문자입니다.

#### 일치 끝

소스 문자열에 있습니다. 문자열 패턴의 경우 일치의 끝 뒤에 오는 첫 번째 문자입니다. 위치상 패턴의 경우 일치 시작과 같습니다.

#### 일치 위치

소스 문자열에 있습니다. 문자열 패턴의 경우 첫 번째 일치 문자입니다. 위치 패턴의 경우 일치하는 문자의 위치입니다.

#### token

템플릿에서 구별되는 구문적 요소(예: 변수, 마침표, 패턴 또는 쉼표)입니다.

위치 패턴의 숫자 값입니다. 이는 상수이거나 변수의 해석된 값일 수 있습니다.

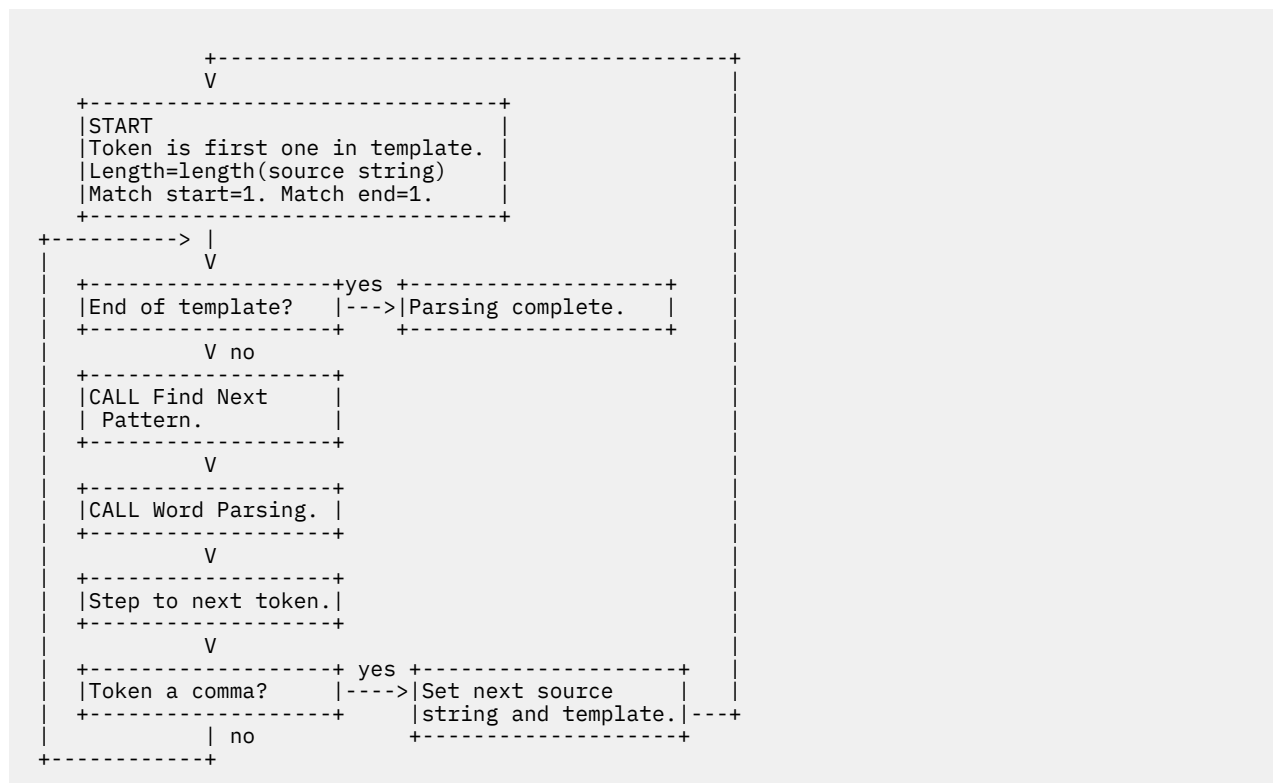


그림 48. 구문 분석의 개념 개요

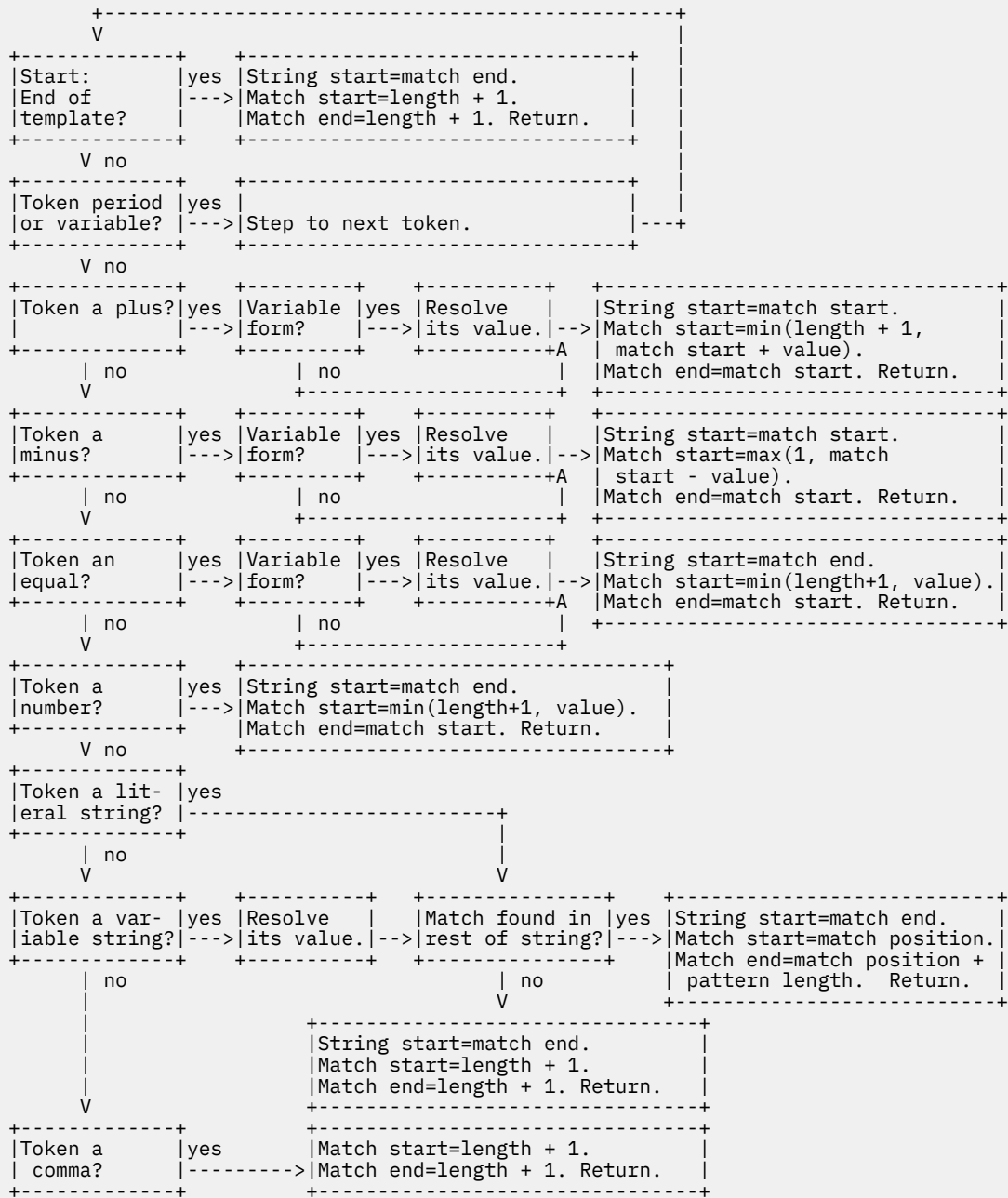


그림 49. 다음 패턴 찾기의 개념 보기





## 제 21 장 숫자와 산술 오퍼레이션

REXX는 최대한 자연스러운 방식으로 일반적인 산술 오퍼레이션(더하기, 빼기, 곱하기 및 나누기)을 정의합니다. 이는 다음의 규칙은 학교 및 대학에서 일반적으로 배우는 규칙임을 의미합니다.

그러나 이러한 기능의 디자인 중에 사람 및 애플리케이션에 따라 간혹 예상 불가능한 방식으로 규칙이 상당히 달라짐(실제로 일반적으로 허용되는 것보다 훨씬 더 크게 다름)이 발견되었습니다. 따라서 여기에 설명된 산술은 대부분의 애플리케이션에서 허용 가능한 결과를 제공해야 하는 중간물(가장 단순하지는 않지만)입니다.

### 소개: 수

Numbers(즉, REXX 산술 연산과 내장 함수에 입력으로 사용된 문자열)은 매우 유연하게 표현될 수 있습니다. 선행 및 후행 공백이 허용되며 지수 표기법이 사용될 수 있습니다.

일부 유효한 수는 다음과 같습니다.

```
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          */
```

지수 표기법에서 수는 지수, 사용 전 숫자가 곱해지는 10의 거듭제곱을 포함합니다. 지수는 소수점이 이동하는 방법을 표시합니다. 그러므로 이전 예에서 4E9는 4000000000을 쓰는 단순한 방법이며 0.73e-7은 0.000000073의 단축형입니다.

산술 연산자는 덧셈(+), 뺄셈(-), 곱셈(\*), 거듭제곱(\*\*), 나누기(/), 접두부 추가(+) 및 접두부 빼기(-)를 포함합니다. 또한 두 개의 추가 나누기 연산자가 있습니다. 정수로 나누기(%)는 나누고 정수 부분을 리턴합니다. 나머지(//)는 나누고 나머지를 리턴합니다.

산수 조작의 결과는 명확한 규칙에 따라 문자열로서 형식됩니다. 이 규칙에서 가장 중요한 점은 다음과 같습니다(세부사항은 226 페이지의 『산술 기능의 정의』 참조).

- 결과는 유효 숫자의 일부 최대 수까지 계산됩니다(기본값은 9지만 NUMERIC DIGITS 명령어로 이를 변경하여 무엇이든 원하는 정확성을 제공할 수 있습니다). 그러므로, 결과는 9 이상 숫자를 필요로 하며, 일반적으로 9자리로 반올림됩니다. 예를 들어, 2를 3으로 나누면 0.666666667이 됩니다(정확한 정확도를 위해 숫자의 무한 수를 필요로 함).
- 나누기와 거듭제곱을 제외하고, 후행 0(영)이 유지됩니다(가장 인기있는 계산기와 상반되며, 결과의 10진수 부분에서 모든 후행 0(영)을 제거함). 예를 들어 다음과 같습니다.

```
2.40 + 2    ->    4.40
2.40 - 2    ->    0.40
2.40 * 2    ->    4.80
2.40 / 2    ->    1.2
```

이 동작은 대부분의 계산(특히 금융 계산)에 대해 바람직합니다.

필요한 경우, STRIP 함수(200 페이지의 『STRIP』 참조) 또는 1로 나누어 후행 0(영)을 제거할 수 있습니다.

- 0 결과는 항상 단일 숫자 0으로 표현됩니다.
- 지수 양식은 NUMERIC DIGITS의 값과 설정에 따라 결과에 사용됩니다(기본값은 9). 10진수 앞에 필요한 자릿수가 NUMERIC DIGITS 설정을 초과하거나 점 뒤의 자릿수가 NUMERIC DIGITS 설정보다 두 배인 경우, 숫자는 지수 표기법으로 표시됩니다.

```
1e6 * 1e6    ->    1E+12          /* not 1000000000000 */
1 / 3E10     ->    3.33333333E-11 /* not 0.000000000333333333 */
```

## 산술 기능의 정의

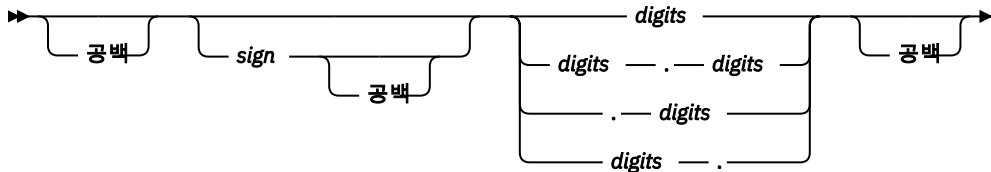
REXX 언어의 정확한 산술 기능의 정의가 제공됩니다.

### 숫자

REXX의 숫자는 선택적으로 소수점이 있는 하나 이상의 10진수를 포함하는 문자열입니다.

이 정의의 확장에 대해서는 230 페이지의 『지수 표기법』의 내용을 참조하십시오. 소수점은 숫자에 임베드될 수 있고 접두부 또는 접미부일 수 있습니다. 이 방식으로 구성된 숫자(및 선택적 소수점)의 그룹에는 선행 또는 후미 공백 및 숫자 또는 소수점 앞에 와야 하는 선택적 부호(+ 또는 -)가 있을 수 있습니다. 이 부호 앞이나 뒤에도 공백이 올 수 있습니다.

따라서 숫자는 다음과 같이 정의됩니다.



#### 공백

하나 이상의 공백입니다.

#### sign

+ 또는 -입니다.

#### digits

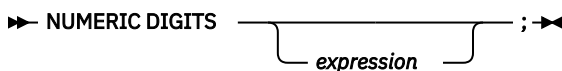
하나 이상의 10진수(0-9)입니다.

마침표 하나만으로는 올바른 숫자가 아님에 유의하십시오.

### 정밀도

정밀도는 오퍼레이션의 결과로 생성될 수 있는 최대 유효 숫자 수입니다.

정밀도는 다음 명령어에 의해 제어됩니다.



*expression*은 평가되고 평가 결과가 양의 정수여야 합니다. 이는 계산이 수행되는 정밀도(유효 숫자의 수)를 정의합니다. 결과는 필요한 경우 해당 정밀도로 반올림됩니다.

이 명령어에 *expression*을 지정하지 않거나 프로그램이 시작된 후 NUMERIC DIGITS 명령어가 처리된 경우 기본 정밀도가 사용됩니다. 기본 정밀도에 대한 REXX 표준은 9입니다.

NUMERIC DIGITS는 기본값 9 미만의 값을 설정할 수 있습니다. 그러나 작은 값은 주의하여 사용하십시오. 정밀도의 손실과 그로 인해 요청되는 반올림은 모든 REXX 계산(예: DO 루프의 제어 변수에 대해 새 값 계산)에 영향을 미칩니다.

### 산술 연산자

REXX 산술은 모두 두 항에서 수행되는 연산자 +, -, \*, /, %, // 및 \*\* (더하기, 빼기, 곱하기, 나누기, 정수 나누기, 나머지 및 거듭제곱)와 둘 다 단일 항에서 수행되는 접두부 더하기 및 빼기 연산자에 의해 수행됩니다.

모든 산술 연산 전에 오퍼레이션되는 항에서는 선행 0이 제거됩니다(소수점의 위치에 주의하고 숫자를 구성하는 모든 숫자가 0인 경우 한 개의 0만 남겨 둡). 그런 다음 계산에 사용되기 전에 DIGITS + 1 유효 숫자로 잘립니다(필요한 경우). (추가 숫자는 “보호” 숫자입니다. 보호 숫자는 숫자가 필요한 정밀도로 반올림되면 오퍼레이션의 끝에서 검사되므로 정확도를 높입니다.) 그러면 이 오퍼레이션은 다음에 오는 개별 오퍼레이션에서 설명된 대로 해당 정밀도의 두 배 정밀도로 수행됩니다. 오퍼레이션이 완료되면 NUMERIC DIGITS 명령어에 의해 지정된 정밀도로 결과가 반올림됩니다(필요한 경우).



반올림은 기존의 방식으로 수행됩니다. 결과의 최소 유효 숫자 오른쪽 숫자("보호 숫자")가 검사되고 값 5 - 9는 반올림되고 0 - 4는 반내림됩니다. 짝수/홀수 반올림에는 항상 전체적 정밀도로 계산하기 위한 기능이 필요하므로 REXX에 대해 정의된 메커니즘은 아닙니다.

기존의 0은 그 앞에 0 숫자가 있지 않는 한 소수점 앞에 제공됩니다. 유효 후미 0은 0의 결과가 항상 단일 숫자 0으로 표현되는 점만 제외하고 적용되는 규칙에 따라 더하기, 빼기 및 곱하기를 위해 보존됩니다.

FORMAT 기본 제공 함수(193 페이지의 『FORMAT』 참조)는 제공된 표준 결과가 요구사항을 충족시키지 못하는 경우 특정 형식으로 표현되도록 합니다.

## 산술 오퍼레이션 규칙: 기본 연산자

기본 연산자(더하기, 빼기, 곱하기 및 나누기)는 다음과 같이 숫자에 대해 작동합니다.

### 더하기와 빼기

두 숫자 중 하나가 0인 경우 필요하면 NUMERIC DIGITS 숫자로 반올림되는 다른 한 숫자가 결과로 사용됩니다 (적절하게 부호 조정). 그렇지 않은 경우에는 두 숫자가 필요에 따라 오른쪽 및 왼쪽으로 최대 DIGITS + 1 숫자로 확장된 다음(따라서 더 작은 절대값을 가지는 숫자는 오른쪽에 있는 숫자 일부 또는 전부를 잃을 수 있음) 적절하게 추가되거나 제해집니다.

예제:

```
xxx.xxx + yy.yyyyy
```

다음과 같이 됩니다.

```
xxx.xxx00
+ 0yy.yyyyy
-----
zzz.zzzzz
```

그러면 결과가 NUMERIC DIGITS의 현재 설정으로 반올림됩니다(필요한 경우 더하기 후에 왼쪽에 있는 추가 "자리 올림 수"를 고려하고 그렇지 않은 경우에는 추가되거나 삭감되는 항의 최상위 유효 숫자에 해당되는 위치에서 계수함). 마지막으로 유효하지 않은 선행 0이 제거됩니다.

접두부 연산자는 동일한 규칙을 사용하여 평가됩니다. 연산 +number 및 -number는 각각 0+number 및 0-number로 계산됩니다.

### 곱하기

숫자를 함께 곱하면("긴 곱하기") 두 피연산자의 합계 길이만큼 길 수 있는 숫자가 결과로 산출됩니다.

예제:

```
xxx.xxx * yy.yyyyy
```

다음과 같이 됩니다.

```
zzzzz.zzzzzzz
```

그러면 결과가 반올림되어 결과의 첫 번째 유효 숫자부터 NUMERIC DIGITS의 현재 설정까지 계수합니다.

### 나누기

다음 나누기의 경우 다음 단계가 수행됩니다.

```
yyy / xxxxx
```

먼저 숫자 yyy는 숫자 xxxxx보다 크게 될 때까지 오른쪽에 0을 사용하여 확장됩니다(0의 확장이 의미하는 10의 거듭제곱으로 변경이 수행됨에 주의하면서). 따라서 이 예제에서 yyy는 yyy00이 될 수 있습니다. 그런 다음 기존의 긴 나누기가 수행됩니다. 이는 다음과 같이 작성될 수 있습니다.

```

      zzzz
      +-----
xxxxx | yyy00

```

결과(zzzz)의 길이는 예제에서 가장 오른쪽 z은 최소한(확장된) y 숫자의 가장 오른쪽 숫자만큼 오른쪽으로 확장되어야 합니다. 나누기 중에 y 숫자는 필요에 따라 추가로 확장됩니다. z 숫자는 나누기가 중지되고 결과가 반올림되는 지점인 NUMERIC DIGITS+1 숫자까지 증가될 수 있습니다. 나누기가 완료된(그리고 필요한 경우 반올림) 뒤에는 유효 후미 0이 제거됩니다.

## 기본 연산자의 예제

다음 예제는 방금 설명한 규칙의 기본적인 의미를 보여줍니다.

```

/* 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

```

**참고:** 모든 기본 연산자에서 오퍼레이션되고 있는 항의 소수점 위치는 임의적입니다. 오퍼레이션은 이후 계산 및 적용되고 있는 지수가 포함된 정수 오퍼레이션으로 수행될 수 있습니다. 따라서 결과의 유효 숫자는 오퍼레이션에 포함된 항 중 하나에 있는 소수점의 위치에 따라 다릅니다.

## 산술 오퍼레이션 규칙: 추가 연산자

거듭제곱(\*\*), 정수 나누기(%) 및 나머지(/) 연산자에 대한 오퍼레이션 규칙을 설명합니다.

### 거듭제곱

\*\* (거듭제곱) 연산자는 숫자를 거듭제곱(양수, 음수 또는 0일 수 있음)으로 높입니다. 거듭제곱은 정수여야 합니다. (오퍼레이션의 두 번째 항은 231 페이지의 『REXX에 의해 직접 사용되는 숫자』에 설명된 대로 정수여야 하고 필요한 경우 DIGITS 숫자로 반올림됩니다.) 음수인 경우 거듭제곱의 절대값이 사용되고 결과는 반전됩니다(1로 나뉘어짐). 거듭제곱을 계산하기 위해서는 거듭제곱으로 표시된 횟수만큼 숫자에 그 숫자 자체를 곱하며 마지막으로 후미의 0은 제거됩니다(결과를 1로 나눈 것처럼).

실제로 거듭제곱은 왼쪽에서 오른쪽 2진 감소의 프로세스로 계산됩니다(이유는 229 페이지의 『1』 참조).  $a^{**}n$ 의 경우:  $n$ 은 2진으로 변환되고 임시 누산기는 1로 설정됩니다.  $n = 0$ 인 경우 초기 계산이 완료됩니다, (따라서  $0^{**}0$ 을 포함하여 모든  $a$ 에 대해  $a^{**}0 = 1$ 입니다.) 그렇지 않은 경우 각 비트(0이 아닌 비트 중 첫 번째 비트에서 시작됨)는 왼쪽에서 오른쪽으로 검사합니다. 현재 비트가 1인 경우에는 누산기에  $a$ 를 곱합니다. 이제 모든 비트 검사가 완료된 경우 초기 계산이 완료됩니다. 그렇지 않은 경우에는 누산기를 제공하고 곱하기를 위해 다음 비트를 검사합니다. 초기 계산이 완료되면 거듭제곱이 음수인 경우 임시 결과가 1로 나뉘어집니다.

곱하기 및 나누기는 산술 오퍼레이션 규칙에 따라 DIGITS + L + 1 숫자의 정밀도로 수행됩니다. L은 정수  $n$ 의 정수 파트 숫자의 길이입니다(즉, 기본 제공 함수 TRUNC( $n$ )가 사용된 것처럼 소수점을 제외함). 마지막으로 결과는 필요한 경우 NUMERIC DIGITS 숫자로 반올림되고 유효하지 않은 후미 0은 제거됩니다.

### 정수 나누기

%(정수 나누기) 연산자는 두 개의 숫자를 나누고 결과의 정수 파트를 리턴합니다. 리턴되는 결과는 피제수가 제수보다 큰 동안에는 피제수에서 제수를 반복해서 뺀 결과로 생성되는 값으로 정의됩니다. 이 빼기 중에 피제수와 제수 둘 다의 절대값이 사용됩니다. 최종 결과의 부호는 일반적인 나누기의 결과와 같습니다.

리턴된 결과에는 분수 파트가 없습니다(즉, 뒤에 소수점 또는 0이 오지 않음). 결과를 정수로 표시할 수 없는 경우 이 오퍼레이션은 오류 상태이고 실패하게 됩니다. 즉, 결과는 NUMERIC DIGITS의 현재 설정보다 많은 수의 숫자를 포함해서는 안됩니다. 예를 들어, 10000000000%3에는 결과(3333333333)에 대해 10개의 숫자가 필요하므로 NUMERIC DIGITS 9가 유효한 경우 실패합니다. 이 연산자는 일반 나눗기를 자른 것(반올림의 영향을 받을 수 있음)과 같은 결과를 제공하지 않을 수 있음에 유의하십시오.

## 나머지

// (나머지) 연산자는 정수 나눗기의 나머지를 리턴하고 이전에 설명한 대로 정수 나눗기 계산 오퍼레이션 후 피제수의 유수로 정의됩니다. 나머지의 부호는 0이 아닌 경우 원래 피제수의 부호와 같습니다.

이 오퍼레이션은 정수 나눗기와 같은 조건에서는 실패합니다(즉, 동일한 두 항에 대한 정수 나눗기가 실패하는 경우 나머지를 계산할 수 없음).

## 추가 연산자의 예제

다음 예제에서는 거듭제곱, 정수 나눗기 및 나머지 연산자를 사용합니다.

```
/* 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
```

## 참고:

1. 거듭제곱을 계산하기 위한 특정 알고리즘은 효율적이고(최적은 아니라도) 수행되는 실제 곱하기의 수를 상당히 줄이므로 이 알고리즘이 사용됩니다. 따라서 반복 곱하기의 더 단순한 정의에 비해 더 나은 성능을 제공합니다. 결과가 반복 곱하기의 결과와 다를 수 있으므로 이 알고리즘이 여기에 정의됩니다.
2. 정수 나눗기 및 나머지 연산자는 표준 나눗기 오퍼레이션의 부산물로 계산될 수 있도록 정의됩니다. 나눗기 프로세스는 정수 결과가 사용 가능해지는 대로 종료됩니다. 피제수의 유수는 나머지입니다.

## 숫자 비교

숫자 문자열을 비교하기 위해 임의의 비교 연산자를 사용할 수 있습니다.

비교 연산자는 [138 페이지의 『비교 연산자』](#) 섹션에 나열되어 있습니다. 그러나 숫자 비교에 ==, \==, ^=, >>, \>>, ^>>, <<, \<< 및 ^<<를 사용해서는 안됩니다. 선행 및 후미 공백 및 선행 0은 이러한 연산자에서 유효 숫자이기 때문입니다.

숫자 값의 비교는 두 숫자를 뺀 다음(차이를 계산) 결과를 0과 비교할 때 유효합니다. 즉, 오퍼레이션은 다음과 같습니다.

```
A ? Z
```

여기서, ? 는 임의의 숫자 비교 연산자이고 다음과 동일합니다.

```
(A - Z) ? '0'
```

따라서 이는 그 등식을 판별하는 REXX 뺄셈 규칙을 적용하여 뺄 때 두 숫자 간 차이입니다.

fuzz라고 하는 수량은 두 숫자의 비교에 영향을 미칩니다. 이는 비교에서 동등한 값으로 간주되기 위해 먼저 허용될 수 있는 두 숫자 간 차이의 양을 제어합니다. FUZZ 값은 다음 명령어를 통해 설정됩니다.

```
➡ NUMERIC FUZZ expression ;➡
```

여기에서 *expression*의 결과는 정수이거나 0이어야 합니다. 기본값은 0입니다.

FUZZ의 효과는 각 숫자 비교에 대한 FUZZ 값만큼 DIGITS의 값을 임시로 줄이는 것입니다. 즉, 비교 중 FUZZ 숫자를 뺀 DIGITS의 정밀도로 숫자를 뺍니다. FUZZ 설정은 명확하게 DIGITS 미만이어야 합니다.

## 예

## 지수 표기법

수에 대한 상기 설명은 숫자를 설명하는 문자열이 매우 길 수 있다는 점에서 "순수한" 숫자를 설명합니다. 예를 들어, 다음과 같습니다.

은 다음을 제공

밋

은 다음을 제공

큰 수 및 작은 수 모두의 경우, 지수 표기법의 일부 양식이 긴 숫자를 더 읽기 쉽게 하고 극단적인 경우에도 실행 가능하게 하는 데 유용할 수 있습니다. 또한 "단순한" 양식이 잘못된 정보를 제공할 때마다 지수 표기법이 사용됩니다.

```
numeric digits 5
say 54321*54321
```

은 긴 양식으로 2950800000을 표시합니다. 명확하게 잘못된 것이므로 결과는 2.9508E+9로 대신 표시됩니다.  
그러므로 수의 정의는 다음으로 확장됩니다.

E 다음에 오는 정수는 수에 적용되는 10의 거듭제곱을 표시합니다. E는 대문자나 소문자일 수 있습니다.

사용자에 숫자인 것처럼 보이지 않더라도 특정 문자열은 수입니다. 특히, 0E123(123 거듭제곱으로 올린 0) 및 1E342(342 거듭제곱으로 올린 1)과 같이 지수 표기법으로의 수 형식이 숫자이기 때문입니다. 또한 0E123=0E567과 같은 비교는 1의 true 결과를 제공합니다(0은 0과 같음). 문제를 방지하기 위해 수가 아닌 문자열 비교 시 엄격한 비교 연산자를 사용하십시오.

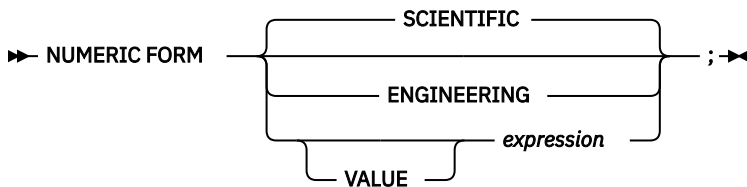
다음은 일부 예제입니다.

```
12E7  = 120000000    /* Displays "1" */
12E-5 = 0.00012      /* Displays "1" */
-12e4 = -120000      /* Displays "1" */
0e123 = 0e456        /* Displays "1" */
0e123 == 0e456       /* Displays "0" */
```

선행하는 수는 항상 입력 데이터에 유효합니다. NUMERIC DIGITS의 설정에 따라 계산의 결과는 기존 또는 지수 양식으로 리턴됩니다. 10진수 앞에 필요한 자릿수가 DIGITS를 초과하거나 해당 위치 뒤의 자릿수는 DIGITS의 두 배를 초과한 경우 지수 양식이 사용됩니다. REXX가 생성하는 지수 양식에는 항상 E 다음에 기호가 있어 가독성을 개선합니다. 지수가 0인 경우, 지수 파트가 생략됩니다. 즉 E+0의 지수 파트가 생성되지 않습니다.

숫자를 지수 양식으로 명시적으로 변환하거나 FORMAT 기본 제공 함수를 사용하여 긴 양식으로 강제로 표시할 수 있습니다(193 페이지의 『FORMAT』 참조).

지수 표기법은 0이 아닌 단일 숫자가 소수점 왼쪽에 표시되므로 10의 거듭제곱을 조정하는 지수 표기법의 양식입니다. 엔지니어링 표기법은 소수점 앞에 1 ~ 3자릿수가 표시되는 지수 표기법의 양식이며(단순히 0이 아님) 10의 거듭제곱은 3의 배수로 항상 표현됩니다. 그러므로 정수 파트는 1 ~ 999일 수 있습니다. 지수 또는 엔지니어링 표기법이 다음 명령어로 사용될 것인지 여부를 제어할 수 있습니다.



지수 표기법은 기본값입니다.

```
/* after the instruction */
Numeric form scientific

123.45 * 1e11    ->    1.2345E+13

/* after the instruction */
Numeric form engineering

123.45 * 1e11    ->    12.345E+12
```

## 숫자 정보

NUMERIC 옵션의 현재 설정을 판별하려면 기본 제공 함수 DIGITS, FORM 및 FUZZ를 사용하십시오.

이러한 함수는 NUMERIC DIGITS, NUMERIC FORM 및 NUMERIC FUZZ이 현재 설정을 각각 리턴합니다.

## 정수

REXX가 이해하는 숫자 세트 내에서는 정수로 정의되는 서브세트를 구분하는 것이 유용합니다.

REXX에서 정수는 모두 0인 10진수 파트를 가지는(또는 10진수 파트가 없는) 숫자입니다. 또한 NUMERIC DIGITS 지시사항을 통해 설정된 정밀도 내에서 해당 정수 파트를 단순히 순자로 표현할 수 있어야 합니다. REXX는 반올림 후에 지수 표시법에서 더 큰 수로 표현되므로 이를 더 이상 안전하게 정수로 설명하거나 사용할 수 없습니다.

## REXX에 의해 직접 사용되는 숫자

논의된 바와 같이 산술 오퍼레이션의 결과는 NUMERIC DIGITS의 설정에 따라 반올림됩니다(필요한 경우).

마찬가지로 REXX가 직접 숫자(산술 오퍼레이션에 관련된 숫자여야 하는 것은 아님)를 사용하는 경우 동일한 반올림 또한 적용됩니다. 이는 0에 숫자가 추가된 것과 마찬가지입니다.

다음과 같은 경우에 사용되는 숫자는 정수여야 하고 사용할 수 있는 가장 큰 숫자는 999999999입니다.

- 구문 분석 템플릿에서의 위치 패턴(변수 위치 패턴 포함)
- 거듭제곱 연산자의 거듭제곱 값(오른쪽 피연산자)
- DO 명령어에서 *expr* 및 *exprf*의 값
- NUMERIC 명령어에서 DIGITS 또는 FUZZ에 대해 제공된 값
- TRACE 명령어에서 숫자 옵션에 사용되는 임의의 숫자.

## 오류

산술 오퍼레이션에서는 두 가지 유형의 오류가 발생할 수 있습니다.

- 오버플로우 또는 언더플로우

이 오류는 결과가 NUMERIC DIGITS 및 NUMERIC FORM의 현재 설정에 따라 형식화될 때 결과의 지수 파트가 언어 프로세서가 처리할 수 있는 범위를 초과하는 경우에 발생합니다. 언어는 지수 파트(즉, 기본 정밀도로 정확한 정수로 표현할 수 있는 가장 큰 숫자)에 대한 최소 기능을 정의합니다. 기본 정밀도가 9이므로 REXX/CICS는 -999999999 - 999999999 범위에 있는 지수를 지원합니다.

이는 (매우) 큰 지수를 허용하므로 오버플로우 또는 언더플로우가 구문 오류로 처리됩니다.

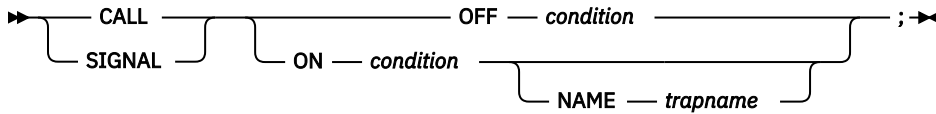
- 충분하지 않은 스토리지

스토리지는 스토리지 부족으로 인해 산술 오퍼레이션이 실패할 수 있는 경우, 계산 및 중간 결과에 필요합니다. 이는 산술 오류가 아니라 일반적으로 종료 오류로 간주됩니다.

## 제 22 장 조건 및 조건 트랩

조건은 CALL ON 또는 SIGNAL ON이 트랩할 수 있는 지정된 이벤트나 상태입니다. 조건 트랩은 REXX 프로그램에서 실행 플로우를 수정할 수 있습니다.

조건 트랩은 SIGNAL 및 CALL 명령어의 ON 또는 OFF 서브키워드를 사용하여 켜거나 끕니다(153 페이지의 『CALL』 and 172 페이지의 『SIGNAL』 참조).



`condition` 및 `trapname`은 상수로 받아들여지는 단일 기호입니다. 이러한 명령어 중 하나를 따라, 조건 트랩은 ON(사용) 또는 OFF(사용 안함)로 설정됩니다. 모든 조건 트랩을 위한 초기 설정은 OFF입니다.

조건 트랩이 사용으로 설정되고 지정된 `condition`이 발생하면, `trapname`을 지정한 경우 제어는 루틴 또는 레이블 `trapname`에 전달합니다. 그렇지 않은 경우, 제어는 루틴 또는 레이블 `condition`에 전달합니다. CALL 또는 SIGNAL은 조건을 위한 최신 트랩이 각각 CALL ON 또는 SIGNAL ON를 사용하여 설정되었는지에 따라 사용됩니다.

**참고:** CALL을 사용하는 경우 `trapname`은 내부 레이블, 기본 제공 함수 또는 외부 루틴일 수 있습니다. SIGNAL을 사용하는 경우 `trapname`은 내부 레이블 전용일 수 있습니다.

트래핑될 수 있는 조건과 해당 이벤트는 다음과 같습니다.

### ERROR

리턴 시 명령이 오류 조건을 표시하는 경우 발생합니다. 명령이 실패를 표시하고 CALL ON FAILURE 또는 SIGNAL ON FAILURE 어느 쪽도 아닌 활성이지 않은 경우에도 발생합니다. 명령을 호출한 절의 마지막에서 조건이 발생하지만 ERROR 조건 트랩이 이미 지연된 상태인 경우 무시됩니다. 조건이 발생되었지만 트랩이 사용(ON) 또는 사용 안함(OFF)로 아직 설정되지 않으면 지연 상태가 조건 트랩의 상태입니다. [노트](#)를 참조하십시오.

CALL ON ERROR 및 SIGNAL ON ERROR는 모든 양수 리턴 코드를 트랩하며 CALL ON FAILURE 및 SIGNAL ON FAILURE이 설정되지 않은 경우에만 음수 리턴 코드를 트랩합니다.

### FAILURE

리턴 시 명령이 실패 조건을 표시하는 경우 발생합니다. 명령을 호출한 절의 마지막에서 조건이 발생하지만 FAILURE 조건 트랩이 이미 지연된 상태인 경우 무시됩니다.

CALL ON FAILURE 및 SIGNAL ON FAILURE는 명령에서 모든 음수 리턴 코드를 트랩합니다.

### HALT

프로그램의 실행을 인터럽트하고 종료하도록 외부 시도가 작성된 경우 발생합니다. 외부 인터럽트 발생 시 처리되는 절의 마지막에서 보통 조건이 발생합니다.

### NOVALUE

초기화되지 않은 변수가 사용된 경우 발생합니다.

- 표현식에서 기간으로서
- PARSE 명령어의 VAR 서브키워드 다음에 오는 `name`로서
- 구문 분석 템플릿, PROCEDURE 명령어 또는 DROP 명령어에서 변수 참조로서.

**참고:** SIGNAL ON NOVALUE는 복합 변수에서 후미를 제외하고 초기화되지 않은 변수를 트랩할 수 있습니다.

```
/* 다음은 NOVALUE를 발생시키지 않습니다. */
signal on novalue
a.=0
say a.z
say 'NOVALUE is not raised.'
exit
```



```
novalue:
say 'NOVALUE is raised.'
```

SIGNAL ON에만 이 조건을 지정할 수 있습니다.

## SYNTAX

프로그램이 실행되는 동안 언어 처리 오류가 감지되면 발생합니다. 이는 숫자가 아닌 항목에 대한 산술 연산 시도와 같이 true 구문 오류 및 "run-time" 오류를 포함한 모든 종류의 오류 처리를 포함합니다. SIGNAL ON에만 이 조건을 지정할 수 있습니다.

조건 트랩에 대한 ON 또는 OFF 참조는 해당 조건 트랩의 이전 상태를 대체합니다(ON, OFF 또는 DELAY 및 *trapname*). 그러므로 CALL ON HALT는 현재 SIGNAL ON HALT(그리고 SIGNAL ON HALT는 현재 CALL ON HALT를 대체)를 대체하며, 새 트랩 이름인 CALL ON 또는 SIGNAL ON은 이전 트랩 이름을 대체하며, OFF 참조는 CALL 또는 SIGNAL 등에 대한 트랩을 사용 안함으로 설정합니다.

**참고:** 각 조건 트랩의 상태(ON, OFF 또는 DELAY, *trapname*)는 서브루틴에 대한 항목에 저장된 다음 RETURN 시 복원됩니다. 이는 CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF가 호출자에서 설정된 조건에 영향을 주지 않고 서브루틴에서 사용될 수 있음을 의미합니다. 서브루틴 호출 중 저장된 다른 정보의 세부사항은 CALL 명령어(CALL)을 참조하십시오.

## 관련 참조

234 페이지의 『조건이 트랩되는 경우 조치』

조건 트랩이 현재 사용되고(ON) 지정된 조건이 발생하는 경우 일반적인 제어 플로우 대신 CALL *trapname* 또는 SIGNAL *trapname* 명령어가 자동으로 처리됩니다.

## 조건이 트래핑되지 않은 경우 조치

조건 트랩이 현재 사용 안함(OFF)으로 설정되어 지정된 조건이 발생한 경우, 기본 조치가 조건에 따라 다릅니다.

- HALT 및 SYNTAX의 경우, 프로그램의 처리가 종료되며 일반적으로 발생한 이벤트의 네이처를 설명하는 메시지(391 페이지의 『제 31 장 오류 번호 및 메시지』 참조)가 조건을 표시합니다.
- 다른 모든 조건의 경우, 조건이 무시되고 해당 조건이 OFF로 남습니다.

## 조건이 트랩되는 경우 조치

조건 트랩이 현재 사용되고(ON) 지정된 조건이 발생하는 경우 일반적인 제어 플로우 대신 CALL *trapname* 또는 SIGNAL *trapname* 명령어가 자동으로 처리됩니다.

CALL ON 또는 SIGNAL ON 명령어의 NAME 하위 키워드 뒤에 *trapname*을 지정할 수 있습니다. *trapname*을 지정하지 않으면 조건 자체의 이름(ERROR, FAILURE, HALT, NOTREADY, NOVALUE, 또는 SYNTAX)이 사용됩니다.

예를 들어, 명령어 call on error는 ERROR 조건에 대해 조건 트랩을 사용하게 할 수 있습니다. 조건이 발생한 경우 오류 ERROR에 의해 식별되는 루틴에 대한 호출이 발생합니다. 명령어 call on error name commanderror는 트랩을 사용하게 하고 조건이 발생한 경우에는 루틴 COMMANDERROR를 트랩 및 호출할 수 있습니다.

조건이 트랩된 후 이벤트의 순서는 SIGNAL 또는 CALL이 처리될지에 따라 다릅니다.

- 수행 조치가 SIGNAL인 경우 현재 명령어의 실행이 즉시 중지되고 조건은 사용 안함으로 설정되며(OFF로 설정됨) SIGNAL은 평소와 정확히 같은 방식으로 수행됩니다(SIGNAL 참조).

조건이 새로운 발생이 트랩될 경우 레이블에 도달하면 다시 사용으로 설정하기 위해 이 조건에 대한 새 CALL ON 또는 SIGNAL ON 명령어가 필요합니다. 예를 들어, SYNTAX가 발생할 때 SIGNAL ON SYNTAX가 사용으로 설정된 다음 SIGNAL ON SYNTAX 레이블 이름을 찾을 수 없는 경우 일반적인 구문 오류 종료가 발생합니다.

- 수행 조치가 CALL(절 경계에서만 발생할 수 있음)인 경우 CALL은 호출이 특수 변수 RESULT에 영향을 미치지 않는 점을 제외하고 일반적인 방식(CALL 참조)으로 수행됩니다. 루틴이 임의의 데이터를 리턴해야 하는 경우 리턴되는 문자열은 무시됩니다.

이러한 조건(ERROR, FAILURE 및 HALT)은 INTERPRET 명령어의 실행 중에 발생할 수 있으므로 INTERPRET의 실행이 인터럽트될 수 있으며 CALL ON이 사용된 경우에는 나중에 재개될 수 있습니다.



조건이 발생하고 CALL이 수행되기 전에 조건 트랩은 지연된 상태로 들어갑니다. 이 상태는 CALL에서 RETURN까지 또는 명시적 CALL(또는 SIGNAL) ON(또는 OFF)가 조건에 대해 수행될 때까지 지속됩니다. 이 지연된 상태는 조건 트랩을 처리하기 위해 호출되는 루틴 시작 시 너무 이른 조건 트랩을 방지합니다. 조건 트랩이 지연된 상태인 경우 사용 가능한 상태로 유지되지만 조건이 다시 발생하면 조건 트랩이 무시되거나(ERROR, FAILURE 또는 NOTREADY의 경우) 다른 조건의 경우 조치(조건 정보의 업데이트 등)는 다음 이벤트 중 하나가 발생할 때까지 지연됩니다.

1. 지연된 조건에 대해 CALL ON 또는 SIGNAL ON이 처리됩니다. 이 경우 CALL 또는 SIGNAL은 새 CALL ON 또는 SIGNAL ON 명령어가 처리된 후 즉시 발생합니다.
2. 지연된 조건에 대해 CALL OFF 또는 SIGNAL OFF가 처리됩니다. 이 경우 조건 트랩이 사용 안함으로 설정되고 조건에 대한 기본 조치가 CALL OFF 또는 SIGNAL OFF 명령어의 끝에 발생합니다.
3. 서브루틴에서 RETURN이 작성됩니다. 이 경우 조건 트랩은 더 이상 지연되지 않고 서브루틴이 즉시 다시 호출됩니다.

CALL로부터의 RETURN 시 원래 실행 플로우가 재개됩니다(즉, 플로는 CALL의 영향을 받지 않음).

#### 참고:

1. 구문 트랩 루틴을 작성할 때에는 더욱 주의해야 합니다. 가능한 경우 루틴은 프로그램의 시작 근처에 넣으십시오. 누락된 인코딩 주석과 같은 특정 스캐닝 오류가 있는 경우 트랩 루틴 레이블을 찾지 못할 수 있으므로 이 조치가 필요합니다. 또한 트랩 루틴은 오류 상태의 프로그램에서 더 많은 부분이 스캔되도록 할 수도 있는 어떠한 명령문도 포함해서는 안됩니다. 이것의 예제는 이름 주변에 따옴표가 없는 기본 제공 함수에 대한 호출입니다. 기본 제공 함수 이름이 대문자이고 따옴표로 묶여 있는 경우 REXX는 내부 레이블을 검색하는 대신 직접 함수로 이동합니다.
2. 모든 경우에 조건은 발견 즉시 발생합니다. SIGNAL ON이 조건을 트랩하면 필요한 경우 현재 명령어가 종료됩니다. 따라서 도중에 이벤트가 발생하는 명령어는 부분만 처리될 수 있습니다. 예를 들어, 지정의 표현식 평가 중에 SYNTAX가 발생하는 경우 지정은 발생하지 않습니다. ERROR, FAILURE, HALT 및 NOTREADY 트랩에 대한 CALL은 절 경계에서만 발생할 수 있음을 참고하십시오. 이러한 조건이 INTERPRET 명령어 중간에 발생하는 경우 INTERPRET의 실행이 인터럽트되어 나중에 재개될 수 있습니다. 마찬가지로 다른 명령어(예: DO 또는 SELECT)는 절 경계에서 CALL에 의해 임시로 인터럽트될 수 있습니다.
3. 각 조건 트랩의 상태(ON, OFF 또는 DELAY 및 임의의 *trapname*)는 서브루틴에 대한 입력 시 저장되어 RETURN 시 복원됩니다. 이는 호출자가 설정한 조건에 영향을 미치지 않고도 CALL ON, CALL OFF, SIGNAL ON 및 SIGNAL OFF를 서브루틴에서 사용할 수 있음을 의미합니다. 서브루틴 호출 중 저장되는 기타 정보에 대한 세부사항은 CALL 명령어(CALL)를 참조하십시오.
4. 조건 트랩의 상태는 외부 루틴이 REXX 프로그램인 경우에도 CALL에 의해 외부 루틴이 호출될 때 영향을 받지 않습니다. REXX 프로그램에 입력 시 모든 조건 트랩의 초기 설정은 OFF입니다.
5. 대화식 추적 중 사용자 입력이 처리되는 동안 모든 조건 트랩은 임시로 OFF 설정됩니다. 이는 예상치 못한 제어 전송(예: SIGNAL ON NOVALUE가 활성화된 동안 사용자가 초기화되지 않은 변수를 실수로 사용하는 경우). 같은 이유로 대화식 추적 중 구문 오류는 프로그램을 종료시키지는 않지만 특별히 트랩된 후 메시지가 지정된 후에 무시됩니다.
6. 시스템 인터페이스는 프로그램 실행이 시작되기 전 또는 프로그램이 종료된 후에 특정 실행 오류를 발견합니다. SIGNAL ON SYNTAX는 이러한 오류를 트랩하지 않습니다.

레이블은 뒤에 콜론이 오는 단일 기호로 구성됨을 참고하십시오. 연속되는 절은 수에 관계없이 레이블일 수 있으며 다중 레이블은 다른 유형의 절 앞에 허용됩니다.

#### 관련 정보

##### 조건 및 조건 트랩

## 조건 정보

조건이 트랩되어 SIGNAL 또는 CALL이 발생하는 경우 이 조건이 트랩된 조건이 되고 이와 연관된 특정 조건 정보가 레코드됩니다.

CONDITION 기본 제공 함수를 사용하여 이 정보를 검사할 수 있습니다(186 페이지의 『CONDITION』 참조).

이 조건 정보에는 다음이 포함됩니다.

- 현재 트랩된 조건의 이름

- 조건 트랩(CALL 또는 SIGNAL)의 결과로 처리되는 명령어의 이름
- 트랩된 조건의 상태
- 해당 조건과 연관된 설명 문자열.

조건 트랩(CALL ON 또는 SIGNAL ON)의 결과로 제어가 레이블에 전달되면 현재 조건 정보가 대체됩니다. 조건 정보는 저장되고 CALL ON 트랩으로 인한 호출을 포함하여 함수 호출 또는 서브루틴 전체에서 복원됩니다. 따라서 CALL ON에 의해 호출되는 루틴은 적절한 조건 정보에 액세스할 수 있습니다. 모든 조건 정보는 루틴이 리턴된 후에도 계속 사용 가능합니다.

### 설명 문자열

설명 문자열은 트랩된 조건에 따라 다릅니다.

#### ERROR

처리되어 결과로 오류 조건을 발생시킨 문자열입니다.

#### FAILURE

처리되어 실패 조건을 발생시킨 문자열입니다.

#### HALT

정지 요청과 연관된 문자열입니다. 문자열이 제공되지 않은 경우 이는 널 문자열일 수 있습니다.

#### NOVALUE

시도된 참조가 NOVALUE 조건을 발생시킨 변수의 파생된 이름입니다. NOVALUE 조건 트랩은 SIGNAL ON을 사용해서만 사용으로 설정할 수 있습니다.

#### SYNTAX

언어 프로세서가 오류와 연관시킨 문자열입니다. 특정 문자열을 제공하지 않은 경우 이는 널 문자열일 수 있습니다. 처리 중인 오류의 특성 및 위치에 대한 정보를 특수 변수 RC와 SIGL이 제공함을 참고하십시오. SIGNAL ON을 사용해서만 SYNTAX 조건 트랩을 사용으로 설정할 수 있습니다.

## 특수 변수

특수 변수는 REXX 프로그램의 처리 중에 자동으로 설정할 수 있는 변수입니다.

세 가지 특수 변수가 있으며 이들은 RC, RESULT 및 SIGL입니다. 이러한 특수 변수는 모두 초기값을 가지지 않지만 프로그램에서 이를 변경할 수 있습니다. (RESULT에 대한 정보는 [171 페이지의 『RETURN』](#)의 내용을 참조하십시오.)

#### RC

ERROR 및 FAILURE의 경우 REXX 특수 변수 RC는 제어가 조건 레이블로 전송되기 전에 평소대로 명령 리턴 코드로 설정됩니다.

SIGNAL ON SYNTAX의 경우 RC는 구문 오류 번호로 설정됩니다.

#### SIGL

CALL 또는 SIGNAL로 인한 제어의 전송에 따라 제어의 전송을 발생시키는 절의 프로그램 행 번호는 특수 변수 SIGNAL에 저장됩니다. 조건 트랩으로 인한 제어 전송의 경우 SIGL에 지정된 행 번호는 CALL 또는 SIGNAL이 발생하기 전에 처리된(현재 서브루틴 레벨에서) 마지막 절의 행 번호입니다. 이는 오류의 행 번호를 사용할 수 있을 때(예: 텍스트 편집기를 제어하기 위해) SIGNAL ON SYNTAX에 대해 특히 유용합니다. 일반적으로 SYNTAX 레이블 뒤에 오는 코드는 데이터의 소스를 찾기 위해 소스를 구문 분석한 다음 편집기를 호출하여 오류의 행에 위치한 소스 파일을 편집할 수 있습니다. 이 경우 편집기에서 작성된 모든 변경사항이 적용되게 하려면 먼저 프로그램을 다시 실행해야 할 수도 있음을 참고하십시오.

또는 다음 예제에서와 같이 SIGL은 오류(예: 함수 호출의 간헐적 실패)의 원인을 판별하는 데 도움을 주기 위해 사용될 수 있습니다.

```
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
```

이 코드는 먼저 오류 코드, 행 번호 및 오류 메시지를 표시합니다. 그런 다음 오류의 행을 표시하고 마지막으로 디버그 모드가 되어 오류의 행에서 사용된 변수의 값을 검사할 수 있도록 합니다.



## 제 23 장 REXX/CICS 텍스트 편집기

REXX/CICS는 일반 용도의 CICS 기반 문서 편집기를 제공합니다.

exec 및 데이터가 CICS 환경 내에서 작성되고, 업데이트되고, 보도록 편집기가 제공됩니다.

REXX/CICS 편집기는 여러 접두부 명령을 포함합니다(예: C, CC, M, MM, B, A, F, P). 편집기를 사용하려면 EDIT를 입력하십시오.

편집기 세션 중, 명령행에 입력된 명령은 각 명령 사이에 ";"를 두어 연결될 수 있습니다. REXX로 작성된 매크로에 대해서도 지원이 제공됩니다. 프로파일 exec로 편집기 설정을 사용자 정의하거나 새 명령을 편집기에 추가하거나 또는 애플리케이션의 일부로 편집기 기능을 사용할 수 있습니다.

**참고:** REXX/CICS 문서 편집기는 2진 파일을 지원하지 않습니다.

### REXX/CICS 문서 편집기 호출

REXX/CICS 편집 세션을 CICS 터미널(지우기 화면), REXX/CICS 세션, 다른 편집 세션 또는 REXX exec에서 시작할 수 있습니다. CICS EDIT 전송 ID(또는 REXX/CICS 편집기를 위한 사이트 정의된 전송 ID)를 실행하여 편집기를 입력할 수 있습니다. 명확한 REXX/CICS 화면을 위해 EDIT 다음에 RFS(REXX File System) 파일 ID를 입력하면 편집 세션이 이 파일에 대해 실행됩니다.

**참고:** exec 이름 없이 CICS 트랜잭션 ID로서 REXX를 지정하는 경우, IBM 제공 REXXTRY 대화식 유틸리티(CICRTRY exec)는 실행됩니다. REXXTRY는 REXX 명령문과 명령을 수행하기 위한 대화형 셸을 제공합니다.

편집기에 있는 동안, EDIT *fileid*를 입력하여 다른 편집 세션을 시작할 수 있습니다. 이 호출 메소드에 대한 구문은 EDIT 명령, 248 페이지의 『EDIT』에 정의됩니다. exec을 쓸 때, 명령으로서 CICS EDIT 트랜잭션 ID를 실행하여 REXX/CICS 편집기 세션을 시작할 수 있습니다. 편집기 호출 시 파일 ID가 지정되지 않은 경우, 파일 ID의 기본값은 현재 RFS 디렉토리의 파일 NONAME입니다. 완전한 파일 ID는 poolid:\dir1\dir2\fn.ft입니다.

그러나 대상 파일이(또는 작성될) 현재 디렉토리(CD 명령으로 지정된)에 있다면 부분 파일 ID를 지정할 수 있습니다.

다음 예제는 편집단계를 시작하기 위한 4가지 다른 방법을 표시합니다.

```
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)
```

첫 번째 예는 REXX/CICS로부터 세션을 시작합니다. 두 번째는 기존 세션으로부터 새 세션을 시작합니다. 세 번째는 REXX exec로부터 세션을 시작합니다. 네 번째는 REXX/CICS REXXTRY 유틸리티로부터 세션을 시작합니다.

### 화면 형식

프로파일 없이 편집기를 호출하면 기본 화면 정의가 다음 그림에 표시된대로 표시됩니다.

```

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=RG T F12=QUI

첫 번째 행은 다음을 포함하는 제목 행에 대해 예약됩니다.

- 완전한 파일 ID
- 열 번호
- 표시된 메시지.

파일의 맨 위 및 맨 아래인 위치를 알게 해주는 정보용 행이 새 편집 세션을 시작할 때 표시되는 유일한 행입니다. 이 행은 접두부 영역 및 데이터 영역으로 구성됩니다. 이 예에서 **\*\*\* TOP OF DATA \*\*\***가 표시되는 행은 현재 행입니다. 강조표시되어 나머지에서 구별되며 명령이 적용되는 대부분의 명령 행입니다. 명령행은 상위 레벨을 기반으로 편집기와 상호작용하게 합니다. 제목 행을 제외하고 모든 화면 행은 편의상 이동될 수 있습니다.

## 접두부 명령

편집기는 여러 접두부 명령을 제공합니다. 이러한 명령은 접두부 영역에 입력되고, 개별 또는 연속적 블록을 기반으로 행을 복사, 이동, 삽입, 삭제 및 복제하는 기능을 제공합니다.

- 개별 행 명령.

다음 명령은 개별 행과 작업하며 단일 문자로 구성됩니다.

/	파일에서 현재 행 지정
I	행 삽입
D	행 삭제
C	행 복사
M	행 이동
R	행 복제
"	복제의 동의어

행의 접두부 영역에서 이전의 명령 중 하나를 입력하면 명령은 해당 행에서 개별 함수를 수행합니다. 행 00000의 접두부 영역에 "I"를 입력한 경우, 편집기는 입력을 위해 바로 아래에 새 행을 엽니다. 접두부 명령의 끝에 수를 추가할 수도 있습니다. 복제 요소로 작동합니다. 숫자 "5"가 "I"에 추가되면, 5행이 1행 대신 입력을 위해 열립니다.

- 연속 블록 명령.

다음 명령은 행의 연속 블록으로 작업하며 두 개의 문자로 구성됩니다.

**DD**

행 블록 삭제

**CC**

행 블록 복사

**MM**

행 블록 이동

**RR**

행 블록 복제

**""**

복제의 동의어

블록 명령은 쌍으로 처리됩니다. 블록의 첫 번째 행의 접두부 영역에 명령 하나를 두고 블록의 마지막 행의 접두부 영역에 동일한 명령을 둡니다. 예를 들어, 행의 블록을 삭제하려면 "DD"를 00001행의 접두부 영역에 두고 다른 "DD"를 00005행의 접두부 영역에 둡니다. 00001행과 00005행 사이를 포함하여 모든 행을 삭제합니다. 복제 요소를 허용하는 블록 접두부 명령만이 복제 명령입니다. 그러므로 복제 블록 명령으로 수를 지정하는 경우 개별 행이 아닌 블록 복제의 수를 알아야 합니다. 복제 요소는 첫 번째 RR 복제 명령으로 지정되어야 합니다.

- 대상 명령.

다음 명령은 호출된 대상 접두부 명령입니다.

**A**

이후

**B**

이전

**F**

다음

**P**

이전

이 명령은 이동 및 복사 접두부 명령을 텍스트 블록을 위한 대상에 지정합니다. 접두부 영역에 이를 입력하면, 복사 또는 이동으로부터의 텍스트가 대상 명령이 지정된 내용에 따라 해당 행 앞뒤에 배치됩니다. 복제 접두부 명령은 대상 접두부 명령을 사용하지 않습니다. 대신에, 복제될 블록 바로 다음에 해당 출력을 위치시킵니다.

## REXX/CICS 편집기 아래에 있는 매크로

편집기는 REXX 매크로를 지원하여, 매크로에 편집기 설정을 변경하고 편집기 화면을 표시하는 기능을 제공합니다. 매크로는 모든 편집기 명령행 명령을 처리할 수 있습니다.

다음 예는 편집기 명령 환경을 다루고, 편집기 설정을 변경합니다.

```
/* 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'
```

다음 예는 Some, More 및 DATA를 3개의 별도 행에 입력한 다음 현재 편집 화면을 표시합니다.

```
/* Macro to use the REXX/CICS editor as an I/O interface */
ADDRESS EDITSVR
'INPUT Some'
'INPUT More'
'INPUT DATA'
'DISPLAY'
```

## 명령행 명령

각 명령행 명령에 대한 구문 및 설명에 대해 설명합니다.

### ARBCHAR

ARBCHAR는 임의 문자를 설정합니다.

➤ ARBCHAR — *arbchar* ➤

#### 피연산자

##### *arbchar*

인쇄할 수 있는(입력 가능) 문자를 지정합니다.

#### 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

#### 예

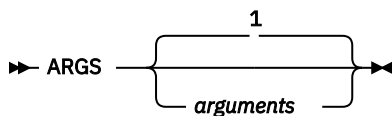
```
'ARBCHAR . '
```

이 예제는 문자 "."를 임의 문자로 정의합니다.

**참고:** 임의 문자는 문자열에서 텍스트의 위치를 차지합니다. 임의 문자의 기본 값은 "."입니다. 임의 문자를 사용하여 검색을 수행하는 방법에 대한 자세한 정보는 FIND 명령 [250 페이지의 『FIND』](#)의 내용을 참조하십시오.

### ARGS

ARGS는 텍스트 편집기 EXEC 명령을 사용하여 호출되면 편집되는 프로그램에 전달될 기본 매개변수를 저장합니다.



참고:

<sup>1</sup> *arguments*가 지정되지 않은 경우 이전에 정의된 인수가 삭제됩니다.

#### 피연산자

##### 인수

전달될 매개변수 문자열을 지정합니다. *arguments*를 지정하지 않으면 이전에 정의된 인수가 삭제됩니다.

#### 리턴 코드

0

일반 리턴

#### 예

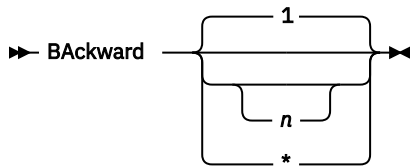
```
'ARGS A B C '  
'EXEC '  
'ARGS '
```



이 예제의 첫 번째 행은 A, B 및 C로 전달될 인수를 정의합니다. 두 번째 행은 현재 편집 중인 파일의 마지막으로 저장된 사본을 실행하여 이를 행 1에서 정의된 인수로 전달합니다. 마지막 행은 인수를 삭제합니다.

## BACKWARD

BACKWARD는 지정된 수의 화면 디스플레이에 대해 파일의 시작 부분을 향하여 뒤로 화면이동합니다.



### 피연산자

***n***

뒤로 화면이동할 화면 디스플레이의 수를 지정합니다. 별표(\*)를 지정하는 경우 화면이 파일의 맨 위로 화면 이동하고 현재 행이 파일의 맨 위로 설정됩니다. *n*을 지정하지 않으면 화면이 한 디스플레이에서 뒤로 화면이 동합니다.

### 리턴 코드

**0**

일반 리턴

**202**

올바르지 않은 피연산자

### 예

```
' BACKWARD '
```

이 예제는 파일의 맨 위를 향하여 한 화면을 화면이동합니다.

**참고:** 편집기는 기본적으로 BACKWARD에 대해 PF7을 설정하고 FORWARD에 대해 PF8을 설정합니다.

## BOTTOM

BOTTOM은 파일의 맨 아래로 화면이동합니다.

➡ BOTTOM ➡

### 리턴 코드

**0**

일반 리턴

### 예

```
' BOTTOM '
```

이 예제는 파일의 맨 아래로 화면이동합니다.

## CANCEL

CANCEL은 변경사항을 저장하지 않고 현재 편집 세션을 종료합니다.

➡ CANCEL ➡

## 리턴 코드

0

일반 리턴

210

요청이 실패함

## 예

```
'CANCEL '
```

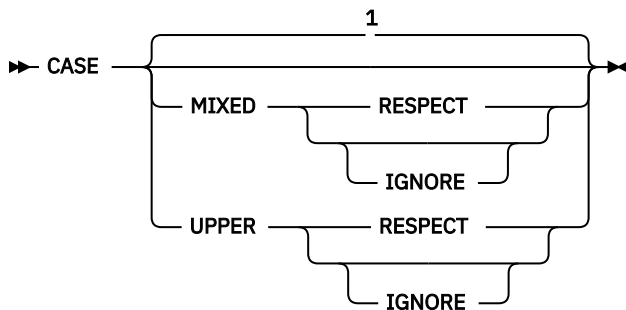
이 예제는 파일 변경사항을 저장하지 않은 채로 현재 편집기 세션을 무조건 종료합니다.

## 참고:

1. CANCEL을 사용하면 변경사항을 저장하지 않고 변경이 작성되었음을 알리는 경고 메시지 없이 편집기를 종료할 수 있습니다.
2. CANCEL은 QQUIT에 대한 동의어입니다.

## CASE

CASE는 대소문자 변환과 해석 환경 설정을 설정합니다.



## 참고:

- 1 기본값은 사용자 프로파일에 설정됩니다.

## 피연산자

### UPPER

입력 시 소문자를 대문자로 변환합니다.

### MIXED

원래 형식으로 각 문자와 함께 작동합니다.

### RESPECT

검색을 하는 동안 각 문자의 대소문자를 사용합니다.

### IGNORE

검색을 하는 동안 각 문자의 대소문자를 무시합니다.

## 리턴 코드

0

정상 요청

202

올바르지 않은 피연산자

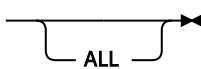
## 예

```
'CASE MIXED RESPECT '
```

이 예는 대소문자를 MIXED로 설정하고 민감성을 RESPECT로 설정합니다. 민감성에 관한 자세한 정보는 FIND 명령, 250 페이지의 『FIND』의 내용을 참조하십시오.

## CHANGE

CHANGE는 파일의 문자열을 변경합니다.

➤ CHANGE — /string1/string2/ 

### 피연산자

#### string1

대체할 문자열을 지정합니다.

#### string2

string1을 대체하는 문자열을 지정합니다.

#### 모두

현재 행부터 파일 끝까지 모든 행의 모든 발생이 변경됨을 나타내는 키워드입니다.

#### /

구분 문자입니다.

### 리턴 코드

#### 0

일반 리턴

#### 202

올바르지 않은 피연산자

#### 210

요청이 실패함

#### 223

검색 인수를 찾을 수 없음

### 예

```
'CHANGE /noeditor/editor/'
```

이 예제는 noeditor의 첫 번째 발생을 editor로 대체합니다.

**참고:** CHANGE 명령은 string1을 찾을 때 및 CASE 명령이나 시스템 기본값에 정의된 대로 string2를 변경할 때 편집기의 감도 설정을 준수합니다.

## CMDLINE

CMDLINE은 명령행 디스플레이 환경 설정을 설정합니다.

➤ CMDLINE  TOP

### 피연산자

#### TOP

화면의 두 번째 행에 명령행을 표시합니다.

#### BOTTOM

화면의 맨 아래 행에 명령행을 표시합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

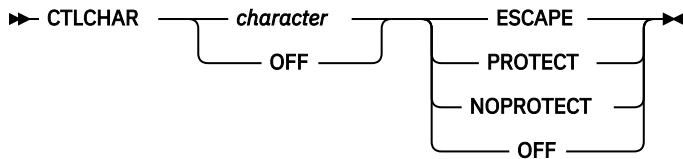
## 예

```
'CMDLINE TOP'
```

이 예제는 명령행을 화면의 두 번째 행에 배치합니다.

## CTLCHAR

CTLCHAR은 제어 문자의 기능을 설정합니다.



## 피연산자

### 문자

사용할 제어 문자를 지정합니다.

### OFF

문자를 지정하지 않은 상태로 OFF가 사용되는 경우 제어 문자의 모든 정의를 삭제하고 문자가 지정된 경우에는 특정 문자를 삭제합니다.

### ESCAPE

RESERVED 명령에 전달되는 문자열에서 다음 문자가 제어 문자가 되도록 지정합니다.

### PROTECT

RESERVED 명령에 전달된 문자열이 사용자 입력으로부터 보호되도록 지정합니다.

### NOPROTECT

RESERVED 명령에 전달되는 사용자 입력이 이 문자열에서 허용되도록 지정합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

## 예

```
'CTLCHAR ! ESCAPE'  
'CTLCHAR % PROTECT'  
'RESERVED 20 HIGH !% Important Info'
```

이 예제는 !를 이스케이프 문자로 정의하고 %를 필드 보호 문자로 정의합니다. 이러한 명령을 입력하고 나면 화면 행 20이 보호되고 제어 문자 !% 뒤에 오는 텍스트를 포함합니다.

## CURLINE

CURLINE은 현재 행 디스플레이 환경 설정을 설정합니다.

➡ CURLINE — *number* ➡

## 피연산자

### *number*

화면 행 번호를 지정합니다.

## 리턴 코드

### 0

일반 리턴

### 202

올바르지 않은 피연산자

## 예

```
'CURLINE 3'
```

이 예제는 현재 디스플레이 행을 화면 행 3으로 설정합니다.

**참고:** 현재 행이 이 명령에 지정된 화면 행 번호에 표시됩니다. 그러나 행 1은 제목 행으로 예약되므로 현재 행은 행 1에 표시될 수 없습니다.

## DISPLAY

DISPLAY는 현재 편집 화면을 표시합니다.

➡ DISPLAY ➡

## 리턴 코드

### 0

일반 리턴

### 210

요청이 실패함

## 예

```
'DISPLAY'
```

이 예제는 현재 편집 화면을 표시합니다.

**참고:** DISPLAY 명령은 매크로에서 실행되는 경우에만 유용합니다. 이 명령은 현재 편집 세션의 화면을 표시합니다. 일반 터미널 편집 세션에서 실행되는 경우 눈에 띄는 효과는 없습니다.

## DOWN

DOWN은 파일에서 앞으로 화면이동합니다.

➡ DOWN — { 1  
                  number } ➡

## 피연산자

### *number*

화면이동할 행의 수를 지정합니다. *number*를 지정하지 않으면 화면이 한 행만 아래로 이동합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

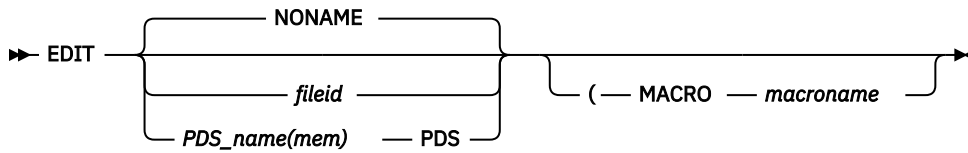
## 예

```
'DOWN 5'
```

이 예제는 파일의 5개 행 앞으로 화면이동합니다.

## EDIT

EDIT은 새 편집 세션을 엽니다.



## 피연산자

### *fileid*

작성되거나 편집될 파일의 파일 ID를 지정합니다.

### *PDS\_name(mem)*

편집될 완전한 MVS PDS 이름이나 멤버 이름을 지정합니다.

### **PDS**

PDS가 편집되는 경우 PDS 멤버 이름 다음에 오는 키워드입니다.

### **MACRO**

편집 중인 파일에 적용되는 명령어 그룹을 지정하는 키워드입니다.

### *macroname*

프로파일 매크로 파일 ID(REXX exec 이름)의 파일 이름 부분을 지정합니다.

## 리턴 코드

0

정상 리턴

203

파일이 없음

204

권한이 부여되지 않음

211

올바르지 않은 파일 ID

226

파일이 현재 편집 중임

299

내부 오류

1748

태스크 입출력 테이블(TIOT)에서 ddname의 항목이 없음

1749

다중 장치 데이터 세트로 내보내기할 수 없음

## 1750

연결에서 ddname에는 둘 이상의 데이터 세트가 있음

### 예

```
'EDIT TEST.EXEC'
```

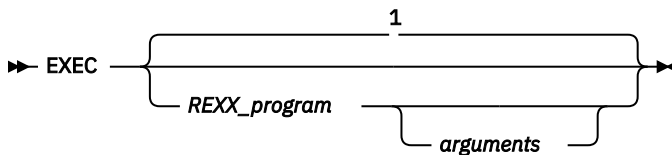
이 예는 파일 TEST.EXEC에 대해 편집 세션을 열어줍니다.

### 참고:

- 파일 편집 시 사용된 디렉토리는 다음과 같이 판별됩니다.
  - 완전한 디렉토리 ID가 명시적으로 주어지면, 항상 사용됩니다.
  - 부분적으로 규정된 파일 ID가 지정된 경우, 현재 디렉토리나 경로는 파일 ID와 일치하는 기존 파일을 찾기 위한 시도로 검색됩니다(해당 디렉토리를 사용하여 완전히 해결된 경우).
    - 이러한 일치가 성공하는 경우, 기존 파일에 대한 편집 세션 및 파일 ID는 파일이 있는 디렉토리를 사용하여 완전히 해결됩니다.
    - 파일이 검색 순서에 없는 경우, 새 파일에 대한 편집 세션이 현재 작업 디렉토리를 지정하는 전체 분석된 파일 ID로 작성됩니다(CD 명령으로 지정된 것처럼).
- MVS PDS 구성원은 구성원 이름 다음의 키워드 PDS를 사용하여 구별됩니다. 구성원은 괄호로 묶이며, 따옴표가 허용되지 않습니다.
- 편집기가 호출을 시도하는 기본 사용자 프로파일 매크로는 CICEPROF입니다. CICEPROF 매크로는 환경과 같이 ISPF/PDF를 작성합니다. CICXPROF라는 두 번째 프로파일 매크로가 제공됩니다. CICXPROF는 환경과 같이 VM/CMS XEDIT를 작성합니다.
- 파일 ID나 PDS 이름이 지정되지 않은 경우 특수 이름, NONAME이 있는 RFS 파일이 작성됩니다.

## EXEC

EXEC은 편집기 세션 내에서 REXX 프로그램을 실행합니다.



### 참고:

<sup>1</sup> REXX\_program이 지정되지 않으면 현재 편집하고 있는 파일의 마지막으로 저장된 사본이 실행됩니다.

### 피연산자

#### REXX\_program

실행할 REXX 프로그램 이름을 지정합니다. 이름을 지정하지 않으면 현재 편집하고 있는 파일의 마지막으로 저장된 사본이 실행됩니다. ARGS 명령을 사용하여 인수가 정의된 경우 이러한 인수는 전달됩니다.

#### 인수

REXX\_program 피연산자에 지정된 프로그램에 전달될 인수를 지정합니다.

### 리턴 코드

**n**

호출된 exec의 종료로 설정된 리턴 코드입니다. [160 페이지의 『EXIT』](#)의 내용을 참조하십시오.

**0**

일반 리턴

**-3**

Exec을 찾을 수 없음

**-10**

Exec 이름이 지정되지 않음

**-11**

올바르지 않은 exec 이름

**-12**

GETMAIN 오류

**-99**

내부 오류

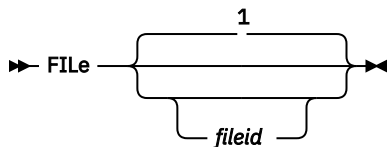
**예**

```
'EXEC TEST1.EXEC X Y Z'
```

이 예제는 프로그램 TEST1.EXEC을 실행하고 인수로 X, Y 및 Z를 전달합니다.

## FILE

FILE은 편집 중인 현재 파일을 저장합니다.



참고:

<sup>1</sup> *fileid*를 지정하지 않으면 파일이 기본 파일 ID로 저장됩니다.

## 피연산자

***fileid***

파일의 파일 ID를 지정합니다. *fileid*를 지정하지 않으면 파일이 기본 파일 ID로 저장됩니다.

## 리턴 코드

**0**

일반 리턴

**202**

올바르지 않은 피연산자

**204**

권한이 부여되지 않음

**207**

파일 풀에 공간이 충분하지 않음

**210**

요청이 실패함

**예**

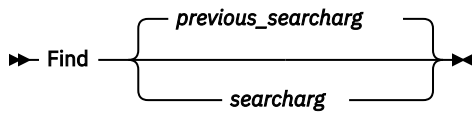
```
'FILE'
```

이 예제는 편집 세션의 현재 파일 ID 스펙을 사용하여 편집 중인 현재 파일을 저장합니다. 현재 파일 ID는 처음에 편집 명령에 지정된 파일 ID에서 가져옵니다.

## FIND

FIND는 파일에서 텍스트의 문자열을 찾습니다.





## 피연산자

### **searcharg**

검색될 텍스트 문자열을 지정합니다. *searcharg*를 지정하지 않은 경우, 검색은 이전 검색 문자열 (*previous\_searcharg*)에서 실행됩니다.

## 리턴 코드

**0**

정상 리턴

**202**

올바르지 않은 피연산자

**223**

검색 인수가 없음

## 예

```
'FIND REDT '
```

이 예는 REDT의 첫 번째 항목을 찾습니다.

```
'FIND Redt '
```

CASE가 RESPECT로 설정되면 예는 REDT의 첫 번째 항목을 찾지 않습니다. Redt의 첫 번째 항목을 찾습니다. 자세한 정보는 CASE 명령, 244 페이지의 『CASE』를 참조하십시오.

*searcharg*는 임의의 문자를 포함할 수 있으며, 그런 경우에 임의의 문자가 임의의 문자의 위치에 삽입될 수 있는 텍스트 문자열을 나타냅니다.

```
'FIND ONE.THREE '
```

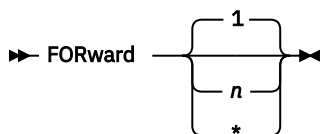
이 예는 다른 문자열과 결합된 ONE 및 THREE로 시작되는 첫 번째 항목을 찾습니다.

## 참고:

1. RESPECT 플래그가 CASE 명령으로 설정되면 *searcharg*의 대소문자가 사용됩니다.
2. 검색은 현재 행에서 시작되어 BOTTOM OF DATA에 도달할 때까지 아래로 계속되거나 일치가 작성됩니다. BOTTOM OF DATA가 일치 없이 도달하면, 현재 행은 FIND가 처리되기 전에 있었던 행에 남으며 BOTTOM OF DATA를 현재 행으로 작성하지 않습니다.

## FORWARD

FORWARD는 지정된 화면 디스플레이의 수만큼 파일의 끝을 향하여 앞으로 화면이동합니다.



## 피연산자

*n*

앞으로 화면이동할 화면 디스플레이의 수를 지정합니다. 별표(\*)를 지정하는 경우 화면이 파일의 맨 아래로 화면이동하고 현재 행이 데이터의 마지막 행으로 설정됩니다. *n*을 지정하지 않으면 화면이 한 디스플레이에서 앞으로 화면이동합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

## 예

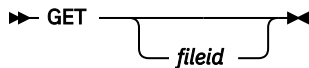
```
'FORWARD'
```

이 예제는 파일의 끝 방향을 향하여 한 화면을 화면이동합니다.

참고: 편집기는 기본적으로 BACKWARD에 대해 PF7을 설정하고 FORWARD에 대해 PF8을 설정합니다.

## GET

GET은 RFS 파일을 현재 편집 세션으로 가져옵니다.



## 피연산자

*fileid*

파일의 파일 ID를 지정합니다.

## 리턴 코드

0

일반 리턴

203

파일을 찾을 수 없음

204

권한이 부여되지 않음

210

요청이 실패함

## 예

```
'GET POOL1:\USERS\USER1\TEST.EXEC'
```

이 예제는 현재 행 뒤에 REXX 파일 시스템 파일 TEST.EXEC을 가져옵니다.

## GETPDS

GETPDS는 MVS PDS의 멤버를 현재 편집 세션으로 가져옵니다. 이 파일은 현재 행 뒤에 삽입됩니다.

➡ GETPDS — *PDS\_name (mem)* ➡

## 피연산자

### ***PDS\_name(mem)***

완전한 MVS PDS 및 멤버 이름을 지정합니다.

## 리턴 코드

**0**

일반 리턴

**203**

파일을 찾을 수 없음

**204**

권한이 부여되지 않음

**210**

요청이 실패함

## 예

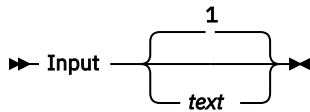
```
'GETLIB MYFILE.PROJ1(MEM1)'
```

이 예제는 PDS 멤버 MEM1을 가져오고 이를 편집 세션의 현재 행 뒤에 넣습니다.

**참고:** CICS 리전에 데이터 세트에 대한 외부 보안 관리자 액세스 권한이 없고 CICS 사인온이 상위 레벨 데이터 세트 접두부와 일치하지 않는 경우 데이터 세트의 멤버를 가져오는 데 GETPDS를 사용할 수 없습니다.

## INPUT

INPUT은 현재 행 뒤에 새 행을 삽입합니다.



참고:

<sup>1</sup> *text*를 지정하지 않으면 새 행은 공백입니다.

## 피연산자

### ***text***

새 행에 삽입할 텍스트를 지정합니다. *text*를 지정하지 않으면 줄 바꾸기는 공백입니다.

## 리턴 코드

**0**

일반 리턴

## 예

```
'INPUT Test Input Data'
```

이 예제는 Test Input Data 텍스트를 현재 행 뒤에 새로 삽입된 행에 배치합니다.

## JOIN

JOIN은 두 행을 하나의 행으로 결합합니다.

➡ JOIN ➡

### 리턴 코드

**0**  
정상 리턴

**210**  
요청 실패

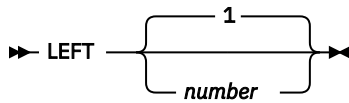
### 예

```
' JOIN '
```

이 예는 커서가 있는 행을 바로 뒤의 행과 결합합니다.

## LEFT

LEFT는 파일 왼쪽으로 화면이동합니다.



### 피연산자

**number**  
화면이동할 문자의 수를 지정합니다. *number*를 지정하지 않으면 화면이 파일에서 왼쪽으로 1자 화면이동합니다. *number*에 대해 0을 지정하면 파일이 왼쪽 끝으로 화면이동합니다.

### 리턴 코드

**0**  
일반 리턴

**202**  
올바르지 않은 피연산자

**210**  
요청이 실패함

### 예

```
' LEFT 20 '
```

이 예제는 왼쪽으로 20자 화면이동합니다.

## LINEADD

LINEADD는 커서 행 뒤에 빈 행을 추가합니다.

➡ LINEADD ➡

### 리턴 코드

**0**  
일반 리턴

**230**  
커서가 파일 영역에 없음

### 예

```
' PFKEY 2 LINEADD '
```

이 예제의 결과로 PF2를 누를 때마다 커서가 있는 행(파일 행인 경우) 뒤에 빈 행이 추가됩니다.

**참고:** LINEADD는 프로그램 기능(PF) 키에 지정되는 경우에 주로 유용합니다. 기본적으로 PF2에 지정됩니다.

## LPREFIX

LPREFIX는 현재 행 접두부 영역에 접두부 명령을 입력합니다.

►► LPREFIX — *prefix* ◀◀

### 피연산자

#### *prefix*

편집 세션 중에 입력되는 표준 접두부(예: C, CC, M, MM, B, A)를 지정합니다.

### 리턴 코드

**0**

일반 리턴

**202**

올바르지 않은 피연산자

### 예

```
'LPREFIX D'
```

이 예제의 결과로 현재 파일 행이 삭제됩니다.

**참고:** LPREFIX는 편집 매크로 내에서 접두부 명령을 사용할 수 있도록 하기 위해 제공됩니다.

## MACRO

MACRO는 매크로를 호출합니다.

►► MACRO — *fileid* ◀◀

### 피연산자

#### *fileid*

실행할 매크로의 파일 ID를 지정합니다. 이 파일 ID에 파일 유형 접미부가 포함된 경우 해당 접미부를 사용하여 exec을 호출하려는 시도가 이루어집니다. 그렇지 않은 경우에는 접미부가 EXEC인 exec을 호출하려는 시도가 이루어집니다.

### 리턴 코드

***n***

호출된 exec의 종료로 설정되는 리턴 코드입니다. [160 페이지의 『EXIT』](#)의 내용을 참조하십시오.

**0**

일반 리턴

**-3**

Exec을 찾을 수 없음

**-10**

Exec 이름이 지정되지 않음

**-11**

올바르지 않은 exec 이름

**-12**

GETMAIN 오류

## 예

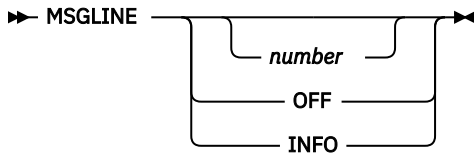
```
'MACRO P00L1:\USERS\USER1\TEST'
```

이 예제는 매크로 P00L1:\USERS\USER1\TEST.EXEC을 호출합니다.

**참고:** 매크로는 REXX/CICS 편집기 서버에 대한 호출을 수행할 수 있습니다. 편집기의 명령행에서 입력할 수 있는 명령은 매크로에서 실행될 수 있습니다.

## MSGLINE

MSGLINE은 메시지 행 디스플레이 환경 설정을 설정합니다.



### 피연산자

#### *number*

해당하는 화면 행에 메시지 행을 표시합니다.

#### OFF

메시지 행을 표시하지 않습니다.

#### INFO

헤더 행에 메시지를 표시합니다.

### 리턴 코드

#### 0

일반 리턴

#### 202

올바르지 않은 피연산자

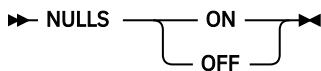
## 예

```
'MSGLINE 2'
```

이 예제는 메시지 행을 화면 행 2에 배치합니다.

## NULLS

NULLS는 화면의 필드가 후미 공백 또는 후미 널과 함께 작성되는지 여부를 제어합니다.



### 피연산자

#### ON

화면의 필드가 후미 널과 함께 작성되도록 지정합니다.

#### OFF

화면의 필드가 후미 공백과 함께 작성되도록 지정합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

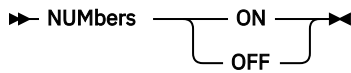
## 예

```
'NULLS ON'
```

이 예제의 결과로 화면의 필드에 후미 널이 표시됩니다.

## NUMBERS

NUMBERS는 접두부 영역 디스플레이 환경 설정을 설정합니다.



## 피연산자

ON

순차 번호를 접두부 영역에서 표시합니다.

OFF

등호를 접두부 영역에서 표시합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

## 예

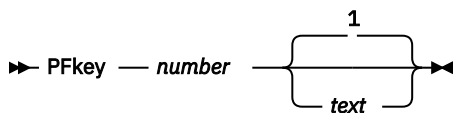
```
'NUMBERS ON'
```

이 예제는 접두부 영역의 순차 번호를 표시합니다.

**참고:** 행 번호 순서 지정은 편집 세션 내 데이터에 대해 수행되지 않지만 편집 세션 중에만 파일 행과 연관되는 의사 행 수입니다.

## PFKEY

PFKEY는 프로그램 기능(PF) 키를 설정하거나 처리합니다.



참고:

<sup>1</sup> *text*가 지정되지 않은 경우 PF 키가 처리됩니다.

## 피연산자

*number*

설정되거나 처리되는 PF 키를 지정합니다.

### **text**

PF 키가 설정되는 텍스트를 지정합니다. *text*를 지정하지 않은 경우 PF 키가 처리됩니다.

### **리턴 코드**

**0**

정상 리턴

**202**

올바르지 않은 피연산자

**229**

범위 밖의 숫자

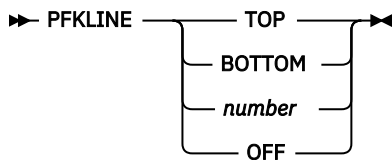
### **예**

```
'PFKEY 3 quit'  
'PFKEY 3'
```

이 예제는 먼저 PFKEY 3를 quit으로 설정한 다음 PF 키를 처리합니다.

## **PFKLINE**

PFKLINE은 프로그램 함수(PF) 키 행 표시장치 환경 설정을 설정합니다.



### **피연산자**

#### **TOP**

화면의 두 번째 행에 PF 키 행을 표시합니다.

#### **BOTTOM**

화면의 맨 아래 행에 PF 키 행을 표시합니다.

#### **number**

화면 행 번호를 지정합니다.

#### **OFF**

표시장치 화면에서 PF 키를 제거합니다.

### **리턴 코드**

**0**

일반 리턴

**202**

올바르지 않은 피연산자

### **예**

```
'PFKLINE BOTTOM'
```

이 예제는 PF 키 행을 화면의 맨 아래 행에 배치합니다.

## **QQUIT**

QQUIT은 변경사항을 저장하지 않고 현재 편집 세션을 종료합니다.



➡ QQuit ➡

## 리턴 코드

0

일반 리턴

## 예

```
'QQUIT'
```

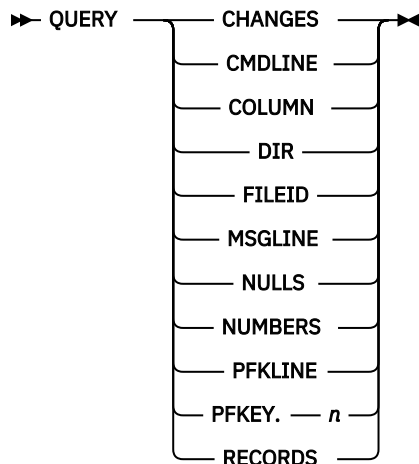
이 예제는 파일 변경사항을 저장하지 않은 채로 현재 편집기 세션을 무조건 종료합니다.

### 참고:

1. QQUIT을 사용하면 변경사항을 저장하지 않고 변경이 작성되었음을 알리는 경고 메시지 없이 편집기를 종료할 수 있습니다.
2. QQUIT에 대한 동의어는 CANCEL입니다.

## QUERY

QUERY는 편집기의 현재 설정을 표시합니다.



### 피연산자

#### CHANGES

마지막으로 저장된 이후 파일에 대한 수정의 수를 표시합니다.

#### CMDLINE

명령행의 현재 설정을 표시합니다. 자세한 정보는 문서 편집기 명령 [245 페이지](#)의 『CMDLINE』의 내용을 참조하십시오.

#### COLUMN

화면에 표시되는 파일의 시작 열을 표시합니다.

#### DIR

파일과 연관된 디렉토리를 표시합니다.

#### FILEID

편집 중인 파일의 이름을 표시합니다.

#### MSGLINE

메시지 행의 현재 설정을 표시합니다. 자세한 정보는 문서 편집기 명령 [256 페이지](#)의 『MSGLINE』의 내용을 참조하십시오.

## NULLS

NULLS의 현재 설정을 표시합니다. 자세한 정보는 문서 편집기 명령 [256 페이지의 『NULLS』](#)의 내용을 참조하십시오.

## NUMBERS

NUMBERS의 현재 설정을 표시합니다. 자세한 정보는 문서 편집기 명령 [257 페이지의 『NUMBERS』](#)의 내용을 참조하십시오.

## PFKLINE

PFKLINE의 현재 설정을 표시합니다. 자세한 정보는 문서 편집기 명령 [258 페이지의 『PFKLINE』](#)의 내용을 참조하십시오.

## PFKEY.*n*

PFKEY *n*을 누르면 처리되는 명령을 표시합니다. *n*은 1부터 24까지의 숫자입니다.

## RECORDS

파일에 있는 행의 수를 표시합니다.

### 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

236

정의되지 않음

### 예

```
'QUERY PFKEY.1'
```

이 예제는 PFKEY 1을 누르면 처리되는 명령을 표시합니다.

## QUIT

QUIT은 현재 편집 세션을 종료합니다.

➡ QUIT ➡

### 리턴 코드

0

일반 리턴

210

요청이 실패함

### 예

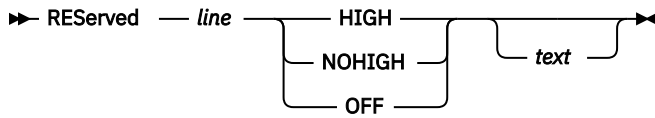
```
'QUIT'
```

이 예제는 편집기를 종료합니다.

**참고:** 현재 파일이 변경된 경우 저장을 수행하거나 QQUIT 명령을 입력할 때까지 편집기를 종료할 수 없습니다.

## RESERVED

RESERVED는 출력을 위해 화면의 한 행을 예약합니다.



## 피연산자

### line

예약되는 행을 지정하고 텍스트가 표시됩니다.

### HIGH

텍스트가 강조표시되도록 지정하는 키워드입니다.

### NOHIGH

텍스트가 보통 강도이도록 지정하는 키워드입니다.

### OFF

예약된 상태에서 행이 해제되도록 지정하는 키워드입니다.

### text

예약된 행에 표시되는 텍스트 문자열을 선택적 제어 문자와 함께 지정합니다.

## 리턴 코드

### 0

일반 리턴

### 202

올바르지 않은 피연산자

### 210

요청이 실패함

### 229

숫자가 범위 밖에 있음

## 예

```
'CTLCHAR ! ESCAPE'
'CTLCHAR % PROTECT'
'RESERVED 20 HIGH !% Important Info'
```

이 예제는 높은 강도로 Important Info를 표시하고 보호 필드를 화면 행 20에 표시합니다.

## RESET

RESET은 보류 접두부 명령을 종료합니다.

```
➡ RESET ➡
```

## 리턴 코드

### 0

일반 리턴

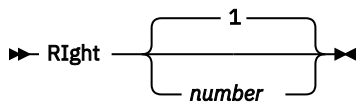
## 예

```
'RESET'
```

이 예제는 보류 접두부 명령을 취소합니다.

## RIGHT

RIGHT은 파일에서 오른쪽으로 화면이동합니다.



## 피연산자

### *number*

화면이동할 문자의 수를 지정합니다. *number*를 지정하지 않으면 화면이 파일에서 오른쪽으로 1자 이동합니다. *number*에 대해 0을 지정하면 파일이 오른쪽 끝으로 화면이동합니다.

## 리턴 코드

**0**

일반 리턴

**202**

올바르지 않은 피연산자

## 예

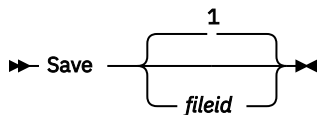
```
'RIGHT 20'
```

이 예제는 오른쪽으로 20자 화면이동합니다.

**참고:** 지정한 값으로 인해 대상이 레코드 외부에 있게 되는 경우 화면 이동이 레코드의 오른쪽에서 중지됩니다.

## SAVE

SAVE는 파일을 RFS 파일 또는 PDS 멤버에 저장합니다.



참고:

<sup>1</sup> *fileid*를 지정하지 않으면 파일이 기본 파일 ID로 저장됩니다.

## 피연산자

### *fileid*

파일의 파일 ID를 지정합니다. *fileid*를 지정하지 않으면 파일이 기본 파일 ID로 저장됩니다.

## 리턴 코드

**0**

일반 리턴

**202**

올바르지 않은 피연산자

**207**

파일 풀에 공간이 충분하지 않음

**210**

요청이 실패함

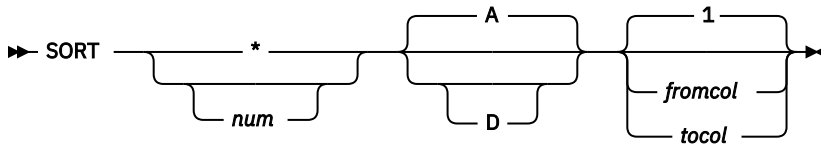
## 예

```
'SAVE SYSTEM:\USERS\USER1\TEST.EXEC'
```

이 예제는 현재 파일을 RFS에 저장하고 이름을 SYSTEM:\USERS\USER1\TEST.EXEC으로 지정합니다.

## SORT

SORT는 현재 행부터 아래로 행을 정렬합니다.



### 피연산자

**\***

현재 행부터 파일 끝까지 모든 행이 정렬되도록 지정합니다.

**num**

num 값에 대해 현재 행부터 행이 정렬되도록 지정합니다.

**A**

행이 오름차순으로 정렬되도록 지정합니다. (기본값입니다.)

**D**

행이 내림차순으로 정렬되도록 지정합니다.

**fromcol**

행이 이 열에서 시작되는 데이터에 대해 정렬되도록 지정합니다. *fromcol tocol*을 지정하지 않으면 정렬이 첫 번째 열에서 시작됩니다.

**tocol**

행이 이 열에서 종료되는 데이터에 대해 정렬되도록 지정합니다.

### 리턴 코드

**0**

일반 리턴

**202**

올바르지 않은 피연산자

**229**

숫자가 범위 밖에 있음

### 예

```
'SORT * A 5 10'
```

이 예제는 파일의 모든 행을 현재 행부터 아래로 정렬하고 5에서 10까지 열에 대해 정렬됩니다.

**참고:** 많은 수의 행을 정렬하는 경우 정렬이 매우 느리게 진행됩니다.

## SPLIT

SPLIT은 한 개의 행을 두 개의 행으로 분할합니다.

➡ SPLIT ⇐

### 리턴 코드

**0**

일반 리턴

**210**

요청이 실패함

## 예

```
'SPLIT'
```

이 예제는 커서가 있는 행을 두 개의 행으로 분할합니다. 한 행은 커서 왼쪽 해당 행의 모든 텍스트를 포함하고 다음 행은 나머지 텍스트(커서의 아래 및 오른쪽)를 포함합니다.

## STRIP

STRIP은 모든 파일 행에서 후미 공백을 제거합니다.

►► STRIP ◄◄

### 리턴 코드

0

일반 리턴

## 예

```
'STRIP'
```

이 예제는 각 파일 행의 모든 후미 공백을 제거합니다.

**참고:** 이 명령은 파티션된 데이터 세트에서 데이터 세트를 가져온 다음 이를 RFS에 저장할 때 특히 유용합니다. 파티션된 데이터 세트는 공백으로 채워지기 때문입니다.

## SYNONYM

SYNONYM은 기타 올바른 명령에 명령 조치를 지정합니다.

►► SYNONYM — *syn* ————— ◄◄  
                                  *command*

### 피연산자

***syn***

동어인 명령 조치를 실행하는 올바른 명령을 지정합니다.

***command***

올바른 명령을 지정합니다.

### 리턴 코드

0

일반 리턴

## 예

```
'SYNONYM GL GETLIB'
```

이 예제는 GL을 명령 GETLIB와 동일하게 만듭니다.

## TOP

TOP은 파일의 맨 위로 화면이동합니다.

►► TOP ◄◄

## 리턴 코드

0

일반 리턴

## 예

```
'TOP'
```

이 예제는 파일의 맨 위로 화면이동합니다.

## TRUNC

TRUNC는 파일의 각 행을 지정된 길이로 자릅니다.

➡ TRUNC — *column* →

## 피연산자

*column*

보존할 마지막 열을 지정합니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

## 예

```
'TRUNC 72'
```

이 예제는 파일의 모든 행을 72자 길이로 자릅니다.

**참고:** 이 명령은 제거해야 하는 순서 번호가 있는 데이터 세트에 대해 작업할 때 유용합니다. 편집기에는 현재 파일에 순서 번호를 배치하거나 유지보수하는 것에 대한 지원이 없습니다.

## UP

UP은 파일에서 뒤로 화면이동합니다(파일의 맨 위 방향으로).

➡ UP — { 1  
          *number* } →

## 피연산자

*number*

화면이동할 행의 수를 지정합니다. *number*를 지정하지 않으면 기본 화면이동 양이 1로 설정됩니다.

## 리턴 코드

0

일반 리턴

202

올바르지 않은 피연산자

예

```
'UP 20'
```

이 예제는 파일에서 뒤로 20행을 화면이동합니다.



## 제 24 장 REXX/CICS 파일 시스템

REXX 파일 시스템(RFS)은 REXX 파일 시스템의 외부에서 가져온 RFS 명령과 데이터를 사용하여 REXX/CICS 편집기로, 그리고 exec에 의해 작성된 텍스트 파일과 exec의 스토리지에 대해 제공됩니다.

RFS는 Advanced Interactive Executive(AIX)와 OS/2 파일 시스템 뒤에 모델링되었습니다. 디렉토리의 개념은 이러한 환경으로부터 가져온 주제입니다. 각 디렉토리를 여러 서브디렉토리로 파티션하는 것은 계층 구조를 제공합니다.

RFS 명령을 사용하여 RFS 함수에 액세스할 수 있습니다. 파일 I/O를 수행할 수 있는 능력을 제공하는 것뿐만 아니라, RFS 명령은 사용자에게 본질적 파일 시스템 유지보수를 하는 기능을 제공합니다.

파일 리스트 유틸리티(FLST)는 REXX 파일 시스템에 전체 화면 인터페이스로서 제공됩니다. 이 유틸리티는 다른 CICS 작업이 수행될 수 있는 플랫폼으로서 사용될 수도 있습니다.

### 파일 풀, 디렉토리 및 파일

파일 풀은 VSAM 파일의 세트입니다. 각 파일 풀에는 루트 디렉토리가 있습니다. 파일은 데이터 파일이거나 REXX exec입니다.

풀의 첫 번째 VSAM 파일은 풀에 대한 정보를 포함하고 데이터를 풀에 기록하려면 먼저 형식화되어야 합니다. 풀에 있는 다른 모든 VSAM 파일은 첫 번째 파일의 확장기능이고 Access Method Services 및 CICS에 정의 및 할당된 후에 VSAM 파일의 풀 목록에 정의되기만 하면 됩니다. VSAM 파일은 파일 풀에 추가되어 해당 풀에 추가 스토리지 공간을 자유롭게 제공할 수 있습니다. 그러나 파일 풀에 공간을 정의하고 추가하는 것은 REXX/CICS 관리자 또는 CICS 시스템 프로그래머의 TASK입니다. 파일 풀에는 1 - 8자의 고유한 이름이 지정됩니다.

파일 풀의 루트 디렉토리 이름은 `file_pool_name:`입니다. 이 디렉토리는 서브디렉토리뿐만 아니라 파일을 포함할 수 있습니다. 서브디렉토리는 다른 디렉토리 안에 있는 디렉토리입니다. 서브디렉토리는 다른 디렉토리 안에 작성할 수 있습니다. 새 디렉토리는 RFS MKDIR 명령을 사용하여 작성할 수 있습니다. 루트 디렉토리를 제외한 모든 디렉토리는 결합된 1 - 8자의 디렉토리 파일 이름, 기간, 1 - 8자 선택적 디렉토리 파일 유형에 의해 구분됩니다. 이를 디렉토리 ID라고 합니다.

파일 풀은 사용자 또는 비사용자 파일 풀에 정의될 수 있습니다. 모든 사용자 파일 풀에 있어야 하는 디렉토리는 USERS 디렉토리입니다. 이 디렉토리는 시스템의 사용자에게 해당하는 여러 서브디렉토리를 포함합니다. 파일을 처음 RFS에 저장하는 경우 USERS 디렉토리에서 새 서브디렉토리가 작성됩니다. CICS에 사인온한 경우 이 새 디렉토리는 사용자 ID로 이름 지정됩니다. 그렇지 않으면 디렉토리 이름은 CICS DFLTUSER 값으로 기본 설정됩니다. 이 디렉토리를 작성하고 나면 해당 개인 디렉토리 안에 원하는 수만큼 서브디렉토리를 작성할 수 있습니다. 작성한 디렉토리 중 어느 하나에 파일을 배치할 수 있습니다.

파일은 데이터 파일이거나 REXX exec입니다. 파일은 디렉토리(파일 이름과 선택적 파일 유형이 기간을 결합함) 동일한 이름 지정 규칙을 사용합니다. REXX exec의 기본 파일 유형은 EXEC입니다. 파일은 CICS 편집기 또는 RFS DISK 명령으로 작성됩니다.

완전한 파일 ID는 뒤에 `:`가 오는 파일 풀 이름, `\`이 뒤에 오는 경로의 각 디렉토리 ID 및 파일 이름 및 선택적 파일 유형으로 구성되는 파일 ID로 구성됩니다.

### 예제

다음 예제는 완전한 파일 ID를 보여줍니다. POOL1은 파일 풀 이름이고 USERS 및 USER1은 디렉토리 ID이고 TEST.EXEC은 파일 ID입니다.

```
POOL1:\USERS\USER1\TEST.EXEC
```

다음 예제는 파일 풀, 디렉토리 및 파일을 보여줍니다.

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

이 예제는 두 개의 파일 풀(POOL1 및 WORK)을 보여줍니다. 파일 풀 POOL1은 파일(TEST1.EXEC)과 두 개의 서브디렉토리(USERS와 PROJECT1)를 포함합니다. USERS 서브디렉토리 안에는 사용자 ID(USER1과 USER2)에 해당하는 두 개의 서브디렉토리(USER1과 USER2)가 있습니다. 사용자 USER1에게는 그 디렉토리 안에 파일(TEST2.EXEC) 및 서브디렉토리(DOCS)가 있습니다. DOCS 서브디렉토리 안에는 다른 파일(TEST3.DOCUMENT)이 있습니다. 사용자 USER2에게는 그 디렉토리 안에 파일(LETTER.DOCUMENT)이 있습니다. 파일 풀 WORK은 파일 (TEST1.DATA)과 서브디렉토리(CHARTS)를 포함합니다. 서브디렉토리 CHARTS 안에는 두 개의 파일(CHART1.DATA 및 CHART2.DATA)이 있습니다.

## 현재 디렉토리 및 경로

현재 디렉토리는 현재 작업 디렉토리이고 RFS(REXX File System)에서 작업할 때 검색 순서에서 첫 번째입니다.

현재 디렉토리는 CD 명령 333 페이지의 『CD』를 사용하여 설정될 수 있습니다. CD 명령은 DOS와 같은 운영 체제의 cd 명령과 비슷한 형식을 가집니다. 구문은 뒤에 부분 또는 완전한 디렉토리 이름이 오는 CD입니다. 서브 디렉토리에서 상위 디렉토리로 다시 변경하려면 CD ..를 입력하십시오. 다른 서브디렉토리로 변경하기 위해서는 CD 뒤에 서브디렉토리 이름을 둘 수 있습니다.

### 예제

다음 예제에서 첫 번째 명령은 현재 디렉토리를 POOL1:\USERS\USER1으로 설정하고 두 번째 명령은 현재 디렉토리를 POOL1:\USERS\USER1\DOCS로 설정합니다. 세 번째 명령은 현재 디렉토리를 다시 POOL1:\USERS\USER1로 변경합니다.

```
'CD POOL1:\USERS\USER1'
'CD DOCS'
'CD ..'
```

PATH 명령은 현재 디렉토리가 검색된 후에 REXX exec에 대한 검색 순서를 정의하는 데 사용됩니다. 자세한 정보는 375 페이지의 『PATH』의 내용을 참조하십시오. 구문은 PATH, 그 다음에는 완전한 디렉토리 이름의 목록(공백으로 구분된)입니다.

다음 예제는 먼저 현재 디렉토리를 설정하고 검색 순서를 정의합니다.

```
'CD POOL1:\USERS\USER1\EXECS'
'PATH POOL1:\ POOL1:\USERS\USER1'
'EXEC TEST2.EXEC'
```

exec 이름은 완전하며 각 디렉토리의 검색이 수행되기 전에 검색에서 각 디렉토리의 디렉토리 ID를 사용합니다. 완전한 이름은 다음과 같습니다.

```
'POOL1:\USERS\USER1\EXECS\TEST2.EXEC'
'POOL1:\TEST2.EXEC'
'POOL1:\USERS\USER1\TEST2.EXEC'
```

REXX/CICS 명령 EXEC을 호출하면 위의 세 디렉토리 모두 검색되어 결과적으로 REXX/CICS가 POOL1:\USERS\USER1 디렉토리에서 exec를 찾게 됩니다. TEST2.EXEC가 POOL1:\ 디렉토리에 있었다면 RFS는 이를 발견했을 때 검색을 중지했을 것입니다. 검색 순서에서 찾은 첫 번째 사본에 액세스합니다.

**참고:** 파일 이름이 완전하지 않을 때마다 RFS는 exec를 검색하는 검색 순서에 따라 현재 디렉토리부터 시작합니다. 찾은 첫 번째 사본이 실행됩니다. 아무 것도 찾지 못하면 대상 파일 또는 exec를 찾을 수 없음을 표시하는 오류가 리턴됩니다.

## 보안

REXX/CICS 파일 시스템 보안에는 파일 액세스 보안과 명령 실행 보안의 두 가지 일반 유형이 있습니다.

- 파일 액세스 보안은 `exec` 및 데이터에 대한 액세스를 제어합니다. RFS 파일 보안은 두 레벨 즉, CICS 레벨과 RFS 디렉토리 레벨에서 제어될 수 있습니다.
  - CICS 레벨에서 파일 풀 VSAM 파일에 액세스하기 위한 권한 부여가 특정 사용자에게 제공될 수 있습니다. 이는 상위 레벨의 보안을 제공합니다.
  - RFS 디렉토리 레벨에서 사용자 디렉토리는 개인용 디렉토리이고 (기본적으로) 소유한 사용자에게 의해서만 액세스될 수 있습니다.

그러나 디렉토리의 소유자는 RFS AUTH 명령을 사용하여 디렉토리를 `publicr`, `publicw` 또는 `secured`로 정의할 수 있습니다. `publicr`은 다른 REXX/CICS 사용자에게 이 디렉토리에 대한 읽기/전용 액세스 권한이 있음을 의미합니다. `publicw`는 다른 REXX/CICS 사용자에게 이 디렉토리에 대한 읽기/쓰기 액세스 권한이 있음을 의미합니다. `secured`는 액세스를 허용해야 하는지 여부를 판별하기 위해 RFS 보안 엑시트가 호출됨을 의미합니다. 자세한 정보는 RFS AUTH 명령 269 페이지의 『AUTH』의 내용을 참조하십시오. 비사용자 디렉토리를 작성할 수 있으며 권한 부여된 사용자가 해당 액세스 레벨을 정의할 수 있습니다.

- 명령 실행 보안은 특정 REXX/CICS 명령 또는 명령 키워드의 사용을 제어합니다. 자세한 정보는 [보안](#)의 내용을 참조하십시오.

## RFS 명령

RFS 명령 환경에서는 RFS와 인터페이스하기 위한 명령을 발행합니다. 명령 환경을 RFS로 설정하는 경우 RFS 명령 앞에 RFS를 지정해서는 안 됩니다.

### 예

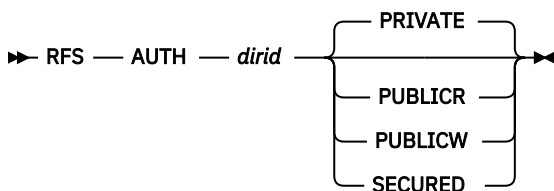
```
'RFS DISKR POOL1:\USERS\USER1\TEST.EXEC DATA.'
```

이 예제는 RFS 파일 TEST.EXEC의 콘텐츠를 REXX 복합 변수 데이터로 읽습니다. TEST.EXEC은 완전한 디렉토리 POOL1:\USERS\USER1\에 있습니다.

RFS 명령에 대한 구문이 뒤에 옵니다.

## AUTH

AUTH는 RFS 디렉토리에 대한 액세스 권한을 부여합니다.



### 피연산자

#### *dirid*

REXX 파일 시스템 디렉토리 ID를 지정합니다. 이는 부분 또는 완전한 ID입니다. 자세한 정보는 CD 명령, 333 페이지의 『CD』의 내용을 참조하십시오.

#### **PRIVATE**

디렉토리의 소유자만이 이 파일에 대한 읽기/쓰기 권한을 가짐을 지정합니다. 이는 기본값입니다.

#### **PUBLICR**

모든 사용자에게 해당 디렉토리의 파일에 대한 읽기 전용 액세스 권한이 있음을 지정합니다.

#### **PUBLICW**

모든 사용자에게 해당 디렉토리의 파일에 읽기/쓰기 액세스 권한이 있음을 지정합니다.

## SECURED

외부 보안 관리자에게 해당 디렉토리의 파일에 대한 액세스 권한이 부여됨을 지정합니다.

### 리턴 코드

RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

### 예

```
'RFS AUTH POOL1:\USERS\USER1\DOCS PUBLICR'
```

이 예는 디렉토리 DOCS에게 공용 디렉토리를 작성합니다. 모든 사용자에게 디렉토리 DOCS에서 파일에 읽기 전용 액세스 권한이 있습니다.

## CKDIR

CKDIR은 기존 RFS 디렉토리 레벨을 확인합니다.

➡ RFS — CKDIR — *dirid* ➡

### 피연산자

#### *dirid*

REXX 파일 시스템 디렉토리 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CD 명령 [333 페이지](#)의 『CD』의 내용을 참조하십시오.

### 리턴 코드

RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

### 예

```
'RFS CKDIR POOL1:\USERS\USER1\DOCS'
```

이 예제는 기존 디렉토리 POOL1:\USERS\USER1에서 DOCS라는 디렉토리를 확인합니다.

## CKFILE

CKFILE은 지정된 부분 또는 완전한 파일 ID가 존재하는지 확인합니다.

➡ RFS — CKFILE — *fileid* ➡

### 피연산자

#### *fileid*

파일 ID를 지정합니다.

### 리턴 코드

RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

### 예

```
'CKFILE POOL1:\USERS\USER1\TEST.EXEC'
```

이 예제는 기존 디렉토리 POOL1:\USERS\USER1에서 TEST.EXEC이라고 하는 파일을 확인합니다.

## COPY

COPY는 파일을 복사합니다.

➡ RFS — COPY — *fileid1* — *fileid2* ➡

### 피연산자

#### *fileid1*

소스 파일 ID를 지정합니다. 완전하거나 부분적인 디렉토리이고 파일 ID일 수 있습니다.

#### *fileid2*

대상 파일 ID를 지정합니다. 완전하거나 부분적인 디렉토리 및 파일 ID일 수 있습니다.

**참고:** 대상 파일(*fileid2*)이 이미 존재하는 경우 *fileid1*의 콘텐츠가 이를 대체합니다.

### 리턴 코드

RFS 명령 377 페이지의 『RFS』의 내용을 참조하십시오.

### 예

```
'RFS COPY POOL1:\USERS\USER1\TEST1.EXEC POOL1:\USERS\USER1\TEST2.EXEC'
```

이 예제는 TEST1.EXEC을 POOL1:\USERS\USER1 디렉토리 내에 있는 TEST2.EXEC에 복사합니다.

## DELETE

DELETE는 RFS 파일을 삭제합니다.

➡ RFS — DELETE — *fileid* ➡

### 피연산자

#### *fileid*

삭제할 파일의 이름을 지정합니다. 이 이름은 부분적이거나 완전한 이름일 수 있습니다. 자세한 정보는 CD 명령 333 페이지의 『CD』의 내용을 참조하십시오.

### 리턴 코드

RFS 명령 377 페이지의 『RFS』의 내용을 참조하십시오.

### 예

```
'RFS DELETE POOL1:\USERS\USER1\TEST1.EXEC'
```

이 예제는 디렉토리 POOL1:\USERS\USER1 안에 있는 파일 TEST1.EXEC을 삭제합니다.

## DISKR

DISKR은 RFS 파일에서 레코드를 읽습니다.

➡ RFS — DISKR — *fileid* — { DATA.  
stem. } ➡

### 피연산자

#### *fileid*

파일 ID를 지정합니다.

#### *stem.*

스탬의 이름을 지정합니다. 스탬은 마침표로 종료되어야 합니다. 144 페이지의 『스탬』의 내용을 참조하십시오. 기본 스탬은 DATA.입니다.

## 리턴 코드

RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

## 예

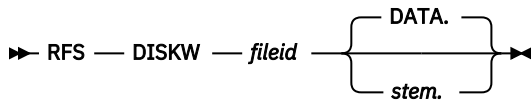
```
'RFS DISKR POOL1:\USERS\USER1\TEST.DATA DATA.'
```

이 예제는 RFS 파일 POOL1:\USERS\USER1\TEST.DATA의 전체 콘텐츠를 DATA. REXX 복합 변수에 저장합니다.

**참고:** DATA.0은 파일에서 읽은 레코드 수로 설정됩니다. DATA.n은 파일에서 읽은 n번째 레코드를 포함합니다.

## DISKW

DISKW는 스템에서 RFS 파일로 레코드를 씁니다. 파일은 스템의 데이터로 오버레이됩니다.



## 피연산자

### *fileid*

파일 ID를 지정합니다.

### *stem.*

스템의 이름을 지정합니다. 스템은 마침표로 종료되어야 합니다. [144 페이지](#)의 『스템』의 내용을 참조하십시오. 기본 스템은 DATA.입니다.

## 리턴 코드

[377 페이지](#)의 『RFS』의 내용을 참조하십시오.

## 예

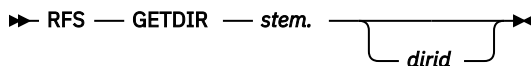
```
'RFS DISKW POOL1:\USERS\USER1\TEST.EXEC DATA.'
```

이 예제는 DATA. REXX 복합 변수의 콘텐츠를 RFS 파일 POOL1:\USERS\USER1\TEST.EXEC에 저장합니다.

**참고:** DATA.0을 파일에 기록될 레코드의 수로 설정하십시오.

## GETDIR

GETDIR은 현재 또는 지정된 디렉토리의 콘텐츠 목록을 지정된 REXX 배열에 리턴합니다.



## 피연산자

### *stem.*

스템의 이름을 지정합니다. 스템은 마침표로 종료되어야 합니다. [144 페이지](#)의 『스템』의 내용을 참조하십시오.

### *dirid*

REXX 파일 시스템 디렉토리 레벨 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CD 명령 [333 페이지](#)의 『CD』의 내용을 참조하십시오.

## 리턴 코드

RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

## 예

```
'RFS GETDIR DIRDOC. POOL1:\USERS\USER1\DOCS'
```

이 예제는 디렉토리 DOCS의 콘텐츠를 DIRDOC. REXX 복합 변수에 배치합니다.

## MKDIR

MKDIR은 새 RFS 디렉토리 레벨을 작성합니다.

►► RFS — MKDIR — *dirid* ►◄

### 피연산자

#### *dirid*

REXX 파일 시스템 디렉토리 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CD 명령 333 페이지의 『CD』의 내용을 참조하십시오.

### 리턴 코드

RFS 명령 377 페이지의 『RFS』의 내용을 참조하십시오.

## 예

```
'RFS MKDIR POOL1:\USERS\USER1\DOCS'
```

이 예제는 기존 디렉토리 POOL1:\USERS\USER1에서 DOCS라는 새 디렉토리를 작성합니다.

**참고:** 권한 부여된 사용자만 해당 \USERS\userid 디렉토리 구조의 외부에서 디렉토리를 작성할 수 있습니다.

## RDIR

RDIR은 지정된 RFS 디렉토리를 제거합니다.

►► RFS — RDIR — *dirid* ►◄

### 피연산자

#### *dirid*

REXX 파일 시스템 디렉토리 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CD 명령 333 페이지의 『CD』의 내용을 참조하십시오.

### 리턴 코드

RFS 명령 377 페이지의 『RFS』의 내용을 참조하십시오.

## 예

```
'RFS RDIR POOL1:\USERS\USER1\DOCS'
```

이 예제는 기존 디렉토리 POOL1:\USERS\USER1에서 DOCS라는 디렉토리를 삭제합니다.

## RENAME

RENAME은 RFS 파일의 이름을 새 이름으로 바꿉니다.

►► RFS — RENAME — *fileid1* — *fileid2* ►◄

## 피연산자

### *fileid1*

소스 파일 ID를 지정합니다. 완전하거나 부분적인 디렉토리이고 파일 ID일 수 있습니다.

### *fileid2*

소스 대상 파일 ID를 지정합니다. 완전하거나 부분적인 디렉토리이고 파일 ID일 수 있습니다.

**참고:** 대상 파일(*fileid2*)이 이미 존재하는 경우 *fileid1*의 콘텐츠가 이를 대체합니다.

## 리턴 코드

RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

## 예

```
'RFS RENAME POOL1:\USERS\USER1\TEST1.EXEC POOL1:\USERS\USER1\TEST2.EXEC'
```

이 예제는 파일 POOL1:\USERS\USER1\TEST1.EXEC의 이름을 POOL1:\USERS\USER1\TEST2.EXEC으로 바꿉니다.

## REXX/CICS 파일 목록 유틸리티

파일 목록 유틸리티(FLST)는 REXX 파일 시스템에 대한 전체 화면 인터페이스를 제공합니다.

실행하면 FLST는 RFS를 관리하거나 exec을 호출하거나 트랜잭션을 시작합니다. REXX, RFS 및 CICS에 대한 상위 레벨 인터페이스로 사용됩니다.

## 호출

FLST를 실행하려면 지워진 CICS 화면 또는 지워진 REXX/CICS 화면으로 이동하여 FLST를 입력하십시오. FLST가 실행을 시작합니다.

FLST 화면 형식은 다음과 같습니다.

```
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
```

사용자 ID는 왼쪽 상단 모서리에 표시됩니다. 현재 디렉토리는 사용자 ID 옆에 표시됩니다. 화면의 나머지는 REXX/CICS 편집기 세션과 매우 비슷하게 표시됩니다. FLST는 편집기에서 RESERVED 명령을 사용하여 표시되는 처음 두 개 행을 제외하고 모든 입출력(I/O)에 대해 편집기를 사용합니다. I/O 셀에 대해 REXX/CICS 편집기 사용의 장점 대형 디렉토리에서 파일 이름 또는 파일 유형을 검색하는 기능 및 디렉토리를 디스크의 파일에 저장하는 기능에서 표시됩니다.



## REXX/CICS 파일 목록 유틸리티 아래 매크로

REXX 파일 목록 유틸리티는 FLST 설정을 변경하고 FLST 화면을 표시할 수 있는 기능을 매크로에 제공하여 REXX 매크로를 지원합니다. 매크로는 FLST 명령 모드를 처리할 수 있습니다.

### 예

다음 예제는 FLST 환경을 처리하고 FLST 설정을 변경합니다.

```
/* Macro to set some FLST settings */  
ADDRESS FLSTSVR  
'SET PFKEY 11 EDIT'  
'SET PFKEY 12 CANCEL'  
'SYNONYM DISCARD RFS DELETE'
```

## FLST 명령

이 섹션은 FLST 명령에 대해 설명합니다. 이러한 명령은 명령행에서 또는 소스 FLST 명령 열 아무 곳이나 입력할 수 있습니다.

**참고:** 데이터를 여러 위치에서 입력하는 경우 프로그램 기능 키, 명령행에 입력된 데이터 및 명령 열에 입력된 데이터 순으로 우선순위를 가집니다.

### CANCEL

CANCEL은 명령 열에서 어떠한 명령도 실행하지 않고 종료합니다.

명령행에서 CANCEL을 입력할 때 다음 구문을 사용하십시오.

➤ CANCEL ➤

### COPY

COPY는 파일을 복사합니다.

FLST 명령 열에 COPY를 입력할 때 다음 구문을 사용하십시오.

➤ COPY — / — *fileid* ➤

### 피연산자

#### *fileid*

결과가 배치되는 파일의 파일 ID를 지정합니다.

**참고:** *fileid*가 이미 존재하는 경우 이 파일 ID는 대체됩니다.

### 예

```
'COPY / TEST3.EXEC'
```

TEST1.EXEC 옆에 있는 명령 열에서 실행되는 이 예제는 TEST1.EXEC과 같은 새 파일 TEST3.EXEC을 작성합니다.

명령행에서 COPY를 입력할 때 다음 구문을 사용하십시오.

➤ COPY — *fileid1* — *fileid2* ➤

### 피연산자

#### *fileid1*

명령이 작동하는 파일의 파일 ID를 지정합니다.

#### *fileid2*

결과가 배치되는 파일의 파일 ID를 지정합니다.

**참고:** *fileid2*가 이미 있는 경우 *fileid1*의 콘텐츠가 이를 대체합니다.

## 예

```
'COPY TEST1.EXEC TEST3.EXEC'
```

명령행에서 실행된 이 예제는 TEST1.EXEC과 같은 새 파일 (TEST3.EXEC)을 작성합니다.

## DELETE

DELETE는 파일을 삭제합니다.

FLST 명령 열에 DELETE를 입력할 때 다음 구문을 사용하십시오.

➡ DELETE ➡

명령행에서 DELETE를 입력할 때 다음 구문을 사용하십시오.

➡ DELETE — *fileid* ➡

## 피연산자

### *fileid*

명령이 작동하는 파일의 파일 ID를 지정합니다.

## 예

```
'DELETE TEST1.EXEC'
```

명령행에서 실행되는 이 예제는 파일 TEST1.EXEC을 삭제합니다.

## DOWN

DOWN은 하나 이상의 행을 화면이동합니다.

명령행에서 DOWN을 입력할 때 다음 구문을 사용하십시오.

➡ DOWN — *n* ➡

## 피연산자

### *n*

아래로 화면이동할 행의 수를 지정합니다.

## 예

```
'DOWN 5'
```

이 예제는 목록에서 5행 앞으로 화면이동합니다.

## END

END는 입력한 모든 명령을 실행한 다음 END가 명령행에 입력되거나 PF 키로 사용되면 종료됩니다.

명령행에서 END를 입력할 때 다음 구문을 사용하십시오.

➡ END ➡

## EXEC

EXEC은 exec을 실행한 다음 종료됩니다.

FLST 명령 열에 EXEC을 입력할 때 다음 구문을 사용하십시오.

➡ EXEC — */ parameter* ➡

***parameter***

인수로 exec에 전달되는 매개변수를 지정합니다.

## 예

```
'EXEC / PARMS'
```

TEST3.EXEC 옆에 있는 명령 열에서 실행되는 이 예제는 `exec TEST3.EXEC`을 실행하고 `PARMS`를 인수로 전달합니다.

명령행에서 EXEC을 입력할 때 다음 구문을 사용하십시오.



## 피연산자

***fileid***

명령이 작동하는 파일의 파일 ID를 지정합니다.

***parameter***

인수로 exec에 전달되는 매개변수를 지정합니다.

## 리턴 코드

$$n$$

호출된 exec의 종료로 설정되는 리턴 코드입니다. 160 페이지의 『EXIT』의 내용을 참조하십시오.

0

## 일반 리턴

**-3**

Exec을 찾을 수 없음

**-10**

Exec 이름이 지정되지 않음

**-11**

올바르지 않은 exec 이름

**-12**

## GETMAIN 오류

-99

## 내부 오류

## 예

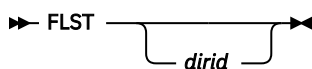
```
'EXEC TEST3 PARMS'
```

이 예제는 `exec TEST3.EXEC`을 실행하고 `PARMS`를 인수로 전달합니다.

**FLST**

FLST는 파일 목록 유틸리티를 호출합니다.

명령행에서 FLST를 입력할 때 다음 구문을 사용하십시오.



## 피연산자

### *dirid*

전체 또는 부분 디렉토리의 ID를 지정합니다. *dirid*을 지정하지 않으면 FLST가 현재 작업 디렉토리로 기본 설정됩니다.

## 예

```
'FLST'
```

이 예제는 현재 작업 디렉토리의 멤버에 대해 파일 목록을 표시합니다.

**참고:** REXX 파일 시스템에 대한 자세한 정보는 [267 페이지의 『제 24 장 REXX/CICS 파일 시스템』](#)의 내용을 참조하십시오.

## MACRO

MACRO는 매크로를 호출합니다.

명령행에서 MACRO를 입력할 때 다음 구문을 사용하십시오.

➤ MACRO — *fileid* ➤

## 피연산자

### *fileid*

실행할 매크로의 파일 ID를 지정합니다. 이 파일 ID에 파일 유형 접미부가 포함된 경우 해당 접미부를 사용하여 *exec*을 호출하려는 시도가 이루어집니다. 그렇지 않은 경우에는 접미부가 EXEC인 *exec*을 호출하려는 시도가 이루어집니다.

## 리턴 코드

*n*

호출된 *exec*의 종료로 설정되는 리턴 코드입니다. [160 페이지의 『EXIT』](#)의 내용을 참조하십시오.

**0**

일반 리턴

**-3**

Exec을 찾을 수 없음

**-10**

Exec 이름이 지정되지 않음

**-11**

올바르지 않은 *exec* 이름

**-12**

GETMAIN 오류

**-99**

내부 오류

## 예

```
'MACRO POOL1:\USERS\USER1\TEST'
```

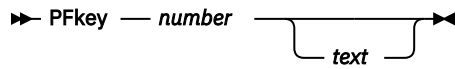
이 예제는 매크로 POOL1:\USERS\USER1\TEST.EXEC을 호출합니다.

**참고:** 매크로는 REXX/CICS FLST 서버에 대한 호출을 수행할 수 있습니다. FLST의 명령행에서 입력할 수 있는 명령은 매크로에서 실행될 수 있습니다.

## PFKEY

PFKEY는 프로그램 기능(PF) 키를 설정하거나 처리합니다.

명령행에서 PFKEY를 입력할 때 다음 구문을 사용하십시오.



## 피연산자

### **number**

설정되거나 처리되는 PF 키를 지정합니다.

### **text**

PF 키가 설정되는 텍스트를 지정합니다.

## 예

```
'PFKEY 3 quit'
'PFKEY 3'
```

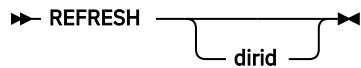
이 예제는 먼저 PFKEY 3를 quit으로 설정한 다음 PF 키를 처리합니다.

**참고:** *text*를 지정하는 경우 PF 키는 텍스트를 사용하여 설정됩니다. *text*를 지정하지 않으면 PF 키가 처리됩니다.

## REFRESH

REFRESH는 파일 목록을 새로 고칩니다.

FLST 명령 옆에 REFRESH를 입력할 때 다음 구문을 사용하십시오.



## 피연산자

### **dirid**

REXX 파일 시스템 디렉토리 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CD 명령 [333 페이지의 『CD』](#)의 내용을 참조하십시오.

## 예

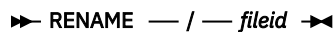
```
'REFRESH'
```

이 예제는 현재 작업 디렉토리의 멤버에 대한 파일 목록을 새로 고칩니다.

## RENAME

RENAME은 파일의 이름을 바꿉니다.

FLST 명령 옆에 RENAME을 입력할 때 다음 구문을 사용하십시오.



## 피연산자

### **fileid**

새 파일 ID를 지정합니다.

**참고:** *fileid*가 이미 존재하는 경우 이 파일 ID는 대체됩니다.

## 예

```
'RENAME / TEST4.EXEC'
```

TEST3.EXEC 옆에 있는 명령 옆에서 실행되는 이 예제는 이름 TEST3.EXEC을 TEST4.EXEC으로 바꿉니다.

명령행에서 RENAME를 입력할 때 다음 구문을 사용하십시오.

➤ RENAME — *fileid1* — *fileid2* ➤

### 피연산자

#### *fileid1*

명령이 작동하는 파일의 파일 ID를 지정합니다.

#### *fileid2*

새 파일 ID를 지정합니다.

참고: *fileid2*가 이미 있는 경우 *fileid1*의 콘텐츠가 이를 대체합니다.

### 예

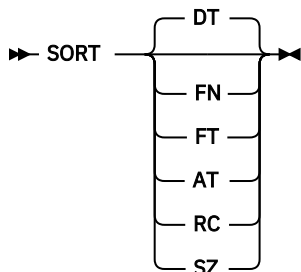
```
'RENAME TEST3.EXEC TEST4.EXEC'
```

명령행에서 실행되는 이 예제는 파일 TEST3.EXEC의 이름을 TEST4.EXEC으로 바꿉니다.

### SORT

SORT는 파일 목록을 정렬합니다.

명령행에서 SORT를 입력할 때 다음 구문을 사용하십시오.



### 피연산자

#### DT

날짜/시간별 파일 정렬을 지정합니다. (기본값입니다.)

#### FN

파일 이름별 파일 정렬을 지정합니다.

#### FT

파일 유형별 파일 정렬을 지정합니다.

#### AT

속성별 파일 정렬을 지정합니다.

#### RC

레코드 번호별 파일 정렬을 지정합니다.

#### SZ

크기별 파일 정렬을 지정합니다.

### 예

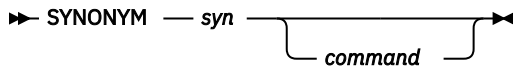
```
'SORT FN'
```

이 예제는 파일 이름별로 파일을 정렬합니다.

### SYNONYM

SYNONYM은 명령 조치를 다른 올바른 명령에 지정합니다.

명령행에서 SYNONYM을 입력할 때 다음 구문을 사용하십시오.



## 피연산자

### **syn**

동의어인 명령 조치를 실행하는 올바른 명령을 지정합니다.

### **command**

올바른 명령을 지정합니다.

## 예

```
'SYNONYM DISCARD RFS DELETE'
```

이 예제는 DISCARD를 RFS 명령 DELETE와 등가가 되도록 합니다.

### **UP**

UP은 위로 한 개 이상의 행을 화면이동합니다.

명령행에서 UP을 입력할 때 다음 구문을 사용하십시오.



## 피연산자

### **n**

화면이동할 행의 수를 지정합니다.

## 예

```
'UP 5'
```

이 예제는 목록에서 5행 뒤로 화면이동합니다.

## FLST 리턴 코드

FLST는 다르게 지정된 경우를 제외하고 REXX/CICS 파일 시스템에 대해 표준 리턴 코드를 사용합니다.

자세한 정보는 RFS 명령 [377 페이지](#)의 『RFS』의 내용을 참조하십시오.

## FLST로부터 exec 및 트랜잭션 실행

대부분의 REXX/CICS exec은 단순히 exec 이름을 입력하여 FLST에서 실행할 수 있습니다. 해당 인수가 포함된 exec 이름은 명령행에 입력되거나 exec 이름은 FLST 명령 열에 입력됩니다. 후자의 경우 파일 ID를 트랜잭션을 위한 인수로 사용합니다. REXX exec은 명령행에서(이 경우 파일 ID가 필요함) 또는 FLST 명령 열에서 EXECUTE 명령을 발행하여 실행됩니다.





## 제 25 장 REXX/CICS 목록 시스템

REXX/CICS는 가상 스토리지에서 테이블 또는 데이터의 목록을 유지보수하기 위한 기능을 제공합니다. 이 기능을 RLS(REXX List System)라고 합니다.

이 시스템은 임시 시스템과 사용자 정보 목록의 관리를 제공합니다. RLS에 액세스하기 위한 외부는 REXX 파일 시스템과 함께 사용된 RFS와 CD 명령 대신에, RLS와 CLD 명령입니다. 또한 RLS는 exec가 아닌, 데이터에 대해 유일합니다.

### 디렉토리 및 목록

RLS에는 한 개의 루트 디렉토리가 있습니다. \*CICREX\*라는 이름의 CICS 임시 스토리지 큐에서 해당 앵커 주소를 읽어 RLS 시스템에 액세스합니다.

이 \*CICREX\* 큐는 RLS의 첫 번째 액세스 후에 한 개의 항목을 포함합니다. 이 항목은 6개의 전자로 된 영역 주소를 포함합니다. RLS 제어 정보 및 루트 디렉토리에 대한 포인터를 포함합니다. 루트 디렉토리 RLS는 \로 이름 지정됩니다. 이 디렉토리는 목록, 저장된 변수 또는 큐라는 특수 목록과 서브디렉토리를 포함할 수 있습니다. 서브디렉토리는 다른 디렉토리 안에 있는 디렉토리입니다. 서브디렉토리는 다른 디렉토리 안에 작성할 수 있습니다. 새 디렉토리는 RLS MKDIR 명령을 사용하여 작성할 수 있습니다. 루트를 제외한 모든 디렉토리는 1 - 250자의 디렉토리 파일 이름으로 구분됩니다. 이를 디렉토리 ID라고 합니다.

항상 존재하는 하나의 RLS 디렉토리는 USERS 디렉토리입니다. 이 디렉토리는 시스템에 있는 사용자에게 해당하는 여러 서브디렉토리를 포함합니다. 목록을 처음 RLS에 저장하는 경우 USERS 디렉토리에서 새 서브디렉토리가 작성될 수 있습니다. CICS에 사인온한 경우 이 새 디렉토리는 사용자 ID로 이름 지정됩니다. 그렇지 않은 경우 디렉토리 이름은 CICS DFLTUSER의 값으로 기본 설정됩니다. 이 디렉토리를 작성하고 나면 해당 개인 디렉토리 안에 원하는 수만큼 서브디렉토리를 작성할 수 있습니다. 작성한 디렉토리 중 어느 하나에 목록을 배치할 수 있습니다.

목록은 항상 데이터 파일입니다. 목록은 디렉토리와 같은 이름 지정 규칙을 사용합니다.

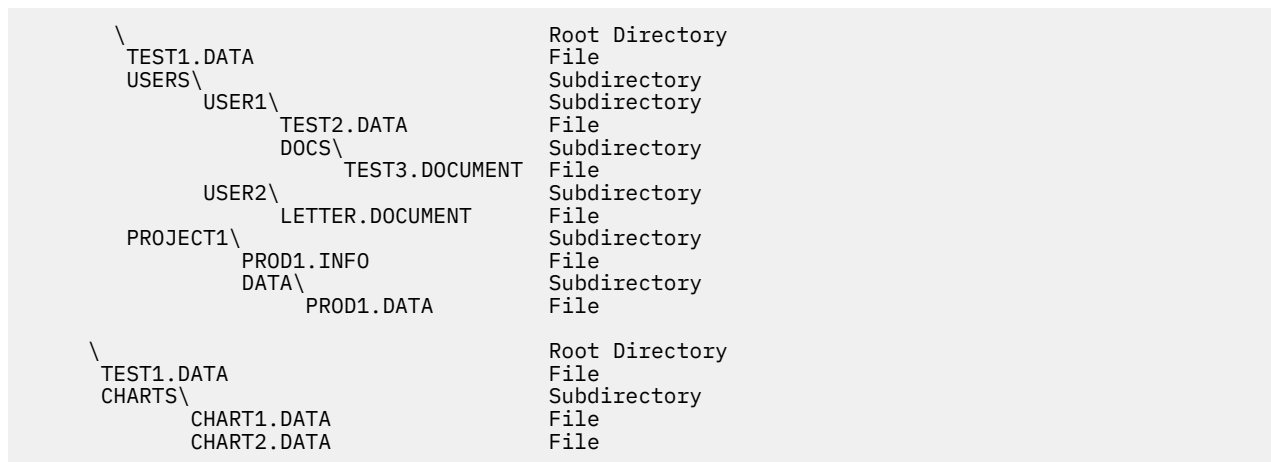
완전한 목록 ID는 \, 뒤에 \가 오는 경로에서 각 디렉토리의 ID 및 목록 ID로 구성됩니다.

### 예제

다음 예제는 완전한 목록 ID를 보여줍니다. USERS 및 USER1은 디렉토리의 ID이고 TEST.DATA는 목록 ID입니다.

```
\USERS\USER1\TEST.DATA
```

다음 예제는 RLS 디렉토리 및 목록을 보여줍니다.



이 예제는 목록 디렉토리 구조를 보여줍니다. 루트 디렉토리는 한 개의 파일(TEST1.DATA)과 두 개의 서브디렉토리(USERS 및 PROJECT1)를 포함합니다. USERS 서브디렉토리 안에는 사용자 ID(USER1과 USER2)에 해당하는 두 개의 서브디렉토리(USER1과 USER2)가 있습니다. 사용자 USER1에게는 그 디렉토리 안에 목록

(TEST2.DATA) 및 서브디렉토리(DOCS)가 있습니다. DOCS 서브디렉토리 내부에는 다른 목록(TEST3.DOCUMENT)이 있습니다. 사용자 USER2에게는 그 디렉토리 안에 파일(LETTER.DOCUMENT)이 있습니다. 루트 디렉토리는 한 개의 파일(TEST1.DATA)과 한 개의 서브디렉토리(CHARTS)를 포함합니다. 서브디렉토리 CHARTS 안에는 두 개의 파일(CHART1.DATA 및 CHART2.DATA)이 있습니다.

## 현재 디렉토리 및 경로

현재 목록 디렉토리는 현재 작업 디렉토리이며 RFS(REXX List System)에서 작업할 때 검색 순서에서 첫 번째입니다.

현재 목록 디렉토리는 CLD 명령 349 페이지의 『CLD』를 사용하여 설정될 수 있습니다. 이 구문은 뒤에 부분 또는 완전한 디렉토리 이름이 오는 CLD입니다. 서브디렉토리에서 상위 디렉토리로 다시 변경하려면 CLD ..를 입력하십시오. 다른 서브디렉토리로 변경하기 위해서는 CLD 뒤에 서브디렉토리 이름을 둘 수 있습니다.

### 예

다음 예제에서 첫 번째 명령은 현재 디렉토리를 \USERS\USER1으로 설정하고 두 번째 명령은 현재 디렉토리를 \USERS\USER1\DOCS로 설정합니다. 세 번째 명령은 현재 디렉토리를 다시 \USERS\USER1로 변경합니다.

```
CLD \USERS\USER1
CLD DOCS
CLD ..
```

**참고:** CLD를 지정하지 않으면 기본 디렉토리는 \USERS\userid\입니다.

## 보안

RLS 명령은 권한 부여된 REXX/CICS 명령입니다.

이는 권한 부여된 사용자만 실행할 수 있거나 리전 시동 JCL의 CICAUTH 또는 CICEXEC ddname에서 로드되는 exec 내에서만 실행될 수 있음을 의미합니다.

## RLS 명령

RLS 명령 환경에서 RLS와 인터페이스하기 위한 명령을 발행합니다.

명령 환경을 RLS로 설정하는 경우 RLS 명령 앞에 RLS를 지정해서는 안 됩니다.

### 예

```
'RLS READ \USERS\USER1\TEST.DATA DATA.'
```

이 예제는 RLS 목록 \USERS\USER1\TEST.DATA의 콘텐츠를 DATA. REXX 복합 변수로 읽습니다.

RLS 명령에 대한 구문이 뒤에 옵니다.

## CKDIR

CKDIR은 기존 RLS 디렉토리 레벨을 확인합니다.

➡ RLS — CKDIR — *dirid* ➡

### 피연산자

#### *dirid*

REXX 목록 시스템 디렉토리 레벨 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CLD 명령 349 페이지의 『CLD』의 내용을 참조하십시오.

### 리턴 코드

RLS 명령 379 페이지의 『RLS』의 내용을 참조하십시오.

예

```
'RLS CKDIR \USERS\USER1\DOCS'
```

이 예제는 기존 디렉토리 \USERS\USER1에서 DOCS라는 디렉토리를 확인합니다.

## DELETE

DELETE는 RLS 목록을 삭제합니다.

➡ RLS — DELETE — *listname* ➡

### 피연산자

#### *listname*

REXX 목록 시스템 목록 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CLD 명령 [349 페이지](#)의 『CLD』의 내용을 참조하십시오.

### 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

예

```
'RLS DELETE \USERS\USER1\TEST.DATA'
```

이 예제는 RLS 목록 TEST.DATA를 삭제합니다.

## LPULL

LPULL은 RLS 큐의 맨 위에서 레코드를 가져옵니다.

➡ RLS — LPULL — *varname* — { *\*QUEUE\**  
*queid* } ➡

### 피연산자

#### *varname*

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

#### *\*QUEUE\**

특수 기본 이름을 지정하는 키워드입니다.

#### *queid*

LPULL, LPUSH 또는 LQUEUE를 통해 액세스하는 특수 RLS 목록 유형의 ID를 지정합니다.

### 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

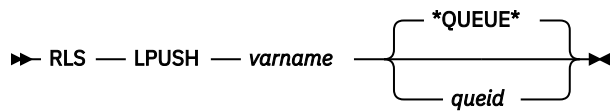
예

```
'RLS LPULL VARA QUEUE1'
```

이 예제는 RLS 큐 QUEUE1의 맨 위에서 레코드를 가져옵니다.

## LPUSH

LPUSH는 RLS 큐(LIFO)의 맨 위에 레코드를 푸시합니다.



### 피연산자

#### **varname**

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

#### **\*QUEUE\***

특수 기본 이름을 지정하는 키워드입니다.

#### **queid**

LPULL, LPUSH 또는 LQUEUE를 통해 액세스하는 특수 RLS 목록 유형의 ID를 지정합니다.

### 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

### 예

```
'RLS LPUSH VARA QUEUE1'
```

이 예제는 레코드(VARA의 콘텐츠)를 RLS 큐 QUEUE1의 맨 위에 푸시합니다.

## LQUEUE

LQUEUE는 RLS 큐의 끝에 레코드를 추가합니다(FIFO).



### 피연산자

#### **varname**

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

#### **\*QUEUE\***

특수 기본 이름을 지정하는 키워드입니다.

#### **queid**

LPULL, LPUSH 또는 LQUEUE를 통해 액세스하는 특수 RLS 목록 유형의 ID를 지정합니다.

### 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

### 예

```
'RLS LQUEUE VARA QUEUE1'
```

이 예제는 레코드(VARA의 콘텐츠)를 RLS 큐 QUEUE1의 끝에 추가합니다.

## MKDIR

MKDIR은 새 RLS 디렉토리 레벨을 작성합니다.

➡ RLS — MKDIR — *dirid* ➡

### 피연산자

#### *dirid*

REXX 목록 시스템 디렉토리 레벨 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CLD 명령 [349 페이지](#)의 『CLD』의 내용을 참조하십시오.

### 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

### 예

```
'RLS MKDIR \USERS\USER1\DOCS'
```

이 예제는 기본 디렉토리 \USERS\USER1에서 DOCS라는 새 디렉토리를 작성합니다.

## READ

READ는 RLS 목록에서 레코드를 읽습니다.

➡ RLS — READ — *listname* — { *DATA.* / *stem.* } ( — UPD — ) ➡

### 피연산자

#### *listname*

목록 ID를 지정합니다.

#### *stem.*

스텝의 이름을 지정합니다. 스텝은 마침표로 종료되어야 합니다. [144 페이지](#)의 『스텝』의 내용을 참조하십시오. 기본 스텝은 DATA.입니다.

### UPD

업데이트할 파일을 인큐하는 키워드입니다.

### 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

### 예

```
'RLS READ \USERS\USER1\TEST.DATA DATA.'
```

이 예제는 DATA. REXX 복합 매개변수에 RLS 목록 \USERS\USER1\TEST.DATA의 전체 콘텐츠를 저장합니다.

**참고:** DATA.0은 목록에서 읽은 레코드의 수로 설정됩니다. DATA.*n*은 목록에서 읽은 *n*번째 레코드를 포함합니다.

## VARDROP

VARDROP은 RLS 저장 변수를 삭제합니다.

➡ RLS — VARDROP — *varname* — *dirid* ➡

## 피연산자

### *varname*

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

### *dirid*

REXX 목록 시스템 디렉토리 레벨 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CLD 명령 [349 페이지](#)의 『CLD』의 내용을 참조하십시오.

## 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

## 예

```
'RLS VARDROP VAR1'
```

이 예제는 현재 디렉토리에서 변수 VAR1을 삭제합니다.

## VARGET

VARGET는 RLS 저장된 변수를 가져와서 이를 동일한 이름의 REXX 변수로 복사합니다.

►► RLS — VARGET — *varname* — *dirid* ◄◄

## 피연산자

### *varname*

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

### *dirid*

REXX 목록 시스템 디렉토리 레벨 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. CLD 명령 [349 페이지](#)의 『CLD』의 내용을 참조하십시오.

## 리턴 코드

RLS 명령 [379 페이지](#)의 『RLS』의 내용을 참조하십시오.

## 예

```
'RLS VARGET VAR1'
```

이 예제는 현재 디렉토리의 변수 VAR1 값을 VAR1이라고 하는 REXX 변수 이름으로 복사합니다.

## VARPUT

VARPUT은 REXX 변수를 가져와서 이를 동일한 이름의 RLS 저장된 변수로 복사합니다.

►► RLS — VARPUT — *varname* — *dirid* ◄◄

## 피연산자

### *varname*

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

### *dirid*

REXX 목록 시스템 디렉토리 레벨 ID를 지정합니다. 이 ID는 부분적이거나 완전합니다. 자세한 정보는 CLD 명령 [349 페이지](#)의 『CLD』의 내용을 참조하십시오.

## 리턴 코드

RLS 명령 379 페이지의 『RLS』의 내용을 참조하십시오.

## 예

```
'RLS VARPUT VAR1'
```

이 예제는 REXX 변수 VAR1의 값을 가져와서 이를 현재 디렉토리의 변수 VAR1에 복사합니다.

## WRITE

WRITE은 레코드를 RLS 목록에 씁니다.



## 피연산자

### *listname*

목록 ID를 지정합니다.

### *stem.*

스탬의 이름을 지정합니다. 스탬은 마침표로 종료되어야 합니다. 144 페이지의 『스탬』의 내용을 참조하십시오. 기본 스탬은 DATA.입니다.

## 리턴 코드

RLS 명령 379 페이지의 『RLS』의 내용을 참조하십시오.

## 예

```
'RLS WRITE \USERS\USER1\TEST.DATA DATA.'
```

이 예제는 REXX 복합 변수 DATA.의 전체 콘텐츠를 RLS 목록 \USERS\USER1\TEST.DATA에 저장합니다.

**참고:** DATA.0을 목록에 쓸 레코드의 수로 설정하십시오.





## 제 26 장 REXX/CICS 명령 정의

REXX/CICS Command Definition Facility는 REXX 명령과 환경을 정의하거나 재정의하기 위한 방법을 제공합니다.

Command Definition Facility는 다음을 위한 기능을 제공합니다.

- REXX exec로부터 새 명령과 환경을 정의합니다.
- 공통 명령 환경을 다중 독립적 개발자와 공유합니다.
- 새 REXX 명령을 REXX 언어로 작성합니다.
- 투명하게 명령의 구현 언어를 변경합니다.
- 선택적으로 권한 부여된 명령을 정의합니다.
- (코드 변경 없이) 기존 명령 이름을 재정의합니다.

### 백그라운드

REXX의 강점은 그 확장성입니다. 자체 외부 기능, 서브루틴 또는 명령을 작성하여 REXX 언어의 기능을 확장할 수 있습니다. 이로 인해 REXX의 한 가지 사용은 애플리케이션 통합 플랫폼으로 사용하는 것입니다.

REXX/CICS는 LOB(Line Of Business) 애플리케이션 기능, REXX 언어 기능, CICS 시스템 기능 및 다양한 소프트웨어 제품을 통합된 소프트웨어 플랫폼으로 완벽하게 통합하는 기능을 제공합니다.

기능을 추가하거나 외부 기능 또는 제품에 대한 인터페이스를 제공하기 위해 REXX 언어를 확장하는 데 사용되는 기본 메소드는 다음과 같습니다.

- REXX 외부 기능(또는 서브루틴)
- REXX 명령(하위 명령이라고도 함)

둘을 구분하는 예로 REXX 배열을 정렬하기 위한 기능을 REXX에 추가하려 한다고 가정해봅시다. 다음과 같이 ARRYSORT라는 정렬 기능을 추가하여 이를 수행할 수 있습니다.

```
x = ARRYSORT(STEM1.)
```

ARRYSORT라고 하는 REXX 명령을 추가하여 이를 수행할 수도 있습니다.

```
'ARRYSORT STEM1.'
```

**참고:** 명령 예제에서는 명령 주위에 따옴표를 배치하지 않아도 됩니다. 그러나 REXX 변수 대체와 관련되지 않은 명령 문자열 부분에 작은 따옴표 또는 큰따옴표를 배치하는 것은 우수한 코딩 사례입니다.

REXX 외부 기능은 일반적으로 REXX 언어의 기능을 확장하는 데 사용되는 반면 REXX 명령은 일반적으로 애플리케이션, 제품 또는 시스템 기능 외부에 인터페이스하는 데 사용됩니다. 그러나 두 메소드 다 어느 방식으로든 사용할 수 있습니다.

REXX 명령 간 몇 가지 주요 차이점은 다음과 같습니다.

- 함수는 항상 결과를 리턴하고 명령은 그렇지 않습니다.
- 함수는 오류 조건을 발생시키지만 명령은 항상 리턴 코드를 설정합니다.

### 명령 정의

DEFSCMD 명령을 사용하여 사용자 ID에만 영향을 미치는 기본 명령 정의를 수행합니다. 시스템 관리자 또는 시스템 프로그래머인 경우 DEFSCMD 명령을 사용하여 시스템 범위 REXX/CICS 명령 정의를 수행합니다.

DEFSCMD 및 DEFSCMD 명령은 REXX exec 내에서 사용되어 REXX 명령 정의를 정의하거나 변경합니다. 특수 권한 부여 없이 DEFSCMD 명령을 사용하여 사용자 고유의 명령 정의를 추가하거나 변경할 수 있습니다. DEFSCMD를 사용하여 다른 REXX/CICS 사용자에게 영향을 미치는 명령 정의를 변경하려면 REXX/CICS 권한 부여된 사용자여야 합니다. 세부사항은 355 페이지의 『DEFSCMD』 및 357 페이지의 『DEFSCMD』의 내용을 참조하십시오.

## REXX 프로그램에 전달되는 명령 인수

REXX/CICS 명령이 REXX에서 작성되고 해당 명령이 사용되는 경우 REXX 프로그램(DEFRCMD 또는 DEFSCMD에 의해 정의됨)이 호출되거나 다시 작동합니다(WAITREQ로 인한 "휴면" 상태에서부터).

프로그램이 호출되면 명령 문자열은 exec에 인수로 전달됩니다. 또한 프로그램이 호출되면 가장 처음 발행된 WAITREQ 명령(있는 경우)은 즉시 실패하고 이 명령 문자열은 REXX 변수 REQUEST에 배치됩니다. REXX exec이 이미 이전에 시작되어 요청을 대기 중인 경우(이전 WAITREQ 명령으로 인해) 명령 문자열은 REXX 변수 REQUEST에만 배치됩니다.

**참고:** REXX에 기록된 명령 프로그램은 REXX exec에서 C2S 및 S2C 명령을 사용하여 호출되도록 하는 REXX 변수의 콘텐츠를 가져오고 설정할 수 있습니다. 자세한 정보는 [354 페이지의 『C2S』](#) 및 [386 페이지의 『S2C』](#)의 내용을 참조하십시오.

## 어셈블러 프로그램에 전달되는 명령 인수

REXX/CICS 명령 프로그램은 어셈블러 언어로 작성할 수 있습니다.

어셈블러 언어 루틴이 적절하게 정의된 CICS 프로그램에 있어야 합니다(예를 들어, CEDA DEFINE PROGRAM 명령을 사용하여). 이러한 프로그램은 DEFRCMD 또는 DEFSCMD 명령에 CICS LINK 옵션이 지정된 경우 EXEC CICS LINK에 의해 호출됩니다.

또는 DEFRCMD 또는 DEFSCMD 명령이 CICS LOAD 옵션을 지정하면 프로그램은 현재 CICS 태스크에 대해 호출되도록 하는 첫 번째 명령에 의해 EXEC CICS 로드되고 그 로드 주소가 기억됩니다. 이 프로그램을 사용하는 동일한 CICS 태스크의 후속 명령은 프로그램으로의 직접 분기 입력을 수행합니다(어셈블러 BASSM 명령어에 의해). 올바른 모드 전환(있는 경우)이 발생하도록 어셈블러 BSM 명령어를 사용하여 이러한 어셈블러 프로그램이 제어를 리턴하는 것이 권장됩니다.

다음 정보는 이러한 프로그램에 입력 시 어셈블러 언어 명령 프로그램이 제어 및 매개변수를 가져올 때의 레지스터 콘텐츠를 설명합니다.

**DEFRCMD CICS LOAD가 지정되는 경우 입력 스펙은 다음과 같습니다.**

명령 프로그램에 대한 코드가 직접 분기를 통해 제어를 가져오는 경우 레지스터의 콘텐츠는 다음과 같습니다.

**레지스터 0**

예측 불가능

**레지스터 1**

CICPARMS 제어 블록의 주소

**레지스터 2-12**

예측 불가능

**레지스터 13**

18개의 전자로 된 레지스터 저장 영역의 주소

**레지스터 14**

리턴 주소

**레지스터 15**

시작점 주소

프로그램은 호출자로 돌아가기 전에 CICPARMS RETCODE 필드에 반영되도록 할 리턴 코드를 배치해야 합니다.

**DEFRCMD CICS LINK가 지정되는 경우 입력 스펙은 다음과 같습니다.**

명령 프로그램에 대한 코드가 EXEC CICS LINK를 통해 제어를 가져오는 경우 CICS commarea는 CICPARMS 제어 블록을 포함합니다.

프로그램은 호출자로 돌아가기 전에 CICPARMS RETCODE 필드에 반영되도록 할 리턴 코드를 배치해야 합니다.

## CICPARMS 제어 블록

이 주제에는 제품별 프로그래밍 인터페이스 정보가 포함됩니다.

다음 테이블은 어셈블러 루틴으로 전달된 매개변수를 맵핑하기 위한 CICPARMS 제어 블록을 보여줍니다.

표 4. CICPARMS 제어 블록			
오프셋(10진수)	바이트 수	필드 이름	설명
0	12		IBM용으로 예약됩니다.
12	4	RXWBADDR	CICGETV 스텝 루틴에 대한 호출(REXX 변수 액세스를 위해) 전에 레지스터 10에 배치되어야 하는 REXX 작업 블록 주소
16	8	ENVNAME	DEFCMD 또는 DEFSCMD 명령 정의에서 가져오는 내부 환경 이름
24	16	CICCMD	명령 정의에서 가져온 내부 명령 이름 또는 별표가 지정된 경우 명령 문자열로부터의 실제 명령 이름
40	4	ARGSTR	명령 문자열에서 명령 이름 뒤 첫 번째 공백이 아닌 문자로 시작되는 명령 인수 문자열의 주소
44	4	ARGLEN	above 인수 문자열의 길이(문자 수 단위)
48	4	PLIST	16진 높은 값(X'FFFFFFFFFFFFFFFF')의 목록 펜스 끝이 뒤에 오며, 8자 토큰으로 구문 분석되는 명령행의 표준 구문 분석된 목록 주소
52	4	EPLIST	이전에 나열되었지만 형식이 다른 표준 PLIST와 일치하는 확장된 매개변수 목록의 주소. 확장된 PLIST에는 각 토큰에 대해 8바이트 항목이 있습니다. 처음 4바이트는 토큰을 포함하는 문자열 시작의 전자 주소입니다. 두 번째 단어는 토큰의 길이(바이트 수)를 포함합니다.
56	4	RETCODE	명령 실행 후 즉시 exec에 반영될 리턴 코드입니다. 이 리턴 코드는 자동으로 특수 REXX 변수 RC에 배치됩니다.
60	4		IBM용으로 예약됨
64	4	USERWORD	사용자용이므로 정보를 여러 명령 루틴 호출에 전달할 수 있습니다.
68	4		IBM용으로 예약됨
72	4		IBM용으로 예약됨
76	1	TYPEFLAG	DEFCMD 또는 DEFSCMD 정의의 호출 유형을 식별하는 한 개의 문자 코드입니다. REXX의 코드는 R이고 CICSLINK의 경우 C이며 CICSLOAD의 경우 L입니다.
77	1	ITRACE	내부 추적 플래그입니다. 이는 한 문자 코드이며 내부 추적이 활성화인지 및 활성화된 추적 레벨을 나타내는 1 - 9까지의 값을 가집니다. 값 0은 정상 상황에서 추적을 하지 않음을 나타냅니다. 1 - 9까지의 값은 점증적으로 세부 추적이 요청됨을 나타냅니다.

## 비REXX 언어 인터페이스

REXX/CICS에서는 REXX 프로세스를 비REXX 프로세스로 변환할 수 있습니다.

이를 수행하려면 처리될 명령을 발행한 REXX exec의 REXX 변수에 비REXX 명령 루틴이 액세스할 수 있어야 합니다. 이를 수행하기 위해 사용되는 루틴은 CICGETV라고 하고 명령 루틴을 사용하여 링크 편집되고 다음 정보에 설명된 대로 호출되어야 합니다.

## CICGETV: REXX 변수 가져오기, 설정하기 또는 삭제를 위한 호출

CICGETV는 어셈블러 REXX/CICS 명령 루틴에서 이 링크 편집 스텝 서브루틴을 호출하여 명령을 발행한 REXX 프로그램에서 REXX 변수를 검색하거나 설정하거나 삭제합니다.

이 주제에는 제품별 프로그래밍 인터페이스 정보가 포함됩니다.

➤ CALL — CICGETV — , — ( — operands — ) ➤

피연산자

➤ *varname\_addr* — , *varname\_len* — , *data\_addr* — , *data\_len* — , *function\_name* ➤

**피연산자**

***varname\_addr***

REXX 변수의 이름을 포함하는 문자열의 주소를 지정합니다. 변수 이름(이 주소가 지정하는)은 대문자여야 합니다.

***varname\_len***

변수 이름의 길이를 지정합니다.

***data\_addr***

변수 콘텐츠의 주소가 있는 전자 주소를 지정합니다.

***data\_len***

*data\_addr*(전자)이 가리키는 영역의 길이를 지정합니다.

***function\_name***

수행할 특정 CICGETV 함수를 지정합니다. 다음과 같이 세 가지 선택사항이 있습니다.

**GET**

REXX 변수의 주소 및 길이를 검색합니다.

**PUT**

REXX 변수를 작성하거나 대체합니다.

**DEL**

REXX 변수를 삭제합니다.

**참고:**

1. CICGETV 링크 편집된 루틴 스텝에 대한 호출 바로 전에 전달된 매개변수(CICPARMS)에서 RXWBADDR 필드의 값과 함께 레지스터 10이 로드되어야 합니다.
2. 존재하지 않는 변수에 대해 가져오기 요청이 발행되면 리턴되는 값은 변수 값과 같습니다.
3. CICGETV는 18개의 전자 길이 영역을 가리키는 R13과 표준 저장 영역 규칙을 사용합니다. R1은 표준 매개변수 목록을 가리킵니다. R14는 리턴 주소를 포함합니다. R15는 CICGETV에 대한 입력 시 CICGETV의 시작점을 리턴합니다. 리턴 시 R15는 리턴 코드를 포함합니다.
4. CICGETV 모듈은 REXX/CICS 배포 라이브러리에 있습니다.
5. 리턴 코드 0은 오퍼레이션의 성공 또는 내부 오류(예: 스토리지 한계가 초과된 상황)를 나타내는 10진수 99를 표시하기 위해 반영됩니다.

## 제 27 장 REXX/CICS Db2 인터페이스

REXX/CICS Db2 인터페이스는 SQL문 및 Db2 명령을 REXX exec에서 실행하는 수단을 제공합니다.

SQL은 동적으로 준비되고 실행됩니다. Db2 인스트루멘테이션 기능 인터페이스(IFI)를 사용하여 Db2 명령을 실행합니다. REXX/CICS Db2 인터페이스는 REXX 사전정의된 변수에서 SQL의 결과를 제공합니다. REXX/CICS Db2 인터페이스는 DB2® 버전 2.3 이상을 지원합니다. 이 정보는 REXX/CICS에서 Db2에 인터페이스를 사용하는 방법을 설명합니다.

SQL문 또는 Db2 명령에 관한 자세한 정보는 [SQL: z/OS용 Db2 제품 문서의 Db2 언어 및 Db2 명령의 내용을 참조하십시오](#). IFI에 관한 자세한 정보는 [z/OS용 Db2 제품 문서의 인스트루멘테이션 기능 인터페이스\(IFI\)에 대한 프로그래밍의 내용을 참조하십시오](#).

**참고:** REXX/CICS 트랜잭션 ID는 Db2 자원 제어 테이블(RCT)에 입력되어야 합니다. 샘플 JCL CICRCT를 참조하십시오.

### 프로그래밍 고려사항

SQL을 REXX exec 내에 임베드하려면 호스트 명령 환경을 변경해야 합니다. 뒤에 환경의 이름이 오는 ADDRESS 명령어가 호스트 명령 환경을 변경하는 데 사용됩니다.

ADDRESS 명령어는 다음 두 가지 양식을 가집니다. 한 가지 양식은 명령어 뒤에 발행된 모든 명령에 영향을 미치고 다른 하나는 단일 명령에만 영향을 미칩니다. 호스트 명령 환경에 대한 자세한 정보는 87 페이지의 『호스트 명령 환경 변경』의 내용을 참조하고 ADDRESS 명령어에 대한 자세한 정보는 151 페이지의 『ADDRESS』의 내용을 참조하십시오.

REXX/CICS Db2 인터페이스를 지원하는 REXX/CICS 명령 환경은 다음과 같습니다.

#### EXECDB2

Db2 명령을 지원하는 명령 환경입니다.

#### EXECSQL

SQL문을 지원하는 명령 환경입니다.

**참고:** EXECSQL 및 EXECDB2는 권한 부여된 명령입니다. EXECSQL 및 EXECDB2 명령 환경을 사용하려면 REXX/CICS 권한 부여된 사용자여야 합니다.

REXX/CICS는 REXX 명령문 및 명령을 대화식으로 처리하는 데 사용할 수 있는 CICRXTRY라고 하는 exec을 제공합니다. CICRXTRY는 의사 대화식일 수 있습니다. PSEUDO 및 SETSYS PSEUDO 명령은 의사 대화식 모드를 켜거나 끄는 데 사용됩니다. 환경이 의사 대화식으로 설정되면 CICRXTRY에서 발행된 SQL문이 커밋됩니다. 환경이 대화식으로 설정되면 CICRXTRY exec에서 발생한 SQL문은 커밋되지 않고 잠겨진 모든 자원은 CICRXTRY exec을 종료하거나 CICS SYNCPOINT 명령을 발행할 때까지 잠긴 상태로 유지됩니다. 긴 REXX exec에 SQL문을 임베드하는 경우 비슷한 고려사항을 작성해야 합니다.

### SQL문 임베드

EXECSQL 명령 환경을 사용하여 SQL을 처리할 수 있습니다. 각 SQL문은 CICS Db2 접속 기능을 사용하여 동적으로 준비되고 실행됩니다.

EXECSQL 환경에 경로 지정되는 REXX 명령으로 올바른 SQL문을 작성하여 각 요청을 작성할 수 있습니다. SQL문은 다음 요소로 구성됩니다.

- SQL 키워드
- 사전 선언된 ID
- 리터럴 값.

다음 구문을 사용하십시오.

```
"EXECSQL statement"
```

또는

```
ADDRESS EXECSQL
"statement"
"statement"
.
.
.
```

SQL은 둘 이상의 행에 있을 수 있습니다. 다음과 같이 명령문의 각 파트는 따옴표로 묶이고 쉼표는 추가 명령문 텍스트를 구분합니다.

```
ADDRESS EXECSQL
"SQL text",
"additional text",
.
.
.
"final text"
```

다음 규칙이 임베디드 SQL에 적용됩니다.

- 다음 SQL을 EXECSQL 명령 환경에 직접 전달할 수 있습니다.

```
ALTER
CREATE
COMMENT ON
DELETE
DROP
EXPLAIN
GRANT
INSERT
LABEL ON
LOCK
REVOKE
SELECT
SET CURRENT SQLID
UPDATE.
```

- 다음 SQL문을 사용할 수 없습니다.

```
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
```

- 호스트 변수는 SQL에서 허용되지 않습니다. 대신 REXX 변수를 사용하여 입력 데이터를 EXECSQL 환경에 전달할 수 있습니다. REXX 변수는 따옴표 안에 임베드되지 않습니다. EXECSQL 환경의 출력은 REXX 사전 정의된 변수에서 제공됩니다(297 페이지의 『결과 수신』 참조).
- SQL SELECT문을 코드화할 때 INTO 절을 사용할 수 없습니다. 대신 REXX/CICS Db2 인터페이스는 스템 이름이 Db2 열 이름과 같은 복합 변수에 요청된 항목을 리턴합니다.
- SELECT문에 대해 리턴되는 기본 행 수는 250입니다. 더 많거나 적은 행이 필요한 경우 SELECT문을 발행하기 전에 REXX 변수 SQL\_SELECT\_MAX을 설정할 수 있습니다.

## 결과 수신

EXECSQL 명령 환경은 사전 정의된 REXX 변수에 결과를 리턴합니다.

이러한 변수는 다음과 같습니다.

### RC

각 오퍼레이션은 이 리턴 코드를 설정합니다. 가능한 값은 다음과 같습니다.

***n***

SQL문의 결과로 오류 또는 경고가 발생한 경우 SQLCODE입니다.

**0**

SQL문이 EXECSQL 환경에 의해 처리되었습니다. SQLCA를 위한 REXX 변수는 SQL문의 완료 상태를 포함합니다.

**30**

SQLDSECT 변수를 빌드하기에는 메모리가 부족합니다.

**31**

SQL문 영역을 빌드하기에는 메모리가 부족합니다.

**32**

SQLDA 변수를 빌드하기에는 메모리가 부족합니다.

**33**

SELECT문의 결과 영역을 빌드하기에는 메모리가 부족합니다.

### SQLCA

SQL문이 처리된 후 일련의 SQLCA 변수가 업데이트됩니다. SQLCA의 항목은 298 페이지의 『SQL 통신 영역 사용』에 설명되어 있습니다.

### SQL\_COLNAME.*n*

데이터가 SELECT문에 의해 리턴된 각 Db2 열의 이름을 포함합니다. SQL\_COLUMNS는 *n*의 최대값으로 사용되어야 합니다.

### SQL\_COLTYPE.*n*

데이터가 SELECT문에 의해 리턴된 각 Db2 열의 유형을 포함합니다. SQL\_COLUMNS는 *n*의 최대값으로 사용되어야 합니다.

**참고:** 모든 데이터 유형이 지원되더라도 모두 표시 가능한 것은 아닙니다. REXX 함수를 사용하여 데이터를 원하는 형식으로 변환할 수 있습니다.

SQL\_COLTYPE에서 발견된 특정 SQLTYPE 코드의 의미에 대한 정보는 [SQL: z/OS용 Db2 제품 문서의 Db2 언어의](#) 내용을 참조하십시오.

### SQL\_COLLEN.*n*

데이터가 SELECT문에 의해 리턴된 각 Db2 열의 길이를 포함합니다. 데이터 유형이 DECIMAL인 경우 스케일은 열의 길이 뒤에(한 개의 공백 간격 뒤에) 배치됩니다. SQL\_COLUMNS는 *n*의 최대값으로 사용되어야 합니다.

### SQL\_COLUMNS

리턴된 열의 수의 개수를 포함합니다.

### *column.n*

SQL SELECT문의 결과는 이러한 REXX 복합 변수에 저장됩니다. *column*은 Db2 열의 이름입니다. 각 항목은 Db2의 한 행에 대한 데이터를 포함합니다. 리턴된 SQL 행 수의 개수는 *column.0*에 포함됩니다. 이 개수는 *n*의 최대값으로 사용되어야 합니다.

### SQLCOLn.1

CURRENT SQLID, MAX 및 AVG와 같은 일부 SELECT 함수는 특정 Db2 열과 연관되지 않습니다. 결과를 보려면 열 이름 SQLCOLn.1을 참조해야 합니다.

n은 1부터 시작되며 SELECT문에 포함된 각 함수에 대해 1씩 증분됩니다. SQLCOLn으로 표현되는 모든 열은 SQL\_COLNAME 복합 변수에 표시됩니다.

## SQL 통신 영역 사용

SQL 통신 영역(SQLCA)을 구성하는 필드는 SQL문을 발행할 때 REXX/CICS Db2 인터페이스에 의해 자동으로 포함됩니다.

SQLCA의 SQLCODE 및 SQLSTATE 필드에는 SQL 리턴 코드가 들어 있습니다. 이러한 값은 각 SQL문이 실행된 후에 REXX/CICS Db2 인터페이스에 의해 설정됩니다.

SQLCA 필드는 연속 데이터 영역이 아닌 별도의 변수로 유지보수됩니다. 유지보수되는 변수는 다음과 같이 정의됩니다.

### SQLCODE

1차 SQL 리턴 코드입니다.

### SQLERRM

오류 및 경고 메시지 토큰입니다. 인접한 토큰은 X'FF'을 포함하는 바이트로 분리됩니다.

### SQLERRP

제품 코드이고 오류가 있는 경우 오류를 리턴한 모듈의 이름입니다.

### SQLERRD.n

진단 정보를 포함하는 6개의 변수입니다. (변수 n은 1 - 6 사이의 숫자입니다.)

참고: DELETE, INSERT 및 UPDATE 명령의 영향을 받는 SQL 행의 수 개수는 SQLERRD.3에 포함됩니다.

### SQLWARN.n

경고 플래그를 포함하는 11개의 변수입니다. (변수 n은 0 - 10 사이의 숫자입니다.)

### SQLSTATE

대체 SQL 리턴 코드입니다.

## SQL문 사용 예제

이 예제에서 프로그램은 부서의 이름을 묻는 프롬프트를 표시하고 EMPLOYEE 테이블에서 해당 부서 모든 구성원의 이름 및 전화번호를 얻고 해당 정보를 화면에 표시합니다.

```
/******  
/* 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
```



## Db2 명령 임베드

EXECDB2 명령 환경을 사용하여 Db2 명령을 처리할 수 있습니다. 각 Db2 명령은 Db2 IFI(Instrumentation Facility Interface)로 전달됩니다.

TSO 하의 DSN을 사용하여 실행된 대부분의 Db2 명령은 REXX/CICS Db2 인터페이스를 사용하여 CICS에서 실행될 수 있습니다. Db2 메시지는 사전 정의된 REXX 복합 변수 DB2\_OUTPUT.n에서 리턴됩니다.

EXECDB2 환경을 지정한 REXX 명령으로서 유효한 Db2 명령을 써서 각 요청을 작성할 수 있습니다. 다음 구문을 사용하십시오.

```
ADDRESS EXECDB2 "DB2 command"
```

또는

```
ADDRESS EXECDB2
"DB2 command"
"DB2 command"
.
.
.
```

Db2 명령은 둘 이상의 행에 존재할 수 있습니다. 명령의 각 파트는 단일 작은따옴표로 묶이며 쉼표는 추가 명령 텍스트를 다음과 같이 구분합니다.

```
ADDRESS EXECDB2
"DB2 command",
"additional text",
.
.
.
"final text"
```

다음 규칙은 임베디드된 Db2 명령에 적용됩니다.

- REXX exec는 많은 Db2 명령을 실행할 수 있지만 어떤 명령을 사용할 수 있는지 주의하십시오. 예를 들어, "-STOP DB2"를 사용하지 마십시오.
- 다음 Db2 명령은 EXECDB2 명령 환경에 직접 전달됩니다.

```
-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

- 다음 Db2 명령을 사용할 수 없습니다.

- START DB2. MVS 콘솔에서만 이 명령을 실행할 수 있습니다.

- EXECDB2 환경에 입력 데이터 전달 시 REXX 변수를 사용할 수 있습니다. REXX 변수는 따옴표로 임베드되지 않습니다. EXECDB2 환경의 출력은 REXX 사전 정의된 변수에서 제공됩니다(300 페이지의 『결과 수신』 참조).
- Db2 명령 문자열은 6 ~ 4092자 사이의 길이어야 합니다.
- Db2 IFI에서 리턴된 모든 메시지의 총 길이는 24K로 제한됩니다. 24K 바이트의 메모리에 맞는 것 보다 더 많은 메시지가 있는 경우, Db2 명령을 다시 실행할 수 있지만 특정 추가 매개변수 값으로 출력 메시지의 수를 줄일 수 있습니다.

## 결과 수신

EXECDB2 명령 환경은 결과로 사전 정의된 REXX 변수를 리턴합니다.

이러한 변수는 다음과 같습니다.

### RC

각 조작은 이 리턴 코드를 설정합니다. 가능한 값은 다음과 같습니다.

#### ***n***

Db2 IFI(Instrumentation Facility Interface)에 대한 호출의 결과를 표시하는 양의 값입니다. Db2 IFI에서 RC가 0(영)이 아닌 경우, REXX 변수 DB2\_RC2는 Db2 IFI 이유 코드를 포함합니다. DB2\_RC2 값은 오류 판별을 위해 Db2 코드를 사용하여 RC와 함께 사용됩니다. [z/OS용 Db2 제품 문서의 Db2 코드의 내용을 참조하십시오.](#)

#### **0**

Db2 명령은 Db2 IFI에서 처리되었습니다.

#### **50**

지정된 Db2 명령은 Db2 IFI에서 처리되도록 허용되기에 너무 짧거나 너무 길입니다. Db2 명령은 6보다 적거나 4092 문자 길이보다 큼니다.

#### **51**

Db2 IFI에 대해 출력 영역을 빌드하기에 메모리가 부족했습니다.

#### **52**

Db2 IFI에 대해 통신 영역을 빌드하기에 메모리가 부족했습니다.

#### **53**

Db2 IFI에 대해 리턴 영역을 빌드하기에 메모리가 부족했습니다.

#### **54**

DB2\_RC2 REXX 변수는 빌드될 수 없습니다.

#### **55**

DB2\_BNM REXX 변수는 빌드될 수 없습니다.

#### **56**

DB2\_OUTPUT.*n* REXX 변수는 빌드될 수 없습니다.

#### **57**

DB2\_OUTPUT.0 REXX 변수는 빌드될 수 없습니다.

## DB2\_OUTPUT.n

Db2 IFI에서 리턴된 Db2 메시지는 이 REXX 복합 변수에 저장됩니다. 각 DB2\_OUTPUT.n 항목은 하나의 Db2 메시지를 포함합니다. DB2\_OUTPUT.0은 리턴된 Db2의 수 개수를 포함합니다. 개수는 *n*에 대한 최대값으로 사용됩니다.

## DB2\_RC2

Db2 IFI의 리턴 코드가 0(영)이 아닌 경우, 이 변수는 리턴 코드와 연관된 이유 코드를 포함합니다. DB2\_RC2 변수는 16진수 값을 포함합니다. 인쇄할 수 있는 값으로 변환하려면 C2X 기본 제공 REXX 함수를 사용하십시오. 187 페이지의 『C2X(문자를 16진수로)』의 내용을 참조하십시오. DB2\_RC2 값은 오류 판별을 위해 Db2 코드를 사용하여 RC와 함께 사용됩니다. [z/OS용 Db2 제품 문서의 Db2 코드](#)의 내용을 참조하십시오.

## DB2\_BNM

Db2 IFI가 모든 Db2 메시지를 리턴할 수 없는 경우, DB2\_BNM은 이동되지 않은 바이트 수를 포함합니다. 리턴된 모든 메시지의 길이가 24K를 초과하면 이 상황이 발생할 수 있습니다. 출력을 줄이기 위해 더 많은 매개 변수 값을 지정하면서 Db2 명령을 다시 실행할 수 있습니다.

## Db2 명령을 사용하는 예

REXX 코드의 예는 데이터베이스의 모든 테이블스페이스가 RW 상태에 있는지 여부를 알기 위해 검사합니다.

```
/******  
/* 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
```



## 제 28 장 REXX/CICS 상위 레벨 클라이언트/서버 지원

REXX/CICS는 REXX 언어 클라이언트/서버 지원을 소개합니다. REXX/CICS는 상위 레벨 클라이언트/서버 기능을 제공합니다.

이 기능은 다음을 포함합니다.

- 상위 레벨의 고유하고 투명한 REXX 클라이언트 인터페이스
- REXX 기반 애플리케이션 클라이언트 및 서버를 지원하십시오.

클라이언트/서버 컴퓨팅의 일부 장점은 다음과 같습니다.

- 메인프레임, 소형 컴퓨터 및 워크스테이션의 장점을 투명한 방식으로 효율적으로 통합하는 기능입니다.
- 컴퓨터의 시스템의 크기를 점진적으로 늘리거나 줄이는 기능입니다(적당한 크기).
- 관리 가능한 클라이언트와 서버 세트로 분류하여 복잡한 애플리케이션 시스템을 단순화합니다.
- 애플리케이션과 데이터는 더 좋은 성능, 무결성 또는 보안을 위한 네트워크 전반에 걸쳐 분포될 수 있습니다.
- 컴퓨터 시스템이나 워크스테이션과 달리 네트워크에 상주할 때마다 데이터와 애플리케이션에 대한 엔터프라이즈 전반 액세스를 위한 기능입니다.

### 상위 레벨의 고유하고 투명한 REXX 클라이언트 인터페이스

REXX/CICS는 REXX에서 ADDRESS 키워드 명령어를 통해 클라이언트 REXX exec에서 애플리케이션 서버로의 상위 레벨이며 사용이 용이한 인터페이스를 지원합니다.

ADDRESS 명령어는 후속 REXX 명령 문자열을 처리하기 위해 호출되는 외부 프로시저의 이름을 판별하는 데 사용되는 외부 환경 이름을 포함합니다.

REXX/CICS는 애플리케이션 서버의 이름이 될 환경 이름(ADDRESS 명령어로 지정됨)에 대한 선택적 기능을 제공합니다. 이 기능은 REXX/CICS DEFCMD 및 DEFSCMD 명령에 의해 제공됩니다.

DEFCMD 명령은 REXX 명령 및 환경을 정의(또는 재정의)하는 기능을 제공하고 환경 명령 조합이 기존 CALLED 루틴 또는 REXX 애플리케이션 서버에 의해 처리되는지 여부를 지정하는 기능을 제공합니다.

### REXX 기반 애플리케이션 클라이언트 및 서버에 대한 지원

REXX 클라이언트 인터페이스 외에도 여러 기능이 REXX로 작성된 애플리케이션 서버의 사용을 위한 지원을 제공합니다.

한 가지 기능은 서버가 클라이언트로부터의 요청을 대기하기 위해 사용하는 WAITREQ 명령입니다. 다른 기능인 C2S 및 S2C 명령은 서버가 클라이언트 변수의 콘텐츠를 폐치하거나 설정할 수 있도록 하는 기능을 제공합니다. 다른 기능인 ASI(Automatic Server Initiation)는 요청이 클라이언트로부터 도착하면 서버가 자동으로 시작될 수 있도록 하기 위해 제공됩니다.

### 클라이언트/서버 계산에서 REXX의 값

클라이언트/서버 솔루션을 구현하기 위해 REXX을 사용하는 것의 몇 가지 장점이 나열됩니다.

- 빠른 개발 주기 및 소스 기반 대화식 디버깅이 포함된 REXX/CICS에서 REXX 해석기 지원의 가용성은 빠른 프로토타입 작성과 복합 시스템 개발을 가능하게 합니다.
- REXX/CICS의 상위 레벨 클라이언트/서버 인터페이스는 개발 생산성을 향상시키고 유지보수 비용을 낮출 수 있습니다.
- REXX/CICS에서는 REXX 클라이언트 및 서버가 비REXX 언어로 레코드될 수 있으므로 필요한 경우 애플리케이션 시스템의 성능 집약 부분을 선택적으로 재작성할 수 있습니다.

REXX/CICS가 제공하는 FLST 및 EDIT 명령은 클라이언트/서버 환경의 예입니다.

## REXX/CICS 클라이언트 exec 예제

```
/* 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 서버 exec 예제

```
/* 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
    그 외의 경우 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'
exit
```

## 제 29 장 REXX/CICS 패널 기능

REXX 패널 기능은 REXX 프로그래머에게 패널 정의 및 3270 유형 터미널에 대한 패널 입출력(I/O)을 위한 단순 도구 및 명령을 제공합니다.

패널 기능을 사용하면 임의의 편집기를 사용하여 패널을 쉽게 정의할 수 있습니다. 요구사항은 패널 소스 정의 파일이 추가로 처리되기 전에 RFS(REXX File System)에 있어야 한다는 것입니다. 패널 입출력(I/O) 명령은 패널 정의 기능을 통해 정적으로 정의된 다수의 필드 속성을 동적으로 변경할 수 있는 기능을 REXX 프로그램 내에서 제공합니다. 다음 예제는 패널 기능의 기능을 보여주고 이 정보에서 설명하는 일반 개념에 대한 이해 및 시각화를 돕습니다. 이 예제의 개요는 다음과 같습니다.

- 패널 필드의 특성을 설정하는 필드 제어 문자를 정의합니다.
- 패널 레이아웃을 정의하기 위해 필드 제어 문자를 사용합니다. 이 제어 문자 정의 및 패널 정의(두 파트를 함께 패널 소스라고 함)는 REXX 파일 시스템에 저장됩니다.
- REXX 프로그램에서 패널을 전송하거나 수신하는 데 사용되는 패널 오브젝트를 생성하기 위해 패널 소스를 사용합니다.

### 패널 정의의 예제

```
** 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.

```

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ %. [CICS 문서에서 사용하는 규칙과 용어](#)도 참조하십시오.

## 패널 정의

패널 정의에는 두 개의 단계가 필요합니다.

1. 편집기를 사용하여 REXX 파일 시스템에서 패널 소스 파일을 작성하십시오. 패널 소스는 필드 제어 문자 및 패널 레이아웃을 포함해야 합니다. 패널 레이아웃은 필드 제어 문자, 일반 표시 가능한 텍스트 문자를 포함하며 임베디드 변수 이름도 포함할 수 있습니다.
2. 패널 입출력(I/O) 명령에 사용할 수 있는 중간 양식(패널 오브젝트)으로 패널 소스를 변환하십시오. 패널 기능은 해당 패널을 참조하고 있는 입출력(I/O) 명령이 처음 호출될 때 또는 REXX 환경의 명시 명령(또는 이 명령을 포함하는 REXX exec)을 호출하여 중간 파일을 생성할 수 있는 경우 자동으로 중간 파일을 생성합니다.

**참고:** 패널 오브젝트를 찾을 수 없을 때마다 또는 패널 소스에 패널 오브젝트보다 이후인 날짜 또는 시간 변경이 있는 경우 이 자동 생성이 실행됩니다. 패널 소스를 변경하면 새 패널 오브젝트가 작성됩니다. 따라서 해당 패널을 사용하는 프로그램이 테스트 단계를 벗어난 후에는 패널 소스를 변경할 때 주의해야 합니다. 프로젝트가 프로덕션 단계로 이동한 후에 패널 소스를 RFS 밖 또는 프로그램이 액세스할 수 없는 RFS 디렉토리로 이동하는 것이 유리합니다.

## .DEFINE verb를 사용하여 필드 제어 문자 정의

필드 제어 정의는 패널에서 필드의 속성을 정의합니다.

이러한 제어 문자는 .DEFINE verb를 사용하여 정의할 수 있으며 패널 레이아웃에 선행해야 합니다. .DEFINE verb는 연속 문자(십표)가 행의 마지막 문자인 경우가 아니면 '행의 끝'에서 종료됩니다. 연속 문자는 .DEFINE verb 바로 뒤에 올 수 없습니다. 십표가 정의되고 있는 제어 문자인지 또는 연속 문자인지 모호하기 때문입니다.

공백은 각 키워드를 구분하고 순서는 정의된 제어 문자가 verb 바로 뒤에 와야 한다는 점만 제외하고 중요하지 않습니다.

1열에서 .DEFINE으로 시작되지 않는 텍스트가 있는 경우 행이 연속이 아니면 이 텍스트는 무시되고 주석으로 처리됩니다.

총 32개의 제어 문자(기본 제어 문자 포함)를 한 번에 활성화로 정의할 수 있습니다.

DROP 키워드를 사용하여 모든 문자를 삭제하는 경우 5개의 기본 제어 문자가 다시 활성화됩니다.

특정 키워드 조합은 호환 가능하지 않고 허용되지 않습니다. 의미가 없는 것으로 표시될 수 있는 다른 문자는 허용됩니다(예: INVISIBLE 및 색상). 이 속성은 REXX 프로그램에서 필드 속성이 동적으로 변경되는 경우(감춰진 필드를 표시 필드로 설정할 수 있음)에 유용할 수 있습니다.

.DEFINE verb는 다음의 5가지 특성을 가집니다.

- 첫 번째 열에서 시작되고 그 뒤에 공백이 오고 그 다음에 대문자화되어야 합니다.
- 연속 문자(십표)가 사용되지 않으면 '행 종료'를 종료합니다.
- 모든 키워드에는 최소 2자 약어가 있습니다.



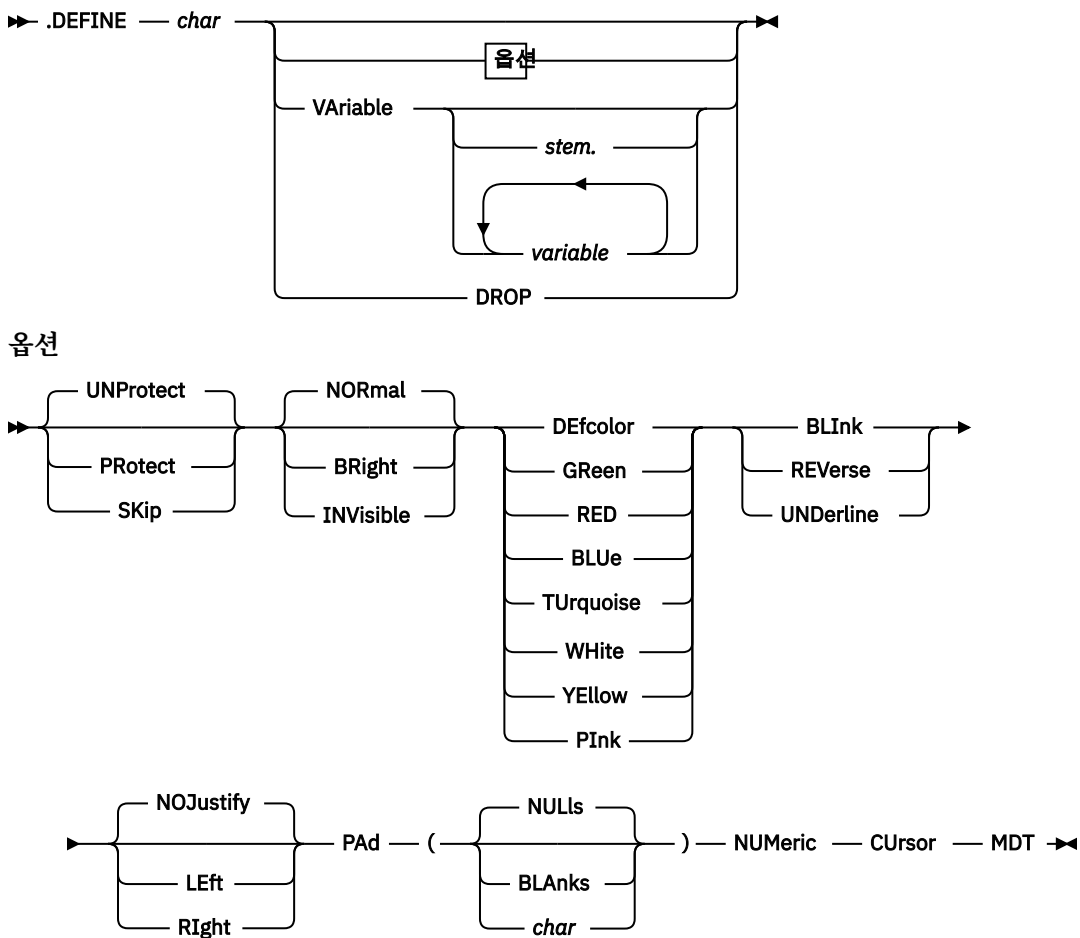
- 한 번에 정의할 수 있는 최대 제어 문자 수는 32입니다. (기본 제어 문자는 이 계수에 포함됩니다.)
- 이 파일은 .PANEL verb 뒤에 배치되어야 합니다.

또한 REXX 변수와 패널 기능 변수를 연관시킬 수 있게 하는 변수 ID 제어 문자를 정의할 수 있습니다. 그런 다음 패널 정의에 REXX 변수 이름 대신 변수 ID 제어 문자를 임베드할 수 있습니다. 동일한 REXX 변수를 다른 변수 ID 제어 문자에 지정할 수 있습니다.

- 스템 이름은 마침표로 끝나야 합니다.
- 변수 목록은 사용되는 것보다 많은 변수를 나열되도록 할 수 있습니다.
- 변수 목록은 사용되는 것보다 적은 변수를 나열되도록 할 수 없습니다.
- 다중 변수 ID 제어 문자를 정의할 수 있고 각각은 서로 독립적입니다.

## .DEFINE

사용자 고유의 제어 문자를 정의하지 않으려는 경우 지정되는 기본 제어 문자와 함께 .DEFINE verb의 형식이 표시됩니다.



### 기본 필드 제어 문자

- # Defcolor 건너뛰기 보통
- + Defcolor 보호 밝음
- % Defcolor 비보호 보통

!

Defcolor 비보호 밝음

&

변수 ID

## 피연산자

### **char**

정의되는 제어 문자를 지정합니다.

### **VARiable**

REXX 변수 ID 제어 문자를 정의합니다. 변수 ID 제어 문자는 패널 기능 제어 문자와 REXX 변수 이름을 연관시키는 데 사용됩니다. 둘 이상의 변수 제어 문자를 한 번에 정의할 수 있습니다. VARIABLE 키워드 뒤에는 변수 이름 목록(*variable*) 또는 단일 스템 이름(*stem.*)이 올 수 있습니다. 변수 목록은 1 - 32,767개의 변수 이름을 포함할 수 있습니다. 한 개의 스템 이름만 지정할 수 있고 스템 이름은 마침표로 끝나야 합니다. 이 마침표는 변수를 스템으로 식별하고 마침표를 빼면 스템 이름이 단순 변수로 해석될 수 있습니다.

변수 목록과 스템 이름은 혼합하여 사용할 수 없습니다. 패널 생성기에서 변수 제어 문자가 발생하면 대체가 수행됩니다. 단순 변수 목록은 나열된 순서와 같은 순서로 대체됩니다. 예를 들어, 세 번째 변수 제어 문자는 해당 제어 문자에 대해 나열된 세 번째 변수로 대체됩니다. 스템 변수는 3자로 된 숫자(후미)를 스템 이름에 추가하여 대체됩니다. 이 숫자는 1에서 시작되고 해당 스템 제어 문자가 발생할 때마다 증분됩니다. 따라서 특정 스템의 10번째 스템 제어 문자의 후미는 10이 됩니다(STEM.10). 이러한 변수는 REXX 변수이므로 REXX 변수 이름 지정 규칙을 따라야 합니다.

### **DROP**

필드 제어 문자로 *char*을 삭제합니다.

## 옵션

### **UNProtect**

필드가 연산자 입력으로부터 보호되지 않도록 지정합니다. (기본값입니다.)

### **PProtect**

필드가 연산자 입력으로부터 보호되도록 지정합니다.

### **SKip**

자동 건너뛰기 기능을 사용하여 보호되는 필드를 지정합니다. 이전 비보호 필드의 마지막 위치에 문자를 입력하는 연산자로 인해 커서가 이 필드를 건너뜁니다.

### **NORmal**

필드가 강조표시되지 않도록 지정합니다. (기본값입니다.)

### **BRight**

필드가 강조표시되도록 지정합니다.

### **INVisible**

필드가 표시되지 않도록 지정합니다.

### **GReen**

### **RED**

### **BLUe**

### **TURquoise**

### **WHite**

### **YELlow**

### **PInk**

### **DEfcolor**

색상의 선택사항입니다.

### 참고:

1. 기본 색상을 지정하지 않으면 색상은 필드 유형 및 강도 값을 기반으로 합니다. 보호/보통은 파란색을 표시하고 보호/밝음은 흰색을 표시하고 비보호/보통은 초록색을 표시하고 비보호/밝음은 빨간색을 표시합니다.

2. 패널의 필드가 명시적으로 색상을 지정한 경우(DEFCOLOR 포함) DEFCOLOR가 지정되거나 색상이 지정되지 않은 모든 밝은 필드는 흰색으로 표시되고 DEFCOLOR가 지정되거나 색상이 지정되지 않은 모든 보통 필드는 초록색으로 표시됩니다. 이는 3270 하드웨어 제한사항이고 패널 기능이 아닙니다.

#### **BLInk**

필드가 깜박이도록 지정합니다.

#### **REVerse**

필드가 되돌리기 비디오 상태임을 지정합니다.

#### **UNDERline**

필드에 밑줄이 그어지도록 지정합니다.

#### **NOJustify**

자리맞춤이 수행되지 않도록 지정합니다(왼쪽으로 자리를 맞추지만 공백을 제거하지 않음).

#### **LEft**

필드가 왼쪽으로 자리를 맞추도록 지정합니다(선행 공백이 제거됨).

#### **RIght**

필드가 오른쪽으로 자리를 맞추니다(후미 공백이 제거됨).

#### **PAd()**

변수가 있는 필드의 컨텍스트에서만 지정됩니다. 비보호 필드에서 채움 문자는 변수 값으로 채워지지 않은 문자 위치를 채웁니다. 보호 필드의 채움 문자는 비보호 필드의 채움 문자와 비슷하지만 채움 영역의 범위가 비보호 필드에서와 같이 전체 필드가 아닙니다. 변수가 시작되는 위치에 의해 바인드되며 보호 필드 내에서 필드의 끝 또는 다음 변수 또는 텍스트의 시작까지 채웁니다.

#### **NULLs**

필드가 널 문자로 채워지도록 지정합니다.

#### **BLAnks**

필드가 공백으로 채워지도록 지정합니다.

#### **char**

필드를 채우는 데 사용될 단일 문자를 지정합니다.

#### **NUMeric**

필드가 숫자임을 지정합니다(비보호 필드에만 해당됨).

#### **CURsor**

커서가 필드의 시작부에 위치하도록 지정합니다. 여러 커서 필드가 정의되는 경우 정의된 마지막 커서 필드에 커서가 포함됩니다. 커서는 커서 필드가 정의되지 않은 경우 맨 위 왼쪽 모서리에 배치됩니다.

#### **MDT**

필드에 대해 수정 비트 태그를 설정합니다. 필드가 연산자에 의해 수정되지 않았더라도 읽기 시 항상 이 필드를 리턴합니다.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ ¢. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

## **.PANEL verb를 사용하여 실제 PANEL 레이아웃 정의**

패널 레이아웃의 시작 신호를 보내는 .PANEL verb는 .DEFINE verb를 따릅니다.

이 verb도 1열에서 시작되고 대문자화되어야 합니다. 표시되게 할 위치와 동일한 위치에서 패널 정의의 행을 지정하십시오. 패널 verb 뒤에 입력된 모든 텍스트는 빈 행을 포함하여 패널 생성기에 대해 유효하고 따라서 주석 행은 허용되지 않습니다. 첫 번째 문자는 보호, 건너뛰기 또는 비보호 제어 문자여야 합니다. 패널 정의는 파일의 끝 또는 1열에서 시작된 다음 .PANEL verb에서 끝납니다. 파일당 한 패널 정의만 지원됩니다.

패널 레이아웃은 표시되는 내용과 비슷하지만 패널이 표시될 때 표시되지 않는 임베디드 변수와 제어 문자는 제외됩니다. 10열에서 시작된 .PANEL 뒤 3번째 행에 입력된 필드는 터미널 화면 3번째 행, 10열에 위치합니다.

.PANEL verb는 다음과 같은 특성을 가집니다.

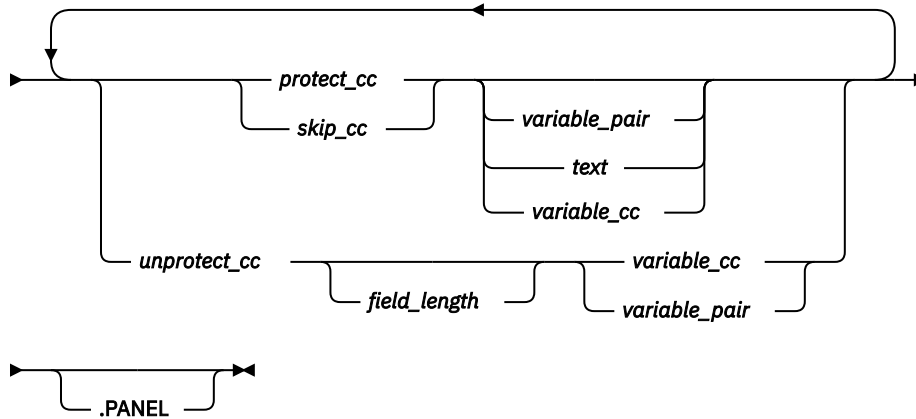
- 첫 번째 열에서 시작되고 그 뒤에 공백이 오고 대문자화되어야 합니다,
- 패널 표시기의 끝인 경우가 아니면 .PANEL과 같은 행에 패널 이름이 있어야 합니다.

- 한 개 이상의 필드가 있어야 하고 .PANEL 행 뒤 첫 번째 문자는 보호, 건너뛰기 또는 비보호 제어 문자여야 합니다.
- 명시 입력 필드 길이가 사용되고 있지 않는 한 필드는 다음 필드의 시작에서 끝납니다. 필드를 종료하기 위해 비어 있는 필드를 사용할 수 있습니다.
- 제어 문자를 일반 표시 가능한 문자로 사용하려면 두 개의 연속 제어 문자를 입력하십시오. 두 개의 연속 제어 문자를 표시하려면 4개의 연속 제어 문자를 입력하십시오. 한 문자로 표시되는 각 제어 문자 쌍의 경우 각 제어 문자 쌍 뒤에 7개 이상의 연속 공백이 있는 경우를 제외하고 필드에서 후속 텍스트는 왼쪽 다음 위치에 표시됩니다.
- 보호 또는 건너뛰기 필드에는 해당 필드에 맞는 수의 변수 또는 텍스트가 있을 수 있습니다. 보호 필드의 변수는 대체에 사용 가능한 영역을 포함하고 변수 ID 제어 문자에서 시작되며 다음 제어 문자 또는 텍스트 바로 앞에서 끝납니다.
- 변수 ID 제어 문자와 변수 이름 사이에는 공백이 허용되지 않습니다. 공백으로 인해 변수 이름이 일반 텍스트로 해석됩니다.
- 비보호 필드에는 한 개의 연관된 변수만 있을 수 있습니다.
- 비보호 필드에는 필드 길이를 명시적으로 표시하는 숫자가 있을 수 있습니다. 이 숫자는 비보호 제어 문자와 변수 제어 문자 사이에 있어야 합니다. 명시 필드 길이가 사용되는 경우 필드는 종료가 필요하지 않습니다. 명시 길이 필드가 행의 마지막 필드인 경우 종료 필드는 필드가 종료되도록 강제 실행하는 건너뛰기 속성을 사용하여 작성됩니다.
- 명시 입력 필드 길이가 사용되는 경우 명시 길이 필드와 바로 뒤 필드가 서로 인접하도록 적절한 맞추기를 강제 실행하기 위해 다음 필드 열 위치 모두 왼쪽 또는 오른쪽으로 조정됩니다. 해당 행의 다른 필드는 그 간격을 원상태로 유지합니다. 이 맞추기는 해당 한 행에만 영향을 미치고 다음 행의 필드는 영향을 받지 않습니다. 예를 들어, 한 행에 5개의 필드가 있고 그 중 첫 번째 및 네 번째 필드에는 명시 길이 및 간격이 있습니다. 필드 1과 2 사이는 4이고, 2와 3 사이는 5이고, 3과 4 사이는 6이며 4와 5 사이는 7입니다. 이 행이 표시되면 필드 1 바로 뒤에서 필드 2가 시작되고(공백 없음) 필드 2와 3 사이의 간격은 여전히 5이고 필드 3과 4 사이의 간격은 여전히 6이고, 필드 5는 필드 4 바로 뒤에 옵니다(공백 없음). 명시 길이로 인해 현재 필드 또는 후속 필드가 다음 필드로 오버플로우되는 경우 패널이 표시될 때 오류가 리턴됩니다.
- 필드가 같은 행에서 종료되지 않으면(예: 필드가 여러 행에 걸쳐 있음) 필드 길이는 패널이 표시되고 있는 화면의 너비에 따라 달라집니다. 또한 마지막 필드에 종료기가 없는 경우 해당 필드는 첫 번째 필드 시작이 발생할 때까지 화면에서 줄을 바꿉니다.
- 변수를 포함하는 필드만 해당 속성을 패널 출력 중 동적으로 변경되게 할 수 있습니다.
- 보호 또는 건너뛰기 필드가 비보호 필드로 동적으로 변경되는 경우 필드의 첫 번째 변수만 연산자 입력이 지정되도록 합니다. 입력 필드의 모든 콘텐츠가 지정됩니다.
- 패널 소스 파일을 런타임 패널 기능이 사용하는 중간 파일(패널 오브젝트)로 처리하려면 우선 이 파일이 REXX 파일 시스템에 있어야 합니다.

## **.PANEL**

.PANEL verb의 형식이 표시됩니다.

➡ .PANEL — *panel\_name* ➡



## 피연산자

### ***panel\_name***

정의되고 있는 패널을 지정합니다. 1 - 8자 길이의 문자여야 하고 REXX 파일 시스템 파일 이름에 대한 규칙을 따릅니다. 267 페이지의 『제 24 장 REXX/CICS 파일 시스템』의 내용을 참조하십시오.

**참고:** *panel\_name*은 RFS 파일 이름과 같아야 합니다. 전체 RFS 파일 이름은 *panel\_name.PANSRC*여야 합니다.

### ***protect\_cc***

보호 필드 제어 문자를 지정합니다.

### ***skip\_cc***

건너뛰기 필드 제어 문자를 지정합니다.

### ***variable\_pair***

뒤에 변수 이름이 오는 변수 제어 문자를 지정합니다. (그 사이에는 공백이 있을 수 없습니다.)

### ***text***

표시 가능한 문자입니다.

### ***variable\_cc***

변수 ID 제어 문자를 지정합니다.

### ***unprotect\_cc***

비보호 제어 문자를 지정합니다.

### ***field\_length***

명시적 입력 필드 길이 값을 지정합니다.

## 패널 생성 및 패널 입출력(I/O)

패널 정의는 REXX 환경 밖에서 수행될 수 있습니다. 그러나 패널 생성 및 입출력(I/O)은 REXX exec 또는 REXX 대화식 환경에서 수행됩니다.

REXX 대화식 환경은 초기 패널 개발을 테스트하기에 이상적인 장소입니다. 패널 표시를 테스트하려면 **TEST** 패널 명령을 사용하십시오. 이는 패널 오브젝트 파일이 작성되지 않은 패널을 표시합니다. 또한 패널의 변수에 대한 대체가 없습니다. 패널 오브젝트를 작성하려면 **GENERATE**, **SEND** 또는 **CONVERSE** 패널 명령을 사용하십시오. **FILE** 키워드를 사용하여 패널 소스를 찾을 RFS의 디렉토리를 명시적으로 설명하거나 현재 디렉토리로 기본값 설정되도록 하십시오.

패널 소스 이름은 파일 이름은 패널 이름이어야 하고 파일 유형은 **PANSRC**여야 합니다. 패널 오브젝트가 작성되고 파일 이름이 소스 파일 이름과 같고 파일 유형이 **PANOBJ**인 패널 소스와 같은 디렉토리에 파일링됩니다. **GENERATE**는 패널 오브젝트를 작성하고 패널을 표시하지 않습니다. **SEND**는 패널 오브젝트를 작성하고 패널을 표시하고 변수 대체를 시도합니다. **CONVERSE**는 내재적 대기 및 수신에 포함된 **SEND**와 비슷합니다.

**참고:** REXX 대화식 환경에 있는 경우 부작용이 있습니다. 여러 패널 키워드가 다르게 작동합니다. SEND에서 커서 위치가 무시되고 키보드 잠금도 SEND 및 CONVERSE에 대해 무시됩니다.

PANEL 명령의 특성은 다음과 같습니다.

- 모든 인수 또는 키워드가 모든 명령에 대해 올바르지 않거나 유의미하지 않습니다.
- REXX exec의 마지막 패널 명령은 END 명령입니다. 이 명령은 이전 패널 명령에 의해 보유된 스토리지를 해제합니다. 이 명령은 다른 피연산자가 필요하지 않습니다.
- 연관된 변수가 있는 필드만 해당 속성을 동적으로 변경되도록 할 수 있습니다.
- 존재하지 않는 경우 패널 오브젝트 파일이 표시되고 있는 패널에 대해 패널 오브젝트 파일이 작성됩니다. 파일 이름은 패널 소스 이름과 같고 파일 유형은 'PANOBJ'입니다.
- 필드가 입력 필드로 변경되는 경우 터미널 연산자에 의해 입력되는 입력은 보호/건너뛰기 필드의 첫 번째 변수에만 지정됩니다.
- 동적 속성 변경사항은 현재 패널이 실행되게 하고 종료되지 않습니다. 후속 패널은 속성이 다시 동적으로 변경되지 않는 한 패널 정의 단계에 의해 정적으로 정의된 속성으로의 되돌리기를 표시합니다.
- SEND는 RECEIVE 전에 수행되어야 하고 패널 이름은 일치되어야 합니다.
- 속성만 변경을 표시하고 다른 항목은 정적으로 정의된 상태로 유지됩니다.
- 위치 인수를 사용하여 전송된 패널은 연산자 입력 REXX 변수에 올바르게 지정되도록 동일한 위치 값으로 수신되어야 합니다.
- 여러 필드가 나열될 때 소괄호를 사용하여 필드 ID 목록을 묶으십시오.
- 각각의 다른 속성 변경에 대해 한 개의 ATTRIBUTE 인수가 있어야 합니다.

**참고:**

- ATTRIBUTE 인수의 수를 동적으로 변경하려면 REXX 변수를 사용해야 합니다(REXX 변수에 리터럴 속성 문자열을 넣고 변수를 사용함). 예를 들어, 프로그램은 연산자 입력에 따라 한 필드를 파란색으로 변경하거나 한 필드를 파란색으로 변경하고 다른 필드를 깜박이도록 변경해야 합니다. 한 가지 솔루션은 다음과 같습니다.

```
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 패널 기능은 오브젝트를 찾을 수 없거나 오브젝트가 소스보다 오래 된 경우 패널 오브젝트를 생성합니다.

## PANEL RUNTIME

PANEL RUNTIME 명령의 형식이 표시됩니다.



## Test

패널을 표시하는 패널 명령입니다. 중간 파일(패널 오브젝트)이 작성되지 않으며 변수 대체가 시도되지 않습니다.

## 종료

패널 세션을 종료하는 명령입니다. 명령은 패널 기능에 의해 보유된 모든 스토리지를 해제합니다. 이 명령에는 인수가 없으며 제공된 모든 인수는 무시됩니다.

## Generate

패널 오브젝트를 작성하는 명시적 명령입니다. 패널이 표시되지 않습니다.

## panel\_name

입출력(I/O)하거나 생성할 패널의 이름을 지정합니다.

## File()

이 패널을 포함하는 RFS 디렉토리의 이름을 지정합니다(directory\_name). (END를 제외한 모든 패널 명령에 대해 지정됩니다.)

## Cursor()

(SEND 및 CONVERSE에 대해서만 지정됨) 패널에서 커서의 위치를 지정합니다.

### field\_id

패널에 커서가 배치되어야 하는 REXX 변수 이름을 지정합니다.

### row

패널 내에서 커서가 배치되어야 하는 행을 지정합니다. 행 값은 패널의 시작 행에 상대적입니다. 패널의 기본 시작 행은 1이지만 POSITION() 키워드를 사용하여 변경될 수 있습니다.

### column

패널 내에서 커서가 배치되어야 하는 열을 지정합니다. 열 값은 패널의 시작 열에 상대적입니다. 패널의 기본 시작 열은 1이지만 POSITION() 키워드를 사용하여 변경될 수 있습니다.

## ATtrib(field\_id attribute\_values)((field\_ids) attribute\_values)

(SEND 및 CONVERSE에 대해서만 지정됨) 패널 정의에서 지정된 속성을 대체할 속성을 동적으로 설정합니다.

### field\_id

속성이 동적으로 설정되는 필드를 지정합니다. 필드와 연관된 변수 이름이어야 합니다. 필드 목록을 지정하는 경우 필드 목록을 소괄호로 묶어야 합니다.

### attribute\_values

설정할 속성을 지정합니다. 명시된 속성만 변경되고 다른 속성은 정적으로 정의된 속성으로 기본값 설정됩니다. 예를 들어, 원래 RED 및 UNDERLINE으로 정의된 필드에 대해 BLUE가 지정된 경우 해당 필드는 BLUE 및 UNDERLINE이 됩니다.

속성 목록은 [315 페이지의 『속성』](#)의 내용을 참조하십시오.

## ALarm

(SEND 및 CONVERSE에 대해서만 지정됨) 패널을 표시할 때 벨 소리가 납니다(기본값은 알람 없음).

## NOErase

(SEND 및 CONVERSE에 대해서만 지정됨) 이 패널을 표시하기 전에 화면을 지우지 않습니다. (기본값은 패널이 쓰기 전에 지우는 것입니다.)

예상치 못한 결과(예: 이 두 번째 패널을 표시하는 동안 기본 패널에서 속성 변경 또는 두 번째 패널의 기본 패널에서 문자 표시)를 방지하려면 다음 가이드라인을 사용하십시오.

- 두 번째 패널에서 변수를 사용하는 경우 기본 패널의 공백 영역에 패널을 배치하십시오.
- 화면의 오른쪽 경계에서 끝나거나 뒤에 기본 패널의 공백 열만 오도록 두 번째 패널을 배치하십시오.

## Position()

(SEND, CONVERSE 및 RECEIVE에 대해서만 지정됨) 출력 화면에 패널을 배치합니다. 행(row) 및 열(column)은 패널의 맨 위 왼쪽 모서리가 시작되어야 하는 위치를 지정합니다. (기본값은 행 1 열 1입니다.) 예를 들어, POS(5 10)의 의미는 행 5 및 열 10에서 패널이 시작되도록 하고 실제로는 아래로 4개 행, 오른쪽으로 9개 열을 이동합니다.



**FReekb**

(SEND 및 CONVERSE에 대해서만 지정됨) 키보드를 해제하여 연산자 입력을 허용합니다. (이는 기본값입니다.)

**LOckkb**

(SEND 및 CONVERSE에 대해서만 지정됨) 키보드를 잠급니다.

**CLrinput**

(SEND 및 CONVERSE에 대해서만 지정됨) 패널을 표시하기 전에 입력 필드를 모두 지웁니다. 변수 대체는 시도되지 않고 채움 문자는 입력 영역을 채웁니다.

**속성****UNProtect**

필드가 연산자 입력으로부터 보호되지 않도록 지정합니다.

**PRotect**

필드가 연산자 입력으로부터 보호되도록 지정합니다.

**SKip**

자동 건너뛰기 기능을 사용하여 보호되는 필드를 지정합니다. 이전 비보호 필드의 마지막 위치에 문자를 입력하는 연산자로 인해 커서가 이 필드를 건너뜁니다.

**NORmal**

필드가 강조표시되지 않도록 지정합니다.

**BRight**

필드가 강조표시되도록 지정합니다.

**INVisible**

필드가 표시되지 않도록 지정합니다.

**GRreen****RED****BLUe****TURquoise****WHite****YELlow****PInk****DEfcolor**

색상의 선택사항입니다.

**참고:**

1. 기본 색상을 지정하지 않으면 색상은 필드 유형 및 강도 값을 기반으로 합니다. 보호/보통은 파란색을 표시하고 보호/밝음은 흰색을 표시하고 비보호/보통은 초록색을 표시하고 비보호/밝음은 빨간색을 표시합니다.
2. 패널의 필드가 명시적으로 색상을 지정한 경우(DEF COLOR 포함) DEF COLOR가 지정되거나 색상이 지정되지 않은 모든 밝음 필드는 흰색으로 표시되고 DEF COLOR가 지정되거나 색상이 지정되지 않은 모든 보통 필드는 초록색으로 표시됩니다. 이는 3270 하드웨어 제한사항이고 패널 기능이 아닙니다.

**BLInk**

필드가 깜박이도록 지정합니다.

**REVerse**

필드가 되돌리기 비디오 상태임을 지정합니다.

**UNDerline**

필드에 밑줄이 그어지도록 지정합니다.

**NOJustify**

자리맞춤이 수행되지 않도록 지정합니다.

입력의 경우 선행 및 후미 공백과 선행 및 후미 채움 문자가 제거되지 않습니다.

출력의 경우 선행 및 후미 공백과 선행 및 후미 채움 문자가 제거되지 않습니다. 필요한 경우 데이터가 오른쪽에서 잘립니다. 잘림으로 인해 실행이 중단되지는 않지만 결과로 리턴 코드 4 및 이유 코드 117이 발생합니다. 채움 문자는 변수 데이터의 오른쪽으로 널 위치를 대체합니다.

#### **LEft**

필드가 왼쪽으로 자리 맞춰지도록 지정합니다.

입력의 경우 선행 및 후미 공백과 선행 및 후미 채움 문자가 제거됩니다.

출력의 경우 선행 공백이 제거되고 변수 데이터의 오른쪽에 있는 널은 채움 문자로 대체됩니다. 선행 공백이 제거된 후에는 필요한 경우 오른쪽에서 데이터가 잘립니다. 잘림으로 인해 실행이 중단되지는 않지만 결과로 리턴 코드 4 및 이유 코드 117이 발생합니다.

#### **RIght**

필드가 오른쪽으로 자리가 맞춰지도록 지정합니다.

입력의 경우 선행 및 후미 공백과 선행 및 후미 채움 문자가 제거됩니다.

출력의 경우 선행 공백이 제거되고 변수 데이터의 왼쪽에 있는 널은 채움 문자로 대체됩니다. 후미 공백이 제거된 후 필요한 경우 데이터는 왼쪽으로 잘립니다. 잘림으로 인해 실행이 중단되지는 않지만 결과로 리턴 코드 4 및 이유 코드 117이 발생합니다. 출력의 경우 채움 문자는 널 및 변수 데이터의 왼쪽에 있는 공백 위치를 대체합니다.

#### **PAd()**

변수가 있는 필드의 컨텍스트에서만 지정됩니다. 비보호 필드에서 채움 문자는 변수 값으로 채워지지 않은 문자 위치를 채웁니다. 보호 필드의 채움 문자는 비보호 필드의 채움 문자와 비슷하지만 채움 영역의 범위가 비보호 필드에서와 같이 전체 필드가 아닙니다. 변수가 시작되는 위치에 의해 바인드되며 보호 필드 내에서 필드의 끝 또는 다음 변수 또는 텍스트의 시작까지 채웁니다.

#### **NULLs**

필드가 널 문자로 채워지도록 지정합니다.

#### **BLanks**

필드가 공백으로 채워지도록 지정합니다.

#### **char**

필드를 채우는 데 사용될 단일 문자를 지정합니다.

#### **NUMeric**

필드가 숫자임을 지정합니다(비보호 필드에만 해당됨).

#### **Cursor**

커서가 필드의 시작부에 위치하도록 지정합니다. 여러 커서 필드가 정의되는 경우 정의된 마지막 커서 필드에 커서가 포함됩니다. 커서는 커서 필드가 정의되지 않은 경우 맨 위 왼쪽 모서리에 배치됩니다.

#### **MDT**

(SEND 및 CONVERSE에 대해서만 지정됨) 패널에서 모든 입력 필드에 대해 수정 비트 태그를 설정합니다.

### **PANEL 변수**

REXX 프로그램이 사용할 수 있고 내재적으로 정의된 PANEL 변수와 PANEL 오류 관련 변수가 나열됩니다.

#### **PAN.AID**

마지막으로 패널 입력을 발생시킨 주의 ID입니다.

ENTER	ENTER
CLEAR	CLEAR
CLRP	CLEAR PARTITION
PEN	SELECTOR PEN
OPID	OPERATOR ID
MSRE	MAGNETIC READER
STRF	STRUCTURE FIELD
TRIG	TRIGGER
PA1	
PA2	
PA3	
PF1PF2PF3PF4PF5PF6PF7	
PF8 PF9PF10PF11PF12PF13	
PF14	
PF15	
PF16	

PF17  
PF18  
PF19  
PF20  
PF21  
PF22  
PF23  
PF24

#### **PAN.CURS**

마지막 패널 입력의 위치입니다. 이는 공백으로 구분된 행 열의 양식입니다. 예를 들어, '10 5'는 행 10 및 열 5가 됩니다. 행 및 열 값은 화면의 시작에 대해 절대적이고 POSITION() 키워드의 영향을 받지 않습니다.

#### **PAN.CNAM**

커서 위치와 연관된 REXX 변수 이름(필드 ID)입니다. 필드에 연관된 변수가 없는 경우 PAN\_CNAM이 업데이트되지 않습니다.

PANEL 오류 관련 변수는 다음과 같습니다.

#### **PAN.REA**

경고 또는 오류가 발생하는 경우의 이유 코드입니다. REXX 리턴 코드인 RC를 먼저 조사해야 합니다. RC가 10인 경우 PAN.REA는 오류 판별에 도움이 되는 상태 코드 및 입력 코드를 포함합니다. 자세한 정보는 319 페이지의 『상태 코드 및 입력 코드』의 내용을 참조하십시오.

#### **PAN.LOC**

내부 위치 코드입니다. IBM 지원 센터에서 사용하는 3자리에서 4자리 숫자입니다. REXX 변수 RC가 값 10을 포함하는 경우 오류 판별을 위해 PAN.LOC를 PAN.REA와 함께 사용하십시오.

#### **PAN.LINE**

소스 패널 정의에서 패널 오브젝트 생성 중에 발생한 오류가 발견된 행 번호입니다.

### **패널 기능 리턴 코드 정보**

패널 명령에 대한 기본 리턴 코드는 REXX 변수 RC에서 설정됩니다. 리턴 코드의 각 레벨은 이유 코드 및 행 번호(해당되는 경우)의 양식으로 된 추가 정보가 수반됩니다.

패널 소스 코드를 처리하는 동안 오류가 발견되고(위치 코드 11xx 또는 12xx) RC가 12 또는 16이 아닌 경우 REXX 변수 PAN.LINE은 오류 상태의 행 번호를 포함합니다.

#### **리턴 코드**

**4**

경고. 패널 기능은 처리를 계속합니다. 다른 리턴 코드 값에 대해서는 처리가 중지됩니다.

**8**

프로그래머 오류입니다.

**10**

프로그래머 오류: PAN.REA에는 오류의 원인을 판별하는 데 도움이 되는 자세한 정보가 포함됩니다. 자세한 정보는 319 페이지의 『상태 코드 및 입력 코드』의 내용을 참조하십시오.

**12**

CICS 명령 오류: CICS EIBRESP가 패널 이유 코드에 리턴됩니다. 오류가 프로그래머가 해결할 수 없는 오류인 경우 필요에 따라 최대한 많은 정보를 저장 및 수집하여 오류를 재작성하고 IBM 지원 센터에 문의하십시오.

**14**

RFS 오류: 이유 코드에 RFS 리턴 코드가 포함됩니다.

**16**

내부 시스템 오류: 필요에 따라 최대한 많은 정보를 저장 및 수집하여 오류를 재작성하고 IBM 지원 센터에 문의하십시오.

#### **시스템 오류 이유 코드**

**401**

명령을 처리하는 동안 패널 기능이 스토리지 공간을 다 사용했습니다.

**402**

내부 제어 문자 ID 테이블 및 제어 문자 정보용 테이블이 동기화되지 않습니다.

**403**

패널 오브젝트 데이터가 손상되었습니다. 먼저 파일이 올바른지 및 패널 오브젝트인지 확인하십시오.

**404**

CICS 수신 버퍼가 손상되었습니다.

**405**

잘못된 유효성 검증 요청입니다.

**406**

스토리지 해제 요청이 실패했습니다.

**407**

스토리지 가져오기 요청이 실패했습니다.

**408**

REXX 변수 가져오기 또는 넣기 시도가 실패했습니다.

**409**

지원을 알 수 없습니다.

**410**

동적 일치 오류입니다.

#### **프로그래머가 소개한 경고 및 오류 이유 코드**

**101**

키워드가 반복되거나 좋아요 카테고리 내 키워드가 반복되었습니다. 예를 들어, RED와 BLUE; UNDERLINE 과 REVERSE.

**102**

키워드가 호환되지 않습니다. 예를 들면, PROTECT와 NUMERIC입니다.

**103**

누락된 키워드 또는 패널 이름입니다.

**104**

정의되고 있는 제어 문자가 올바르지 않거나 누락되었습니다.

**105**

패널이 화면에 비해 너무 큼니다.

**108**

소괄호가 누락되었습니다.

**109**

채움 문자가 올바르지 않습니다.

**110**

태스크에 연관된 터미널이 없습니다.

**111**

패널에 수신 중 정의된 입력 필드가 없습니다.

**112**

패널 이름이 올바르지 않습니다.

**115**

REXX 변수 이름(또는 필드 ID)이 올바르지 않습니다.

**116**

숫자가 올바르지 않게 지정되었습니다. 패널 소스의 명시적 길이 값 또는 또는 행 열 값입니다.

**117**

변수 값이 너무 길어서 출력 필드에 맞게 잘렸습니다.

**118**

텍스트 필드가 잘렸습니다. 명시적 길이가 다른 필드를 오버레이하도록 후속 필드를 강제 실행하지 않았는지 확인하십시오.

- 119** 잘못되거나 누락된 패널 명령입니다. SEND, RECEIVE, CONVERSE, TEST 또는 END여야 합니다.
- 122** 수정된 필드가 수신되었지만 해당되는 입력 필드 정의가 없습니다.
- 124** 비어 있는 수신 버퍼입니다. 지우기, ENTER 및 PA 키로 인해 이 오류 코드가 발생합니다.
- 125** 파일 이름이 올바르지 않습니다.
- 126** ATTRIBUTE 키워드의 필드 ID를 패널에서 찾을 수 없습니다.
- 129** 명령에 너무 많은 인수가 제공되었습니다.
- 130** SEND는 RECEIVE 전에 수행되어야 합니다.
- 131** 패널 소스에 패널 정의가 없습니다. .PANEL 이 열 1에 있지 않거나 뒤에 공백이 없습니다.
- 132** 발생한 키워드가 알 수 없는 키워드입니다.
- 133** DROP이 현재 활성이 아닌 제어 문자를 요청했습니다.
- 136** 연속이 적용되지만 소스의 종료가 발생했습니다.
- 137** 지정된 행 또는 열이 현재 표시장치 CURSOR() 또는 POSITION()에 대해 너무 커서 오류의 원인일 가능성이 높습니다,
- 138** 변수 ID 제어 문자가 정의되었고 둘 이상의 스템 이름이 나열되었습니다.
- 139** 제어 문자 정의에 나열된 것보다 더 많은 수의 변수 필드가 정의되었습니다.
- 140** 명시적 입력 필드 값으로 인해 필드가 화면 끝을 지나쳤습니다.
- 143** 파일의 패널 이름이 패널 런타임 명령의 패널 이름과 일치하지 않습니다.
- 144** 패널에 현재 화면에서 허용하는 수보다 많은 수의 행이 있습니다.
- 145** 패널에 현재 화면에서 허용하는 수보다 많은 수의 열이 있습니다.

## 상태 코드 및 입력 코드

리턴 코드 10의 경우 이유 코드(PAN.REA에서)는 특수한 의미를 가집니다.

이유 코드는 두 개의 2자리 숫자 코드로 구성되며 그 중 첫 번째 숫자는 처리되고 있는 키워드를 명시하고(*state code*) 두 번째 숫자는 오류 조건을 발생시킨 입력을 명시합니다(*input code*). PAN.LOC의 위치 코드를 사용하여 사용할 상태 코드 목록 및 입력 코드 목록을 판별하십시오.

예를 들어, 다음과 같습니다.

```
RC = 10
PAN.REA = 0199
PAN.LOC = 1177
```

11xx 위치 코드에 대해 상태 코드 및 입력 코드 목록을 사용하는 경우 처리되는 키워드는 상태 코드 01에 대해서는 .DEFINE이고 입력 코드 99에 대해서는 알 수 없는 기호를 포함한 입력 코드입니다.

## **11xx 위치 코드에 대한 상태 코드**

**01**

.DEFINE 및 제어 문자

**02**

필드 유형(보호/건너뛰기/비보호)

**03**

색상(빨간색/파란색/초록색/...)

**04**

강도(밝음/보통/볼 수 없음)

**05**

자리맞추기(왼쪽/오른쪽/자리맞추기 안함)

**06**

숫자

**07**

확장된 강조표시(공백/반전/밑줄)

**08**

MDT

**09**

커서

**10**

채움()

**11**

가변적

**12**

삭제

## **20xx 위치 코드에 대한 상태 코드**

**01**

패널 명령(전송/수신/대화/...)

**02**

파일()

**03**

커서()

**04**

위치()

**05**

알람

**06**

지우지 않음

**07**

키보드 잠금(키보드 잠금/키보드 잠금 해제)

**08**

Clrinput

**09**

속성

**10**

필드 유형(보호/건너뛰기/비보호)

**11**

색상(빨간색/파란색/초록색/...)

- 12** 강도(밝음/보통/볼 수 없음)
- 13** 자리맞추기(왼쪽/오른쪽/자리맞추기 안함)
- 14** 숫자
- 15** 확장된 강조표시(공백/반전/밑줄)
- 16** MDT
- 17** 커서
- 18** 채움()
- 19** ATTRIBUTE 인수의 닫는 소괄호

#### **11xx and 20xx 위치 코드에 대한 입력 코드**

- 01** 필드 유형(보호/건너뛰기/비보호)
- 02** 색상(빨간색/파란색/초록색/...)
- 03** 강도(밝음/보통/볼 수 없음)
- 04** 자리맞추기(왼쪽/오른쪽/자리맞추기 안함)
- 05** 숫자
- 06** 확장된 강조표시(공백/반전/밑줄)
- 07** MDT
- 08** 커서
- 09** 채움()
- 10** 가변적
- 11** 삭제
- 12** (사용되지 않음)
- 13** 파일()
- 14** 커서()
- 15** 위치()
- 16** 알람

- 17** 지우지 않음
- 18** 키보드 잠금(키보드 잠금/키보드 잠금 해제)
- 19** Clrinput
- 20** 속성
- 21** ATTRIBUTE 인수의 닫는 소괄호
- 98** 명령의 종료, 추가 피연산자가 예상되었음
- 99** 알 수 없는 기호, 키워드 또는 제어 문자가 예상되었음

#### **12xx 위치 코드에 대한 상태 코드**

- 01** 패널 이름
- 02** 보호/건너뛰기 필드
- 03** 비보호 필드
- 04** 보호/건너뛰기 필드 내 텍스트
- 05** (아직 구현되지 않음)
- 06** 명시적 입력 필드 길이 수
- 07** 비보호 변수
- 08** 보호/건너뛰기 변수

#### **12xx 위치 코드에 대한 입력 코드**

- 01** 표시 가능한 일반 텍스트
- 02** 명시적 길이 수
- 03** 보호 필드 제어 문자
- 04** 비보호 필드 제어 문자
- 05** 변수 제어 문자
- 07** 패널의 끝
- 08** 올바르지 않거나 알 수 없는 입력



## 위치 코드

1000 아래에 있는 수는 주요 프로세서에 있습니다.

### 10xx

패널 생성기 공통 프로세서

### 11xx

.DEFINE verb 프로세서

### 12xx

.PANEL verb 프로세서

### 20xx

패널 런타임 명령 프로세서

### 21xx

동적 속성 해상도 프로세서

### 30xx

출력 3270 데이터 스트림 프로세서

### 40xx

입력 3270 데이터 스트림 및 REXX 변수 지정 프로세서

### 90xx

CICS 인터페이스 프로세서

## 샘플 패널의 예제

### 예제 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
```

### 예제 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
```

### 예제 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.             |+
#|                                     |+
#++-----+++
```

### 예제 4

```
.DEFINE ) protect bright
.DEFINE + drop
.DEFINE & var msg.
A panel to display output dynamic messages.
.PANEL msgbox2
)+-----+##
)|                                     |##
)| &                                 |##
)| &                                 |##
)|                                     |##
)+-----+##
```

### 예제 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

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다. @ # \$ €. [CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.](#)

## REXX 패널 프로그램의 예제

```
/* 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 &lnot;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;
            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;

```

```

    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;
    ELSE
        MONTH = MONTH + 1;

```

```

    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;

```

```

    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.
*****
```





## 제 30 장 REXX/CICS 명령

모든 REXX/CICS 명령에 대한 세부 참조 정보가 제공됩니다. 모든 명령을 위한 리턴 코드 정보는 특별한 REXX 변수 RC의 명령 실행 후 리턴됩니다.

명령 환경 이름 REXXCICS와 함께 이 섹션의 모든 명령을 사용할 수 있습니다. 이는 또한 기본값입니다. 그러나 명령을 정의하는 방법에 따라 더 많은 특정 환경 이름(CICS와 같은)을 대신 사용할 수 있습니다. 다른 명령 환경이 사용 중이기 때문에 명령 환경을 재설정해야 하는 경우, REXX/CICS 명령을 실행하기 전에 다음 명령을 입력하십시오.

```
ADDRESS REXXCICS
```

REXX/CICS는 다음을 제외한 모든 EXEC CICS 명령을 지원합니다.

- 시스템 프로그래밍(SPI) 명령
- Handle Condition
- Handle Aid
- Handle Abend
- Ignore Condition
- Push
- Pop

REXX/CICS 하의 CICS 명령의 구문은 [애플리케이션 개발 참조서](#)에 설명되어 있습니다. REXX 명령을 사용한 기존 EXEC CICS 명령 정의의 사이 매핑은 다음과 같습니다.

- EXEC CICS 대신에, CICS 명령을 위한 접두부로서 CICS를 사용하십시오.
- 모든 데이터 값 필드는 리터럴 문자열 또는 REXX 변수 이름으로서 지정될 수 있습니다.
- 모든 데이터 영역 필드는 REXX 변수 이름으로 지정될 수 있으며, 의도된 데이터를 위한 소스나 대상입니다.
- CICS 명령에서 소스와 대상 필드 모두와 같은 REXX 변수를 사용하지 마십시오. 이를 수행하는 경우, 명령 실행 결과는 예측 불가능합니다.
- LENGTH 옵션을 지정하지 않으면 길이는 자동으로 관련된 REXX 변수 또는 문자열의 길이에서 판별됩니다.
- REXX/CICS로부터 CICS ENQ 명령을 사용하면, LENGTH 매개변수를 사용하십시오. 그렇지 않으면 예측 불가능한 결과가 발생할 수 있습니다.
- NOHANDLE는 모든 CICS 명령에 대해 자동으로 지정됩니다. 각 명령의 실행으로부터의 EIBRESP 값은 REXX 특수 변수 RC에서 리턴됩니다. 또한 EIB 필드는 REXX 변수 DFHEIBLK, EIBRESP, EIBRESP2와 EIBRCODE에 위치합니다.
- 리턴 코드 값에 대한 설명은 [애플리케이션 개발 참조서](#)의 내용을 참조하십시오. 값이 음수인 리턴 코드에 대한 자세한 정보는 401 페이지의 [『제 32 장 리턴 코드』](#)의 내용을 참조하십시오.

### REXX/CICS 명령 매핑에 대한 EXEC CICS의 예

- 비 REXX:

```
EXEC CICS XCTL PROGRAM('PGMA') COMMAREA(COMA) LENGTH(COMAL)
```

- REXX/CICS:

```
"CICS XCTL PROGRAM('PGMA') COMMAREA(COMA)"
```

**참고:** EXEC CICS READ, WRITE 및 DELETE 명령은 REXX/CICS 권한 부여된 명령으로 기본적으로 구현되어 사용을 제어합니다. [권한 부여된 REXX/CICS 명령과 권한 부여된 명령 옵션 및 보안을 참조하십시오.](#)

## ALLOC

ALLOC은 데이터 세트를 데이터 정의 이름과 연관시킵니다.

권한 부여된 명령입니다.

➤ ALLOC — *ddname* — *dsname* — ( — *member* — ) — SHR —  
OLD —

### 피연산자

#### *ddname*

데이터 정의 이름을 지정합니다.

#### *dsname*

완전한 데이터세트 이름을 지정합니다.

#### *member*

데이터 세트에서 멤버 이름을 지정합니다.

#### SHR

데이터 세트는 존재하지만 독점 제어가 필요하지 않습니다.

#### OLD

데이터 세트가 존재하며 독점 제어가 필수입니다.

### 리턴 코드

#### SVC 99에서 지정된 리턴 코드

자세한 정보는 [z/OS MVS Programming: Authorized Assembler Services Guide](#)의 내용을 참조하십시오.

#### 1702

올바르지 않은 피연산자

#### 참고:

- *dsname*은 보안상의 이유로 상위 레벨의 접두부로서 사용자 ID를 포함합니다. 또한 사용자 종료는 데이터 정의 이름과 데이터세트 이름에 추가 보안 처리를 위해 제공됩니다. ALLOC 명령으로 멤버 이름을 남기면, 순차 데이터 세트가 할당됩니다. 성능 상의 이유로, 데이터 세트가 마이그레이션한 경우 할당 요청이 거부당합니다.
- 할당 샘플 엑시트, CICSECX1은 설치가 보안 검사를 수행하게 하도록 이 명령을 제공합니다.

## AUTHUSER

AUTHUSER는 사용자 ID 목록에 권한을 부여합니다.

다음은 권한 부여된 명령입니다.

➤ AUTHUSER — *userid* —

### 피연산자

#### 사용자 ID

권한 부여된 REXX/CICS이 되는 CICS 사인은 사용자 ID입니다.

### 리턴 코드

#### 0

일반 리턴

#### 2602

올바르지 않은 피연산자 또는 피연산자 누락

#### 2621

지정된 사용자 ID의 길이가 올바르지 않음

## 2642

사용자 ID를 저장하는 중에 오류 발생

### 예

```
'AUTHUSER USER2 SYSPGMR'
```

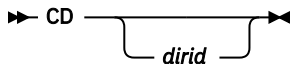
이 예제는 USER2와 SYSPGMR REXX/CICS를 권한 부여된 사용자로 만듭니다.

### 참고:

1. 권한 부여된 사용자인 경우 권한 부여된 REXX/CICS 라이브러리에서 로드된 `exec`에서 실행 중인지 여부와 상관없이 REXX/CICS 권한 부여된 명령을 사용할 수 있습니다.
2. 사용자 ID 목록에서 오류를 발견하면 오류가 있는 사용자 ID는 REXX 특수 변수 `RESULT`에 배치되고 목록 처리는 중지됩니다.
3. `AUTHUSER` 명령은 누적되며 CICS 리전이 재순환될 때까지 이전 `AUTHUSER` 명령 정의는 유효한 상태로 남습니다.

## CD

CD는 RFS 파일 시스템 디렉토리를 변경합니다.



### 피연산자

#### *dirid*

새 현재 작업 디렉토리가 되는 부분 또는 전체 REXX 파일 시스템 디렉토리를 지정합니다.

*dirid*가 지정되지 않은 경우, 현재 작업 디렉토리를 변경하지 않고 현재 작업 디렉토리가 검색되어 REXX 특수 변수 `RESULT`에 배치합니다.

전체 디렉토리 ID의 형식은 *poolid*:*\dirid1*\...\i>diridn입니다.

전체 디렉토리 ID를 지정하면 이전 디렉토리 설정을 완전히 대체합니다.

부분 디렉토리 ID는 *poolid*로 시작하지 않습니다. 이 경우, 부분 디렉토리 ID는 기존 디렉토리 ID의 끝에 추가됩니다. 부분 디렉토리 ID가 두 마침표로 시작되는 경우, 이는 새 부분 디렉토리 ID가 끝에 추가되기 전에 하나의 디렉토리 레벨이 제거되었음(오른쪽에서부터)을 표시합니다. 이런 경우 백슬래시가 디렉토리 ID 앞에 필요합니다.

예를 들어 현재 디렉토리가 `P00L1:\USERS\USER1\ABC`이고 `CD ..\XYZ`를 입력하면, 현재 새 디렉토리는 `P00L1:\USERS\USER1\XYZ`입니다.

사용자를 위한 기본 디렉토리 ID는 *poolid*:*\USERS\userid*이며, 여기서 *poolid*는 정의된 첫 번째 RFS 파일 풀의 파일 풀 ID이며 *userid*는 사용자의 CICS 사인온 사용자 ID입니다. CICS로 사인온하지 않은 경우, *userid*의 기본값은 CICS `DFLTUSER`의 값입니다.

### 리턴 코드

#### 0

정상 리턴

#### 521

파일 풀 정의를 검색하는 중에 오류 발생

#### 522

기본 RFS 디렉토리를 작성하는 중에 오류 발생

#### 523

현재 RFS 디렉토리 정보를 저장하는 중에 오류 발생

**524**

RFS 디렉토리는 존재하지 않거나 액세스 권한이 부여되지 않음

**525**

디렉토리 정보를 검색하는 중에 오류 발생

**526**

올바르지 않은 파일 풀/디렉토리

**527**

지난 루트 디렉토리로 되돌아갈 수 없음

**528**

오류 설정 결과 값

## 예

```
'CD \USERS\USER2\XYZ'
```

이 예는 이전 디렉토리 설정과 상관 없이 현재 작업 디렉토리를 현재 사용 중인 파일 풀의 \USERS\USER2\XYZ로 변경합니다.

현재 디렉토리가 \USERS\USER2이면 다음을 입력하십시오.

```
'CD XYZ'
```

현재 디렉토리는 \USERS\USER2\XYZ로 변경됩니다.

**참고:** CD 명령은 REXX exec의 실행을 위한 검색 순서를 식별하는 PATH 명령과 함께 작동합니다. 현재 디렉토리(CD 명령에서 지정됨)는 항상 exec에 대해 먼저 검색됩니다. 그런 다음, PATH 명령에 나열된 디렉토리 및 MVS 파티션된 데이터 세트가 검색됩니다. 마지막으로, CICS 리전시작 JCL에 할당된 MVS 파티션된 데이터 세트가 검색됩니다.

## CEDA

자원 정의 온라인(RDO)에 대해 CEDA 명령을 실행합니다.

►► CEDA — *RDO\_Command* ◄◄

### 피연산자

#### *RDO\_Command*

CEDA 트랜잭션 프로그램에 대한 입력으로 전달되는 명령 문자열을 지정합니다.

### 리턴 코드

*n*

오류가 발견되면 CICS에 의해 다시 전달되는 리턴 코드입니다.

**0**

일반 리턴

**-101**

올바르지 않은 명령

모든 경고 또는 오류 메시지는 변수 CEDATOUT에 배치됩니다. 실행의 결과(있는 경우)는 변수 CEDAEOUT에 배치됩니다. CEDAEOUT에서 리턴되는 최대 길이는 약 28K 바이트입니다. 각 변수의 형식은 다음과 같습니다.

- 필드의 포함적 길이를 포함하는 2진 반자.
- 생성된 메시지 수를 포함하는 2진 반자.
- 가장 높은 메시지 심각도를 포함하는 2진 반자: 0 및 4는 실행을 계속하고 8 및 12는 실행을 계속하지 않습니다.
- 변수 길이 데이터는 다음을 포함합니다.

- CEDATOUT의 경우: 진단 메시지
- CEDAEOUT의 경우: 메시지 등 CEDA 화면에 일반적으로 표시되는 데이터. 각 행은 줄 바꾸기(NL) 문자로 시작되고 그렇지 않은 경우에는 공백 및 대문자 영숫자 문자로 구성됩니다.

이 데이터의 형식은 릴리스 간 보장되지 않지만 CEDA에 의해 표시되는 형식과 같습니다.

## 예

```
'CEDA INSTALL PROGRAM(XYZ) GROUP(ABC)'
```

이 예제는 실행을 위해 CEDA 트랜잭션 프로그램에 전달되는 CICS 명령을 보여줍니다.

## CEMT

CEMT는 REXX에서 CICS 마스터 터미널 명령을 실행합니다.

➡ CEMT — *master\_term\_cmd* ➡

### 피연산자

#### *master\_term\_cmd*

CEMT 트랜잭션 프로그램에 대한 입력으로 전달되는 명령 문자열을 지정합니다.

### 리턴 코드

*n*

오류가 발견되면 CICS에 의해 다시 전달되는 리턴 코드입니다. REXX에서 CEMT를 호출하는 데에는 DFHEMTA(CEMT 프로그래밍 가능한 인터페이스)가 사용되므로 리턴되는 코드는 CEMT 명령에 대한 코드가 됩니다. 가능한 리턴 코드는 다음과 같습니다.

- 1     찾을 수 없음
- 2     클래스를 찾을 수 없음
- 3     ERROR
- 4     DFH로 시작됨
- 5     변경이 올바르지 않음
- 6     새로 사본을 작성할 수 없음
- 7     권한이 부여되지 않음
- 8     옵션 충돌
- 9     우선순위 > 255
- 10    콘솔에 적용되지 않음
- 11    프로그램을 찾을 수 없음
- 12    올바르지 않은 AUTHID

- 13**      올바르지 않은 ATI / TTI
- 14**      INTRA가 아님
- 15**      열기/전환 실패
- 16**      SDUMP 사용 중
- 17**      성공하지 못함
- 18**      0>=트리거>32767
- 19**      간접에 적용되지 않음
- 20**      0>최대>999
- 21**      추가가 아님
- 22**      열기/닫기가 실패함
- 23**      SDUMP가 억제됨
- 24**      C로 시작됨
- 25**      활성이 아님
- 26**      닫히지 않음
- 27**      SYS LOG에 적용되지 않음
- 28**      이미 존재함
- 29**      카탈로그 입출력(I/O) 오류
- 30**      1>최대 태스크>2000
- 31**      원격에 적용되지 않음
- 32**      NOSTG DSALIMIT
- 33**      0>유효 기간>65535
- 34**      AKP가 시스템에 없음
- 35**      MAXT. < AMAXT.
- 36**      시스템에 없음
- 37**      올바르지 않은 COMAUTHID

- 38** 50>AKP>65535
- 39** 카탈로그가 가득 참
- 40** 2M>DSALIMIT>16M
- 41** 1>MAXACTIVE>999
- 42** 올바르지 않은 DUMPCODE
- 43** 올바르지 않은 DB2ID
- 44** 500>실행. >2700000
- 45** 100>시간>3600000
- 46** 잘못된 트랜잭션 클래스
- 47** 시간< 스캔 지연
- 48** 상한치에 도달함
- 49** 1>MROBATCH>255
- 50** 48M>EDSALIM>2047M
- 51** 달기에 실패함
- 52** BDAM에 적용되지 않음
- 53** 시계가 작동하지 않음
- 54** 올바르지 않은 VTAM®
- 55** 경로에 적용되지 않음
- 56** 열린 파일
- 57** 닫는 중
- 58** 즉시 닫는 중(BEING IMMCLOSED)
- 59** 0>스캔 지연>5000
- 60** SNASVCMG에 적용되지 않음
- 61** 이 태스크에 적용되지 않음

- 62**     사용자의 조건에 적용되지 않음
- 63**     올바르지 않은 MSGQUEUE
- 64**     사용자의 행에 적용되지 않음
- 65**     큐가 사용 안함 설정됨
- 66**     NOSTG EDSALIMIT
- 67**     부분 덤프
- 68**     DDNAME을 찾을 수 없음
- 69**     획득 중
- 70**     강제 닫기 중
- 71**     HOLD PROG에 적용되지 않음
- 72**     로드에 실패함
- 73**     SDUMP가 실패함
- 74**     비어 있거나 닫히지 않음
- 75**     사용 안함으로 설정되지 않음
- 76**     닫기가 요청됨
- 77**     열려 있음
- 78**     사용 불가능함
- 79**     사용 안함으로 설정됨
- 80**     일시 정지 중
- 81**     MSG DFHIR3793 참조
- 82**     시작/전환 실패
- 83**     올바르지 않은 계획
- 84**     올바르지 않은 간격
- 85**     OUT &-REL이 올바르지 않음
- 86**     MSG DFHIR3768 참조



- 87** MSG DFHIR3786 참조
- 88** MAPSET에 적용되지 않음
- 89** MSG DFHIR3771 참조
- 90** 파티션에 적용되지 않음
- 91** MSG DFHIR3773 참조
- 92** 올바르지 않은 DSRTPROGRAM
- 93** MSG DFHIR3775 참조
- 94** MSG DFHIR3776 참조
- 95** MSG DFHIR3777 참조
- 96** MSG DFHIR3778 참조
- 97** MSG DFHIR3779 참조
- 98** MSG DFHIR3780 참조
- 99** MSG DFHIR3781 참조
- 100** MSG DFHIR3791 참조
- 101** VTAM에만 적용됨
- 102** 밖으로 이동
- 103** 숫자 지정
- 104** 숫자 오류
- 105** NEGPOLL이 올바르지 않음
- 106** >20000
- 107** 올바르지 않은 ENDOFDAY
- 108** 최대 | 시스템 종료
- 109** 행 DCB가 열리지 않음
- 110** 올바르지 않은 PLANEXITNAME
- 111** 설정에 실패함

- 112**  
제거에 실패함
- 113**  
올바르지 않은 SIGNID
- 114**  
파일 개수 > 0
- 115**  
올바르지 않은 STATSQUEUE
- 116**  
백아웃에 실패함
- 117**  
올바르지 않은 COMTHREADLIM
- 118**  
>최대
- 119**  
올바르지 않은 PURGECYCLE
- 120**  
인다우트임
- 121**  
4>TCBLIMIT>2000
- 122**  
연결 -ACQD
- 123**  
데이터 세트 일시 정지
- 124**  
진행 중
- 125**  
데이터 세트 사용 불가능
- 126**  
데이터 세트가 일시 정지됨
- 127**  
백업 발생
- 128**  
자원이 누락됨
- 129**  
STATS 누락됨
- 130**  
IS SIT 매개변수
- 131**  
PLT를 로드할 수 없음
- 132**  
올바르지 않은 THREADLIMIT
- 133**  
데이터 세트가 없음
- 134**  
복구가 필요함
- 135**  
영구 제거에 실패함
- 136**  
사용 중인 파일임

- 137**  
BDAM에만 적용됨
- 138**  
중지됨
- 139**  
MSG DFHIR3798 참조
- 140**  
프로그램이 URM임
- 141**  
사용 중
- 142**  
ICE가 사용 중임
- 143**  
PCT가 사용 중임
- 144**  
해제되지 않음
- 145**  
삭제에 실패함
- 146**  
올바르지 않은 RECOVSTAT
- 147**  
고장이 아님
- 148**  
올바르지 않은 PROTECTNUM
- 149**  
올바르지 않은 DB2ENTRY
- 150**  
사용 설정된 종료
- 151**  
올바르지 않은 DTRPROGRAM
- 152**  
지원에서 사용 중임
- 153**  
세션에 적용되지 않음
- 154**  
파이프라인에 적용되지 않음
- 155**  
버릴 수 없음
- 156**  
로컬 시스템이 아님
- 157**  
시스템에 적용되지 않음
- 158**  
모텔에 적용되지 않음
- 159**  
조치가 없음
- 160**  
XLT를 로드할 수 없음
- 161**  
ISC가 정의되지 않음

- 162**  
이미 활성화됨
- 163**  
올바르지 않은 TRANSID
- 164**  
중복 TRANSID
- 165**  
BWO 지원이 없음
- 166**  
DSNB가 올바르게 않음
- 167**  
DSNB BDAM 또는 경로
- 168**  
업데이트 개수 > 0
- 169**  
DSN → SMS가 관리됨
- 170**  
BWO 오류
- 171**  
DFHTMP 오류
- 172**  
사전 설정된 사인온 오류
- 173**  
CPSVCMG에 적용되지 않음
- 174**  
PRM이 사용 불가능함
- 175**  
XLN이 수행된 경우에는 아님
- 176**  
올바르지 않은 PROGAUTOINST
- 177**  
올바르지 않은 PROGAUTOCTLG
- 178**  
올바르지 않은 PROGAUTOEXIT
- 179**  
해제가 완료되지 않음
- 180**  
해제 중임
- 181**  
재사용이 정의됨
- 182**  
블록 해제가 정의됨
- 183**  
0 < 최대값 < 999999999
- 184**  
형식이 VARBLE이 아님
- 185**  
LSRPOOL이 없음
- 186**  
연결 중

- 187**  
올바르지 않은 빈도
- 188**  
0>PURGET.>1000000
- 189**  
올바르지 않은 PSDINT
- 190**  
XRF를 사용하지 않음
- 191**  
SETLOGON 실패
- 192**  
이전 레벨 VTAM
- 193**  
ACB가 닫힘
- 194**  
복구 오류
- 195**  
연기됨
- 196**  
로컬 시스템에 적용되지 않음
- 197**  
MSG DFHIR3799 참조
- 198**  
APPC에만 적용됨
- 199**  
ESM이 비활성임
- 200**  
등록 오류
- 201**  
등록 취소 오류
- 202**  
지원이 취소됨
- 203**  
DB2 대기 중
- 204**  
DB2가 비활성임
- 205**  
올바르지 않은 INITPARM
- 206**  
필요하지 않음
- 207**  
여전히 닫는 중
- 208**  
취소된 지원이 없음
- 209**  
DFSMS 카탈로그 오류
- 210**  
DSNB가 제거됨
- 211**  
RLS 지원이 없음

- 212**  
시스템이 잠김
- 213**  
SDTRAN을 찾을 수 없음
- 214**  
SDTRAN이 사용 안함으로 설정됨
- 215**  
SDTRAN SHUT DIS
- 216**  
DSN → DFSMS VSAM
- 217**  
데이터 세트가 마이그레이션됨
- 218**  
올바르지 않은 유틸
- 219**  
LU61 또는 LU62가 아님
- 220**  
NETID 0이 PRFRM 사용
- 221**  
MSG DFHZC0178 참조
- 222**  
GR이 등록되지 않음
- 223**  
NETID 입력
- 224**  
선호도를 찾을 수 없음
- 225**  
세션이 사용 중임
- 226**  
MSG DFHZC0176 참조
- 227**  
인플라이트 삭제
- 228**  
간접에서 사용됨
- 229**  
IRC가 열림
- 230**  
오류 콘솔임
- 231**  
SDTRAN이 원격임
- 232**  
XRF가 활성이 아님
- 233**  
SYSID가 오류 상태임
- 234**  
오류: 처리 지연된 UOW
- 235**  
MSG DFHZC0173 참조
- 236**  
RLS 및 CMT

- 237**  
연결을 끊는 중
- 238**  
키 길이 오류
- 239**  
레코드 크기 오류
- 240**  
누락된 폴 이름
- 241**  
올바르지 않은 이름
- 242**  
폴을 찾을 수 없음
- 243**  
CONTEN 및 RECOV
- 244**  
올바르지 않은 조치
- 245**  
마지막 사용<간격
- 246**  
대기 중
- 247**  
AUDITLOG가 없음
- 248**  
매개변수 불일치
- 249**  
CFDT 서버가 없음
- 250**  
TCPIP가 닫힘
- 251**  
사용 중인 포트임
- 252**  
권한이 부여된 포트가 아님
- 253**  
올바르지 않은 상태
- 254**  
프로파일을 찾을 수 없음
- 255**  
주소를 알 수 없음
- 256**  
올바르지 않은 Q-TYPE
- 257**  
1>MAXOPENTCBS>2000
- 258**  
JVMCLASS 세트가 없음
- 259**  
JVM 프로그램이 아님
- 260**  
올바르지 않은 JVMCLASS
- 261**  
올바르지 않은 DSNAME

- 262**  
1>MAXSOCKETS>65535
- 263**  
엄격한 한계를 초과함
- 264**  
MAXSOCKET에 있음
- 265**  
TCIPSERVICE가 열리지 않음
- 266**  
SESSBEANTIME > 143999
- 267**  
자원이 서비스 상태가 아님
- 268**  
누락된 CORBASERVER 이름
- 269**  
DJAR이 해결을 보류 중임
- 270**  
올바르지 않은 DB2GROUPID
- 271**  
DB2ID & GROUPID가 입력됨
- 272**  
1>MAXJVMTCBS>999
- 273**  
1>MAXXPTCBS>999
- 274**  
IIOPLISTENER가 아님
- 275**  
DSNAPRH를 찾을 수 없음
- 276**  
MAXOPENTCBS < DB2CONN TCBLIMIT
- 277**  
TCBLIMIT > MAXOPENTCBS
- 278**  
DB2 GROUPID를 찾을 수 없음
- 279**  
DB2 ID를 찾을 수 없음
- 280**  
DJAR 충돌(CORBASERVER 스캔)
- 281**  
JVMPROFILE이 올바르지 않음..프로그램 설정
- 282**  
시작됨
- 283**  
다시 로드되는 중
- 284**  
사용으로 설정되는 중
- 285**  
버려지는 중
- 286**  
시작되지 않음



- 287**  
중지되지 않음
- 288**  
DB2 다시 시작 - 라이트
- 289**  
캐시 크기가 올바르지 않음
- 290**  
TCPIP가 비활성임
- 291**  
강제 영구 제거 시도
- 292**  
1>MAXSSLTCBS>1024
- 293**  
파이프라인이 사용으로 설정되지 않음
- 294**  
누락된 JVMPROFILE
- 295**  
DFHRPL에 대해 올바르지 않음
- 296**  
DFHRPL에 대해 예약된 랭크 10개
- 297**  
범위를 벗어난 랭크
- 298**  
전송 = 0에 대해 획득하지 않음
- 299**  
MAXJVMTCBS가 초과됨
- 300**  
PSTYPE=NOPS 및 PSDI > 0
- 301**  
올바르지 않은 FILELIMIT
- 302**  
올바르지 않은 PROGRAMLIMIT
- 303**  
올바르지 않은 TSQUEUELIMIT
- 304**  
사용 중인 MQNAME
- 305**  
올바르지 않은 MQNAME
- 306**  
MQNAME을 찾을 수 없음
- 307**  
QMGR 대기 중
- 308**  
1>스레드 한계>256
- 309**  
추가 스레드가 없음
- 310**  
스레드가 제한됨
- 311**  
유출 중

**312**

MSG DFHPI2024 참조

**313**

과도한 요소 수

**314**

1M&gt;TSMMAINLIM&gt;32G

**315**

영구 제거 시도 1차

**316**

JVMSERVER 사용 중

**317**

올바르지 않은 REUSELIMIT

**318**

&gt;MEMLIMIT의 25.00%

**319**

번들에 의해 정의됨

**320**

MSG DFHSO0123 참조

**321**

JVM PROG에 적용되지 않음

**322**

MGMTPART에 의해 사용됨

**참고:** 모든 리턴 코드가 모든 CICS 릴리스에서 발생 가능한 것은 아닙니다.**0**

일반 리턴

**-102**

올바르지 않은 명령

모든 경고 또는 오류 메시지는 변수 CEMTTOUT에 배치됩니다. 실행의 결과(있는 경우)는 변수 CEDAEOUT에 배치됩니다. CEMTEOUT에서 리턴되는 최대 길이는 약 28K 바이트입니다. 각 변수의 형식은 다음과 같습니다.

- 필드의 포함적 길이를 포함하는 2진 반자.
- 생성된 메시지 수를 포함하는 2진 반자.
- 가장 높은 메시지 심각도를 포함하는 2진 반자: 0 및 4는 실행을 계속하고 8 및 12는 실행을 계속하지 않습니다.
- 변수 길이 데이터는 다음을 포함합니다.
  - CEMTTOUT의 경우: 진단 메시지
  - CEMTEOUT의 경우: 메시지 등 CEMT 화면에 일반적으로 표시되는 데이터. 각 행은 줄 바꾸기(NL) 문자로 시작되고 그렇지 않은 경우에는 공백 및 대문자 영숫자 문자로 구성됩니다.

이 데이터의 형식은 릴리스 간 보장되지 않지만 CEMT에 의해 표시되는 형식과 같습니다.

CEMTEOUT의 콘텐츠를 구문 분석하기 위해 다음 예제를 사용할 수 있습니다.

```
PARSE VAR CEMTEOUT BUFFLEN 3 MSGCOUNT 5 MAXRC REST
```

이 예제는 다음과 같이 배치합니다.

- 첫 번째 반자(CEMTEOUT의 포함적 길이 포함)를 REXX 변수 BUFFLEN에 배치
- 두 번째 반자(생성된 메시지의 수를 포함)를 REXX 변수 MSGCOUNT에 배치
- 세 번째 반자(가장 높은 메시지 심각도를 포함)를 REXX 변수 MAXRC에 배치
- CEMTEOUT의 나머지를 REXX 변수 REST에 배치

처음 3개의 반자에 포함된 값은 일반적으로 기간으로 표시됩니다. 이러한 값은 인쇄 불가능한 2진 필드이기 때문입니다. 이러한 필드의 값을 표시하려면 해당 값을 변환해야 합니다. 예를 들어, BUFFLEN의 값을 표시하기 위해 다음 중 하나를 사용할 수 있습니다.

- SAY C2X(BUFFLEN)
- VAR1 = C2X(BUFFLEN)  
SAY VAR1

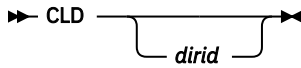
## 예

```
'CEMT SET PROGRAM(XYZ) NEWCOPY'
```

이 예제는 실행을 위해 CEMT 트랜잭션 프로그램에 전달되는 CICS 명령을 보여줍니다.

## CLD

CLD는 현재 RLS 목록 디렉토리를 변경합니다.



## 피연산자

### dirid

사용자를 위한 새 현재 작업 디렉토리가 되는 부분 또는 전체 REXX 목록 시스템 디렉토리를 지정합니다.

*dirid*가 지정되지 않은 경우, 현재 작업 디렉토리가 검색되고 현재 작업 디렉토리 변경하는 대신, REXX 변수 RESULT에 배치됩니다.

전체 디렉토리 ID는 슬래시로 시작하고 형식은 \*dirid1*\...\*diridn*입니다.

전체 디렉토리 ID를 지정하면 이전 디렉토리 설정을 완전히 대체합니다.

부분 디렉토리 ID는 슬래시로 시작하지 않습니다. 이 경우, 부분 디렉토리 ID는 기존 디렉토리 ID의 끝에 추가됩니다. 부분 디렉토리 ID가 두 마침표로 시작되는 경우, 이는 새 부분 디렉토리 ID가 끝에 추가되기 전에 하나의 디렉토리 레벨이 제거되었음(오른쪽에서부터)을 표시합니다. 이런 경우 백슬래시가 디렉토리 ID 앞에 필요합니다.

예를 들어 현재 디렉토리가 \USERS\USER1\ABC이고 CLD ..\XYZ를 입력하면, 현재 새 디렉토리는 \USERS\USER1\XYZ입니다.

사용자를 위한 기본 디렉토리 ID는 \USERS\*genid*이며, 여기서 *genid*는 사용자의 CICS 사인온 사용자 ID입니다. CICS로 사인온하지 않은 경우, *genid*의 기본값은 DFLTUSER의 값입니다.

## 리턴 코드

0

정상 리턴

923

현재 RLS 디렉토리 정보를 저장하는 중에 오류 발생

924

RLS 디렉토리가 없거나 액세스 권한이 부여되지 않음

925

디렉토리 정보를 검색하는 중에 오류 발생

926

올바르지 않은 디렉토리

927

지난 루트 디렉토리로 되돌아갈 수 없음

928

오류 설정 결과 값

예

```
'CLD \USERS\USER2\XYZ'
```

이 예는 이전 디렉토리 설정과 상관 없이 현재 작업 목록 디렉토리를 \USERS\USER2\XYZ로 변경합니다.

현재 디렉토리가 \USERS\USER2이면 다음을 입력하십시오.

```
'CLD XYZ'
```

현재 디렉토리는 \USERS\USER2\XYZ로 변경됩니다.

참고:

1. 현재 디렉토리(CLD 명령으로 지정됨)는 항상 첫번째로 검색되며 적절한 목록 이름으로 RLS 목록을 찾으려고 시도합니다.
2. 완전한 RLS 파일 이름은 사용자의 디렉토리에 대한 검색을 무시합니다.

## CONVTMAP

CONVTMAP은 MVS 순차 파일 또는 PDS의 멤버를 읽고 DSECT(이전에 어셈블된 BMS 맵에 의해 작성됨)를 구조로 변환한 후 결과를 REXX 파일 시스템 파일에 저장합니다.

►► CONVTMAP — *mvs\_dataset\_name* — *rfs\_fileid* ►►

CONVTMAP에 대한 입력으로 사용되는 BMS 맵은 어셈블러 언어 형식이어야 합니다. 결과로 생성되는 출력 파일은 REXX 파일 구조로 형식화됩니다. [449 페이지의 『제 36 장 BMS\(Basic Mapping Support\) 예』](#)의 내용을 참조하십시오.

피연산자

***mvs\_dataset\_name***

완전한 MVS 데이터 세트 이름을 지정합니다. 데이터 세트는 실제 순차 데이터 세트여야 합니다. 또는 데이터 세트가 PDS인 경우에는 PDS 멤버가 소괄호 안에 지정되어야 합니다.

***rfs\_fileid***

완전한 REXX 시스템 파일 또는 REXX 파일 이름만 지정합니다. 완전한 이름이 제공되지 않으면 파일을 저장하는 데 현재 REXX 디렉토리가 사용됩니다.

리턴 코드

***n***

MVS 데이터 세트 처리 시도로 발생하는 리턴 코드입니다.

**0**

일반 리턴

**-302**

올바르지 않은 피연산자

**-321**

올바르지 않은 입력 레코드

**-322**

출력 파일을 쓰는 동안 RFS 오류 발생

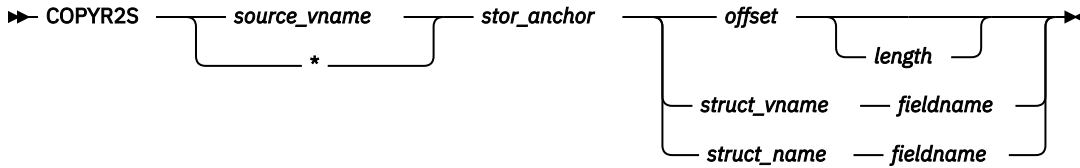
예

```
'CONVTMAP USER1.TEST.DATA(MAP1) P00L1:\USERS\USER1\MAP1.DATA'
```

이 예제는 형식화되어 RFS 파일 POOL1:\USERS\USER1\MAP1.DATA에 기록되는 MVS PDS 데이터 세트 USER1.TEST.DATA의 멤버인 입력 BMS 맵 DSECT, MAP1을 보여줍니다.

## COPYR2S

COPYR2S는 GETMAIN 요청에 의해 이전에 얻은 31비트 스토리지에 REXX 변수 콘텐츠를 복사합니다. 다음은 권한 부여된 명령입니다.



*stor\_anchor*에 의해 주소가 지정되는 계획된 스토리지 영역이 32767바이트를 초과하는 경우 GETMAIN 요청이 FLENGTH 옵션을 지정했는지 확인하십시오.

**참고:** COPYR2S는 31비트 주소에서 작동할 수 있습니다. COPYR2S는 64비트 주소에서 작동할 수 없습니다.

### 피연산자

#### *source\_vname*

GETMAIN 요청에 의해 이전에 얻은 스토리지 영역에 복사된 값을 포함하는 REXX 변수를 지정합니다.

**참고:** 이 값은 대체가 발생하지 않도록 따옴표로 묶어야 합니다.

\*

모든 REXX 변수가 복사되도록 지정합니다. 별표(\*)를 지정하는 경우 *fieldname*을 지정할 수 없습니다.

#### *stor\_anchor*

GETMAIN 요청에 의해 이전에 얻은 대상 스토리지 영역에 대한 앵커를 포함하는 REXX 변수를 지정합니다. 이 앵커는 4바이트로 구성되며 GETMAIN 요청에 의해 얻은 스토리지의 31비트 주소를 포함합니다. 앵커가 31비트 주소를 지정하는지 확인하십시오. 그렇지 않으면 결과가 예측 불가능할 수 있습니다.

#### *offset*

REXX 변수의 콘텐츠가 복사되는 스토리지 영역(GETMAIN 요청으로 얻음)에 대한 변위를 지정합니다. 이 영역의 첫 번째 바이트는 변위 0으로 표시됩니다.

#### *length*

수행되는 복사의 10진수 바이트로 길이를 지정합니다. 이 길이를 지정하면 소스 REXX 변수의 콘텐츠가 잘려서 이 길이를 맞추기 위해 공백으로 채워진 다음 복사됩니다. 그러나 소스 REXX 변수는 이 프로세스에서 변경되지 않습니다. 이 길이가 생략되면 소스 REXX 변수의 현재 길이가 사용됩니다.

#### *struct\_vname*

GETMAIN 요청에 의해 얻은 스토리지 영역에서 필드의 구조 정의(또는 맵핑)를 포함하는 REXX 변수를 지정합니다. 이 변수에서 데이터의 형식은 다음과 같습니다. *field1\_name length ... fieldn\_name length*. 이 기능은 중앙 위치에서 필드 변위를 쉽게 계산하고 변경할 수 있도록 제공됩니다.

#### *struct\_name*

GETMAIN 요청에 의해 얻은 스토리지 영역에서 필드의 구조 정의(또는 맵핑)를 포함하는 구조 파일 ID를 지정합니다.

구조는 다음 형식으로 된 레코드로 구성됩니다. *fieldname location length type*, 여기서,

#### *fieldname*

1 - 12자의 필드 기호 이름을 지정합니다.

#### *location*

구조에서 이 필드가 시작되는 위치를 지정합니다(첫 번째 위치는 1임).

#### *length*

이 필드의 10진수 길이를 바이트 단위로 지정합니다.

#### *type*

필드 데이터 유형 C(문자), F(전자) 또는 H(반자)를 지정합니다.

### **fieldname**

이 사본의 대상 필드와 연관된 1 - 12자의 기호 이름입니다. 이 이름은 위에 지정된 REXX 변수 또는 구조 정의 파일에 있어야 합니다. *fieldname*는 *struct\_vname* 또는 *struct\_name*이 지정될 때 지정되어야 합니다.

### **리턴 코드**

**0**

일반 리턴

**2002**

올바르지 않은 피연산자

**2021**

올바르지 않은 구조 정의

**2022**

올바르지 않은 변수 구조 정의

**2023**

필드 이름을 찾을 수 없음

**2025**

GETVAR 요청 처리 실패

**2026**

올바르지 않은 숫자 입력

**2027**

RFS 읽기 오류

**2028**

올바르지 않은 오프셋

**2029**

올바르지 않은 길이 값

### **예제**

다음 예제는 200바이트의 가상 스토리지를 요청하고 '00000000'x의 16진 값을 해당 영역의 4 - 7바이트로 복사합니다.

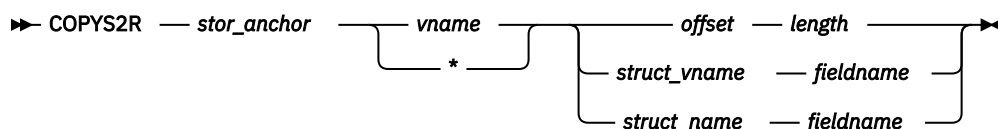
```
/* Needed if entering example from the REXXTRY utility */
'PSEUDO OFF'
'CICS GETMAIN SET(WORKANC) LENGTH(200)'/* get 200 bytes of working storage */
VAR1 = '00000000'x /* set a REXX variable with 4 bytes of hex */
'COPYR2S VAR1 WORKANC 4'
```

다음 예제는 200바이트의 가상 스토리지를 요청하고 문자열 ABC를 앵커 WORKANC가 참조하는 GETMAIN 요청에 의해 얻은 영역의 위치 7에 복사합니다.

```
'CICS GETMAIN SET(WORKANC) LENGTH(200)'/* get 200 bytes of working storage */
VAR1 = 'ABC' /* set a REXX variable with 3 characters */
struct1 = 'flda 4 fldb 2 fldc 3 fldd 8 flde 5'
'COPYR2S VAR1 WORKANC STRUCT1 FLDC'
```

## **COPYS2R**

**I** COPYS2R은 GETMAIN 요청에 의해 이전에 얻은 31비트 스토리지의 데이터를 REXX 변수에 복사합니다. 다음은 권한 부여된 명령입니다.



*stor\_anchor*에 의해 주소가 지정되는 계획된 스토리지 영역이 32767바이트를 초과하는 경우 GETMAIN 요청이 FLENGTH 옵션을 지정했는지 확인하십시오.

**참고:** COPYS2R은 31비트 주소에서 작동할 수 있습니다. COPYS2R은 64비트 주소에서는 작동할 수 없습니다.

## 피연산자

### *stor\_anchor*

GETMAIN 요청에 의해 이전에 얻은 대상 스토리지 영역에 대한 앵커를 포함하는 REXX 변수를 지정합니다. 이 앵커는 4바이트로 구성되며 GETMAIN 요청에 의해 이전에 얻은 31비트 스토리지의 주소를 포함합니다. 앵커가 31비트 주소를 지정하는지 확인하십시오. 그렇지 않으면 결과가 예측 불가능할 수 있습니다.

### *vname*

GETMAIN 요청에 의해 이전에 얻은 스토리지 영역에 복사될 값을 포함하는 REXX 변수를 지정합니다..

**참고:** 이 값은 대체가 발생하지 않도록 따옴표로 묶어야 합니다.

\*

모든 REXX 변수가 복사되도록 지정합니다. 별표(\*)를 지정하는 경우 *fieldname*을 지정할 수 없습니다.

### *offset*

REXX 변수의 콘텐츠를 복사하는 원본 스토리지 영역(GETMAIN 요청으로 얻음)으로의 변위를 지정합니다. 이 영역의 첫 번째 바이트는 변위 0으로 표시됩니다.

### *length*

수행되는 복사의 10진수 바이트로 길이를 지정합니다. 이 길이를 지정하면 소스 REXX 변수의 콘텐츠가 잘려서 이 길이를 맞추기 위해 공백으로 채워진 다음 복사됩니다. 그러나 소스 REXX 변수는 이 프로세스에서 변경되지 않습니다. 이 길이가 생략되면 소스 REXX 변수의 현재 길이가 사용됩니다.

### *struct\_vname*

GETMAIN 요청에 의해 얻은 스토리지 영역에서 필드의 구조 정의(또는 맵핑)를 포함하는 REXX 변수를 지정합니다. 이 변수에서 데이터의 형식은 다음과 같습니다. *field1\_name length ... fieldn\_name length*. 이 기능은 중앙 위치에서 필드 변위를 쉽게 계산하고 변경할 수 있도록 제공됩니다.

### *struct\_name*

GETMAIN 요청에 의해 얻은 스토리지 영역에서 필드의 구조 정의(또는 맵핑)를 포함하는 구조 파일 ID를 지정합니다.

구조는 다음 형식으로 된 레코드로 구성됩니다. *fieldname location length type*, 여기서,

#### *fieldname*

1 - 12자의 필드 기호 이름을 지정합니다.

#### *location*

구조에서 이 필드가 시작되는 위치를 지정합니다(첫 번째 위치는 1임).

#### *length*

이 필드의 10진수 길이를 바이트 단위로 지정합니다.

#### *type*

필드 데이터 유형 C(문자), F(전자) 또는 H(반자)를 지정합니다.

### *fieldname*

이 사본의 대상 필드와 연관된 1 - 12자의 기호 이름입니다. 이 이름은 위에 지정된 REXX 변수 또는 구조 정의의 파일에 있어야 합니다. *fieldname*은 *struct\_vname* 또는 *struct\_name*이 지정될 때 지정되어야 합니다.

## 리턴 코드

0

일반 리턴

### 2102

올바르지 않은 피연산자

### 2121

올바르지 않은 구조 정의

### 2122

올바르지 않은 변수 구조 정의

**2123**

필드 이름을 찾을 수 없음

**2125**

GETVAR 요청 처리 실패

**2126**

올바르지 않은 숫자 입력

**2127**

RFS 읽기 오류

**2128**

올바르지 않은 오프셋

**2129**

올바르지 않은 길이 값

**예**

```
var1 = '' /* set REXX variable VAR1 to null */
struct1 = 'flda 4 flddb 2 fldc 3 fldd 8 flde 5'
'COPYS2R WORKANC VAR1 STRUCT1 FLDC'
```

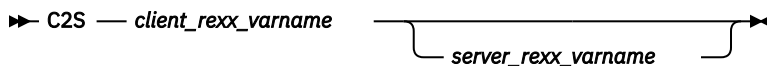
이 예제는 REXX 변수 WORKANC에서 전자 주소에 의해 앵커된 이전 GETMAINed 스토리지 영역의 위치 7 - 9에서 데이터의 3바이트를 복사하고 이를 REXX 변수 VAR1에 복사합니다.

**참고:**

1. 앵커 주소는 GETMAIN 명령으로 설정되는 주소로 제한되지 않습니다. 예를 들어, COPYS2R을 사용하여 GETMAIN 요청에 의해 얻은 영역의 전자 주소를 후속 COPYS2R 또는 COPYR2S에 대한 앵커로 사용되는 REXX 변수에 복사할 수 있습니다.
2. GETMAIN 요청에 의해 얻은 영역에 대해 다중 구조(필드) 정의가 있을 수 있으며 이러한 정의는 오버랩하고, 중첩되고 필드를 재정의하는 데 사용될 수 있습니다.
3. 앵커 주소는 GETMAIN 요청에 의해 얻은 영역의 시작을 가리키지 않아도 됩니다. 그러나 앵커 주소가 사용자가 소유하는 GETMAIN 요청에 의해 얻은 영역 내에 있는 것과 이에 대한 조작이 영역의 경계를 초과하지 않는 것은 중요합니다.

**C2S**

C2S는 클라이언트 REXX 변수를 서버 REXX 변수로 복사합니다.

**피연산자*****client\_rexx\_varname***

복사할 클라이언트 REXX 변수를 지정합니다.

***server\_rexx\_varname***

복사할 대상 서버 REXX 변수를 지정하는 선택적 이름입니다. 지정되지 않은 경우 기본값은 *client\_rexx\_varname*과 동일합니다.

**리턴 코드****0**

정상 리턴

**2440**

지정된 변수 이름

**2441**

오류 검색 변수



## 2442

오류 저장 변수

## 2448

클라이언트 사용 불가능

### 예

```
'C2S VARA VARB'
```

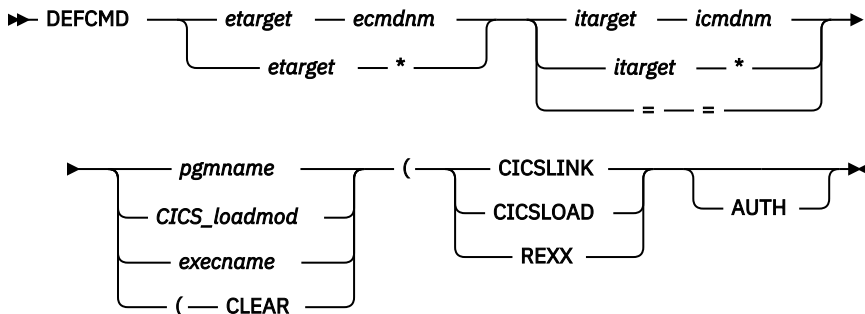
이 예는 클라이언트 REXX 변수 VARA의 콘텐츠가 서버 REXX 변수 VARB로 복사됨을 표시합니다. VARB의 길이는 VARA의 길이와 같습니다.

### 참고:

1. 이 명령에 지원된 최대 변수 이름 길이는 250자입니다.
2. 이 명령은 REXX/CICS 서버 execs에서만 사용하기 위한 것입니다(예를 들어, DEFSCMD 또는 DEFSCMD로 정의된 exec).

## DEFSCMD

DEFSCMD는 REXX 사용자 명령을 정의(또는 재정의)합니다.



### 피연산자

#### etarget

이 명령을 실행하는 REXX exec에 사용했던 외부 대상 환경의 1 ~ 8자의 이름입니다. 이는 명령 문자열을 지시하는 외부 환경 이름입니다. 이 환경 이름은 테이블에서 검색되며 명령 이름과 함께 이 명령 문자열을 처리하도록 지시하는 REXX 프로그램을 판별합니다.

**참고:** REXXCICS가 현재 환경인 경우(기본 조건임), 외부 대상은 ADDRESS 키워드 명령어에서 환경 이름과 일치할 수 있거나 명령 문자열의 첫 번째 토큰으로 지정될 수 있습니다.

#### ecmdnm

이 명령을 실행하는 데 사용된 첫 번째 명령 이름 토큰입니다. 알려진 것처럼 이는 명령 이름의 첫 번째 단어입니다. 별표(\*)의 특수 값이 정의되고(이 정의의 일부인), *etarget*의 환경 이름으로 실행되고 보다 명시적으로 다른 곳에서 정의되지 않은 모든 명령이 이 명령 정의로 처리됩니다. 명령 이름은 최대 16자 길이일 수 있습니다.

#### itarget

이 명령 정의가 명령 문자열을 처리하는 에이전트에 전달하는 내부 환경 이름을 지정합니다. 사용자에게 알려진 외부 환경 이름이 이러한 명령을 처리하는 에이전트의 파손 없이 재정의될 수 있도록 필요합니다. 내부와 외부 이름이 동일한 경우, 내부 이름을 지정할 필요가 없습니다. "="의 특수한 값은 *itarget*이 *etarget*과 동일함을 표시합니다.

#### icmdnm

내부 명령 이름의 첫 번째 단어입니다. 처리되는 명령을 지정하기 위해 REXX 명령 에이전트에 전달된 명령 이름의 첫 번째 파트입니다. *ecmdnm*과 다른 경우에만 지정됩니다. "="의 특수한 값은 *icmdnm*이 *ecmdnm*과 동일함을 표시합니다.

**pgmname**

명령을 처리하기 위해 EXEC CICS LINK에서 호출된 CICS 프로그램을 지정합니다.

**CICS\_loadmod**

CICSLOAD 옵션이 지정되었기 때문에 호출된 CICS 프로그램의 이름을 지정합니다.

**참고:** 프로그램은 명령의 첫 번째 인스턴스에서만 로드되며 해당 주소는 다음 명령에 기억됩니다.

**execname**

이 명령(또는 명령들)을 처리하는 REXX 명령 서버로 호출된 exec를 지정합니다. 이 서버 exec가 이미 실행 중인 경우, 이 명령은 실행 서버로 라우팅됩니다. 이 이름으로 REXX 서버가 실행 중이 아닌 경우 Automatic Server Initiation(ASI)는 자동으로 서버를 시작하는 데 사용됩니다. *execname*은 파일 이름(여기서 파일 유형은 EXEC로 기본 설정됨) 또는 *filename.filetype*의 유형일 수 있습니다.

**CICSLINK**

정의된 REXX 명령에 대한 처리 에이전트가 EXEC CICS LINK에서 호출된 표준 CICS 프로그램임을 표시하는 키워드입니다.

**CICSLOAD**

처리 에이전트가 EXEC CICSLOAD로 로드된 CICS 프로그램임을 표시하는 키워드입니다.

**REXX**

이 REXX 명령에 대해 처리 에이전트가 명령 서버로 작동하는 REXX exec임을 표시하는 키워드입니다.

**AUTH**

권한 부여된 옵션입니다.

권한 부여된 REXX/CICS 명령임을 표시하는 키워드입니다. 권한 부여된 REXX/CICS 사용자(AUTHUSER 명령에서 지정됨) 또는 권한 부여된 라이브러리에서 로드된 exec 내에서만 실행될 수 있는 명령입니다.

**CLEAR**

이 DEFCMD의 용도가 지정된 외부 대상 환경과 명령 이름에 대한 이전 정의를 지우는 것임을 표시하는 키워드입니다.

**리턴 코드****0**

정상 리턴

**1001**

올바르지 않은 명령

**1021**

프로그램을 로딩할 수 없음

**1023**

항목을 찾을 수 없음

**1048**

클라이언트 사용 불가능

**1099**

내부 오류

**예**

```
'DEFCMD CICS SEND = = SENDPGM (CICSLINK'
```

이 예는 이 사용자에게 대해서만 SEND를 호출한 명령을 정의합니다. 사용자는 다음을 입력하여 REXXCICS의 기본 명령 환경 하에 이 명령을 실행할 수 있습니다.

```
'CICS SEND arg1 arg2 ... argn'
```

이 예는 CICS LINK 명령에서 이 명령을 처리하기 위해 호출된 프로그램 SENDPGM을 표시합니다.

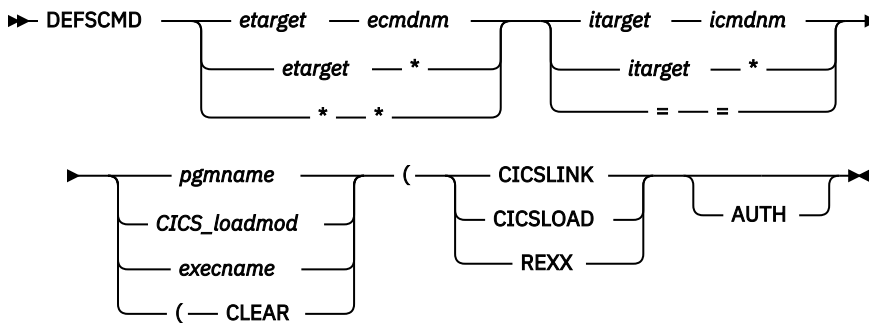
**참고:**

1. REXX/CICS 환경 이름이 REXXCICS인 경우(모든 exec 또는 매크로 호출 시 기본값임), 명령 문자열의 첫 번째 토큰이 ADDRESS 환경 REXX 명령어와 함께 사용된 환경 이름입니다. 이는 더 통합된 명령 환경을 제공하고, ADDRESS 명령어로 전환된 상수 환경에 대한 필요성을 제거합니다.
2. EXEC CICS LINK 및 어셈블러 BASSM 명령어(CICSLOAD 옵션)에서 제어를 수신한 명령 프로그램에 대한 호출 및 매개변수 전달 시퀀스가 유사합니다. 명령 프로그램 작성에 관한 자세한 정보는 [291 페이지의 『제 26 장 REXX/CICS 명령 정의』](#)의 내용을 참조하십시오.
3. DEFSCMD를 사용하여 사용자별 또는 애플리케이션별로 동적으로 설정된 사용자 명령을 사용자 조정할 수 있습니다. DEFSCMD 명령은 사용자의 PROFILE EXEC 또는 애플리케이션 exec에 배치할 수 있습니다. DEFSCMD는 시스템 명령 정의를 대체하는 데 사용될 수도 있습니다.
4. DEFSCMD REXXCICS \*는 허용되지 않습니다.
5. 사용자 명령 정의는 시스템 명령 정의 전에 검색됩니다(대체될 수 없는 DEFSCMD 제외).
6. REXX 명령은 REXX로 작성될 수 있습니다. 이 REXX 명령은 빌딩 블록 방식으로 REXX로 작성된 다른 REXX 명령을 차례로 호출합니다. DEFSCMD가 REXX 사용자(프로그래머)에서 구현 세부사항을 숨기기 때문에, 중요한 성능이 되는 경우 명령은 REXX로 신속하게 작성되며 나중에 다른 언어로 투명하게 다시 작성될 수 있습니다.

## DEFSCMD

DEFSCMD는 REXX 시스템 명령을 정의(또는 재정의)합니다.

권한 부여된 명령입니다.



### 피연산자

#### etarget

이 명령을 실행하는 REXX exec에 사용했던 외부 대상 환경의 이름입니다. 이는 명령 문자열을 지시하는 외부 환경 이름입니다. 이 환경 이름은 테이블에서 검색되며 명령 이름과 함께 이 명령 문자열을 처리하도록 지시하는 프로그램, REXX exec 또는 큐를 판별합니다.

**참고:** REXXCICS가 현재 환경인 경우(기본값임), 외부 대상은 ADDRESS 키워드 명령어에서 환경 이름과 일치할 수 있거나 명령 문자열의 첫 번째 토큰으로 지정될 수 있습니다.

#### ecmdnm

이 명령을 실행하는 데 사용된 첫 번째 명령 이름 토큰입니다. 알려진 것처럼 이는 명령 이름의 첫 번째 단어입니다. 별표(\*)의 특수 값이 정의되고(이 정의의 일부인), etarget의 환경 이름으로 실행되고 보다 명시적으로 다른 곳에서 정의되지 않은 모든 명령이 이 명령 정의로 처리됩니다.

#### itarget

이 명령 정의가 명령 문자열을 처리하는 에이전트에 전달하는 내부 환경 이름을 지정합니다. 사용자에게 알려진 외부 환경 이름이 이러한 명령을 처리하는 에이전트의 파손 없이 재정의될 수 있도록 필요합니다. 내부와 외부 이름이 동일한 경우, 내부 이름을 지정할 필요가 없습니다. "="의 특수한 값은 itarget이 etarget과 동일함을 표시합니다.

#### icmdnm

내부 명령 이름의 첫 번째 단어입니다. 처리되는 명령을 지정하기 위해 REXX 명령 에이전트에 전달된 명령 이름의 첫 번째 파트입니다. ecmdnm과 다른 경우에만 지정됩니다. "="의 특수한 값은 icmdnm이 ecmdnm과 동일함을 표시합니다.

**pgmname**

명령을 처리하기 위해 EXEC CICS LINK에서 호출된 CICS 프로그램을 지정합니다.

**CICS\_loadmod**

CICSLOAD 옵션이 지정되었기 때문에 호출된 CICS 프로그램의 이름을 지정합니다.

**참고:** 프로그램은 명령의 첫 번째 인스턴스에만 로드되며 해당 주소는 다음 명령에 기억됩니다.

**execname**

이 명령(또는 명령들)을 처리하는 REXX 명령 서버로 호출된 exec를 지정합니다. 이 서버 exec가 이미 실행 중인 경우 이 명령은 실행 서버로 라우팅됩니다. 이 이름으로 REXX 서버가 실행 중이 아닌 경우 Automatic Server Initiation(ASI)는 자동으로 서버를 시작하는 데 사용됩니다. *execname*은 파일 이름(여기서 파일 유형은 EXEC로 기본 설정됨) 또는 *filename.filetype*의 유형일 수 있습니다.

**CICSLINK**

정의된 REXX 명령에 대한 처리 에이전트가 EXEC CICS LINK에서 호출된 표준 CICS 프로그램임을 표시하는 키워드입니다.

**CICSLOAD**

처리 에이전트가 EXEC CICSLOAD로 로드된 CICS 프로그램임을 표시하는 키워드입니다.

**REXX**

이 REXX 명령에 대해 처리 에이전트가 명령 서버로 작동하는 REXX exec임을 표시하는 키워드입니다.

**AUTH**

권한 부여된 REXX/CICS 명령임을 표시하는 키워드입니다. 권한 부여된 REXX/CICS 사용자(AUTHUSER 명령에서 지정됨) 또는 권한 부여된 라이브러리에서 로드된 exec 내에서만 실행될 수 있는 명령입니다.

**CLEAR**

이 DEFSCMD의 용도가 지정된 외부 대상 환경과 명령 이름을 위한 이전 정의를 지우는 것임을 표시하는 키워드입니다.

**리턴 코드**

**0**

정상 리턴

**1101**

올바르지 않은 명령

**1121**

프로그램을 로딩할 수 없음

**1123**

항목을 찾을 수 없음

**1148**

클라이언트 사용 불가능

**1199**

내부 오류

**예**

```
'DEFSCMD CICS SEND = = SENDPGM (CICSLINK'
```

이 예는 이 사용자에게 대해서만 SEND를 호출한 명령을 정의합니다. 사용자는 다음을 입력하여 REXXCICS의 기본 명령 환경 하에 이 명령을 실행할 수 있습니다.

```
'CICS SEND arg1 arg2 ... argn'
```

이 예는 CICS LINK 명령에서 이 명령을 처리하기 위해 호출된 프로그램 SENDPGM을 표시합니다.

**참고:**

1. REXX/CICS 환경 이름이 REXXCICS인 경우(모든 exec 또는 매크로 호출 시 기본값임), 명령 문자열의 첫 번째 토큰이 ADDRESS 환경 REXX 명령어와 함께 사용된 환경 이름입니다. 이는 더 통합된 명령 환경을 제공하고, ADDRESS 명령어로 전환된 상수 환경에 대한 필요성을 제거합니다.

2. EXEC CICS LINK 및 어셈블러 BASSM 명령어(CICSLOAD 옵션)에서 제어를 수신한 명령 프로그램에 대한 호출 및 매개변수 전달 시퀀스가 유사합니다. 명령 프로그램 작성에 관한 자세한 정보는 291 페이지의 『제 26 장 REXX/CICS 명령 정의』의 내용을 참조하십시오.
3. DEFSCMD의 처음 두 피연산자가 모두 별표(\*)이면, 추가 특정 명령 정의의 범위 하에 없는(일치하지 않음) REXX 명령에 대해 실행된 명령 처리 에이전트를 지정하는 캐치업 정의입니다.
4. 사용자 명령 정의는 시스템 명령 정의 전에 검색됩니다(대체될 수 없는 DEFSCMD 제외).
5. REXX 명령은 REXX로 작성될 수 있습니다. 이 REXX 명령은 빌딩 블록 방식으로 REXX로 작성된 다른 REXX 명령을 차례로 호출합니다. DEFSCMD는 REXX 사용자(프로그래머)에서 구현 세부사항을 숨기기 때문에, 중요한 성능이 되는 경우 명령은 REXX로 신속하게 작성되어 나중에 다른 언어로 투명하게 재작성될 수 있습니다.

## DEFTRNID

DEFTRNID는 특정 CICS 트랜잭션 ID에 대해 호출될 exec의 이름을 정의하는 데 사용할 수 있는 리전 범위 권한 부여된 명령입니다.

이는 권한 부여된 명령입니다.

➡ DEFTRNID — *trnid* — *execname* — CLEAR

### 피연산자

#### *trnid*

1 - 4자의 CICS 트랜잭션 ID를 지정합니다.

#### *execname*

1 - 17자의 REXX/CICS exec 이름을 *filename.filetype* 양식으로 지정합니다(REXX 파일 시스템에 있는 경우). exec이 MVS 파티션된 데이터 세트에 존재하는 경우 이는 1 - 8자의 이름입니다.

#### CLEAR

이 트랜잭션 ID에 대해 정의가 제거됨을 표시하는 키워드입니다.

### 리턴 코드

#### 0

일반 리턴

#### 1202

올바르지 않은 피연산자

#### 1222

올바르지 않은 옵션

#### 1223

트랜잭션 테이블(trantable) 정보를 저장하는 중에 오류 발생

#### 1225

트랜잭션 테이블(trantable) 정보를 검색하는 중에 오류 발생

#### 1226

Exec 이름 길이 오류

#### 1228

트랜잭션 테이블(trantable) 값을 설정하는 중에 오류 발생

#### 1233

테이블에서 트랜잭션을 찾을 수 없음

### 예

XYZ이라는 이름의 새 CICS 트랜잭션 ID를 정의하고 이를 호출 exec TESTEXEC으로 설정한 후 다음 단계를 완료하십시오.

1. ddname CICUSER에 할당되거나 연결된 데이터 세트에서 TESTEXEC을 작성하십시오.
2. REXX/CICS REXXTRY 유틸리티에서 DEFTRNID XYZ TESTEXEC 명령을 입력하고 REXXTRY 유틸리티를 종료하십시오.
3. RDO에서 CICS용 REXX 개발 시스템에 대해 다음을 입력하십시오.

```
CEDA DEFINE TRAN(XYZ) PROGRAM(CICREXD) GROUP(CICREXX)
```

CICS용 REXX 런타임 기능에 대해 다음을 입력하십시오.

```
CEDA DEFINE TRAN(XYZ) PROGRAM(CICREXR) GROUP(CICREXX)
```

그리고 나서 다음을 입력하십시오.

```
CEDA INSTALL TRAN(XYZ) GROUP(CICREXX)
```

4. 화면을 지우고 CICS 트랜잭션 ID XYZ를 입력한 후 Enter를 누르십시오. 이제 TESTEXEC exec이 실행되어야 합니다.

#### 참고:

1. 일반적으로 DEFTRNID 정의는 REXX/CICS 시동 시 실행되는 CICSTART exec에 배치되어야 합니다.
2. 트랜잭션 ID는 제공된 CICREXD 또는 CICREXR 프로그램을 호출하도록 정의되어야 합니다. 예제에서 [360 페이지의 『3』](#) 단계를 참조하십시오.

## DIR

DIR은 현재 디렉토리 콘텐츠를 표시하거나 선택적으로 REXX 복합 변수에 디렉토리 콘텐츠를 리턴합니다.



#### 피연산자

##### dirid

표시되는 부분적 또는 전체 REXX 파일 시스템 디렉토리를 지정합니다. 이를 생략하면 현재 디렉토리가 표시됩니다.

##### stem.

스텝의 이름을 지정합니다. 스텝은 마침표로 종료되어야 합니다. [144 페이지의 『스텝』](#)의 내용을 참조하십시오. Stem.0은 항목에 요소의 수를 포함합니다. 이를 생략하면 콘텐츠가 화면에 표시됩니다.

#### 리턴 코드

##### 0

일반 리턴

##### 321

현재 RFS 디렉토리 정보에 액세스할 수 없음

##### 322

올바르지 않은 스텝 이름

##### 325

RFS 디렉토리 검색 중 오류 발생

#### 예

```
'DIR \USERS\USER2 (X.'
```

이 예제는 \USERS\USER2의 디렉토리 콘텐츠를 REXX 복합 변수 X.1 - X.n에 배치합니다. X.0은 리턴된 요소의 수를 포함합니다.

참고: 현재 디렉토리는 CD 명령에 의해 지정됩니다.

## EDIT

EDIT은 새 편집 세션을 엽니다.



### 피연산자

#### NONAME

파일 ID가 지정되지 않습니다. 이는 기본값입니다.

#### fileid

편집 중인 파일의 파일 ID입니다.

#### PDS\_name(mem)

완전한 MVS PDS 이름 및 멤버 이름을 지정합니다.

#### PDS

PDS가 편집되고 있을 때 PDS 멤버 이름 뒤에 오는 키워드입니다.

#### MACRO

편집되고 있는 파일에 명령어 그룹이 적용되도록 지정하는 키워드입니다.

#### macroname

프로파일 매크로 파일 ID(EXX exec 이름이)의 파일 이름 부분입니다.

### 리턴 코드

#### 0

일반 리턴

#### 201

올바르지 않은 명령

#### 211

올바르지 않은 파일 ID

#### 226

파일이 현재 편집되고 있음

#### 299

내부 오류

### 예

```
'CD \USERS\USER2\' /* specify current working directory */  
'EDIT TEST.EXEC' /* edit an existing file in the PATH */  
/* or create new file in current dir */
```

이 예제는 디렉토리 \USERS\USER2에서 멤버 TEST.EXEC을 편집합니다.

REXX/CICS 편집기에 대한 자세한 정보는 [239 페이지의 『제 23 장 REXX/CICS 텍스트 편집기』](#)의 내용을 참조하십시오.

## EXEC

EXEC이 하위 레벨의 REXX exec을 호출합니다(중첩된 exec으로). 이 새 exec에 대한 모든 변수는 중첩 exec이 종료될 때까지 일시중지되는 상위 레벨 exec과는 별도로 보존됩니다.

➡ EXEC — *execid* — *args* ➡

## 피연산자

### *execid*

1 - 17자의 exce ID이며 형식은 *name.modifier*입니다. 여기서, *name* 및 *modifier*는 최대 8자입니다. *modifier*를 지정하지 않으면 EXEC이 가정됩니다.

### *args*

호출된 exec에 대한 인수 문자열입니다.

## 리턴 코드

### *n*

호출된 exec에 의해 설정된 리턴 코드입니다. [160 페이지의 『EXIT』](#)의 내용을 참조하십시오.

### 0

일반 리턴

### -3

Exec을 찾을 수 없음

### -10

Exec 이름이 지정되지 않음

### -11

올바르지 않은 exec 이름

### -12

GETMAIN 오류

### -99

내부 오류

## 예

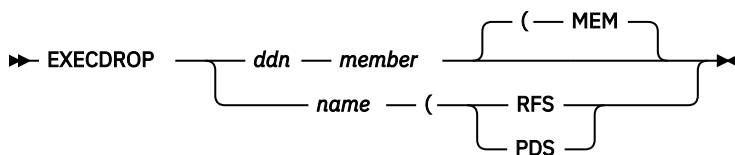
```
'EXEC ABC'
```

이 예제에서는 exec ABC.EXEC을 실행합니다.

## EXECDROP

EXECDROP은 이전에 EXECLOAD 명령을 사용하여 로드된 가상 스토리지에서 exec를 제거합니다.

이는 권한 부여된 명령입니다.



## 피연산자

### *ddn*

읽히는 ddname을 지정합니다.

### *member*

사용되는 파티션된 데이터 세트의 멤버를 지정합니다.

### 이름

완전한 RFS 파일 이름 또는 완전한 PDS 이름 및 멤버를 지정합니다.

### MEM

ddname 및 멤버 이름이 지정되었음을 표시합니다.



## RFS

RFS 파일이 지정되었음을 표시합니다.

## PDS

파티션된 데이터 세트 이름 및 멤버가 지정되었음을 표시합니다.

## 리턴 코드

### 0

일반 리턴

### 1401

올바르지 않은 명령

### 1402

올바르지 않은 피연산자

### 1423

EXECLOAD 정보를 저장하는 중에 오류 발생

### 1425

EXECLOAD 정보를 검색하는 중에 오류 발생

### 1448

사용 가능한 클라이언트가 없음

## 예

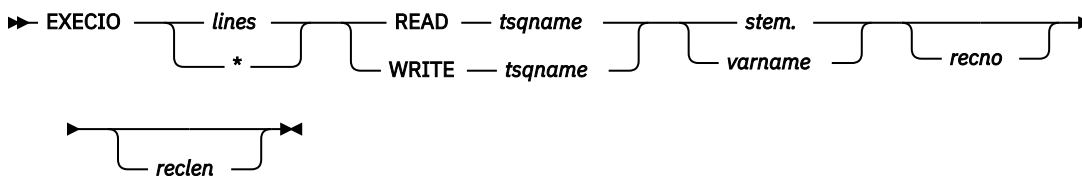
```
'EXECDROP P00L1:\USERS\USER2\TEST.EXEC (RFS'
```

이 예제는 가상 스토리지에서 RFS 파일을 제거합니다.

**참고:** 부분 디렉토리 ID가 제공되는 경우 완전한 디렉토리 ID를 가져오기 위해 이 ID는 임시로 현재 작업 디렉토리 값의 끝에 추가됩니다.

## EXECIO

EXECIO는 CICS 임시 저장영역 큐에 파일 입력 또는 출력을 수행합니다.



## 피연산자

### lines

읽거나 쓸 행 수를 지정합니다. 별표(\*)는 READ 조작에만 지정된 특별한 경우이며, 대상 행(또는 대상 행이 지정되지 않은 경우 1행)부터 파일 끝까지 파일이 읽혀짐을 의미합니다.

### READ

CICS 임시 저장영역 큐(TSQ)에서 하나 이상의 레코드를 읽습니다.

### WRITE

CICS 임시 저장영역 큐(TSQ)에 하나 이상의 레코드를 씁니다.

### tsqname

1 ~ 8자 임시 저장영역 큐 이름을 지정합니다.

### stem.

스텸의 이름을 지정합니다. 스텸은 마침표로 종료되어야 합니다. [144 페이지의 『스텸』](#)의 내용을 참조하십시오.

### **varname**

이 EXECIO 조작을 위한 소스나 대상인 REXX 변수 이름을 지정합니다.

### **recno**

READ 또는 WRITE가 시작되는 임시 저장영역 큐에 레코드 번호를 지정합니다.

### **reclen**

CICS 임시 저장영역에 기록된 레코드의 길이를 지정합니다. *reclen*이 생략되면, 길이는 80 바이트로 기본설정됩니다.

## **리턴 코드**

### **n**

오류가 감지되면 CICS에서 다시 전달되는 리턴 코드

### **0**

정상 리턴

### **-202**

올바르지 않은 피연산자

### **-221**

너무 많은 피연산자가 지정됨

### **-222**

Recno 피연산자가 범위 밖에 있음

### **-224**

올바르지 않은 행 피연산자

## **예**

```
x.1 = 'line 1'  
x.2 = 'Line Two'  
'EXECIO 2 WRITE QUEUE1 X.'
```

이 예는 데이터를 CICS 임시 저장영역 큐에 기록합니다.

```
'EXECIO 2 READ QUEUE1 Y.'  
say y.0 /* ==> 2 */  
say y.1 /* ==> 'line 1' */  
say y.2 /* ==> 'Line Two' */
```

이 예는 임시 저장영역 큐에서 데이터를 읽습니다.

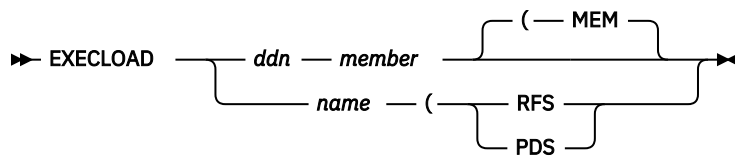
### **참고:**

1. 허용된 레코드의 최대 길이는 256바이트입니다.
2. *stem*이 READ 조작에 지정된 경우(그리고 둘 이상의 레코드가 기록된 경우 *stem*이 지정되어야 함), 읽힌 실제 레코드 수는 *stem.0*에 배치됩니다.
3. CICS 제공 CEBR 트랜잭션을 사용하여 임시 저장영역 큐를 찾습니다. 예를 들어, CEBR QUEUE1을 입력하여 예에서 작성된 큐를 찾습니다.
4. CEBR은 CICS 트랜지언트 데이터 큐와 CICS 임시 저장영역 큐 사이에 복사하는 데 사용할 수 있는 PUT 및 GET 함수를 제공합니다.

## **EXECLOAD**

EXECLOAD는 exec을 가상 스토리지로 로드합니다.

다음은 권한 부여된 명령입니다.



## 피연산자

### **ddn**

읽히는 ddname을 지정합니다.

### **member**

사용되는 파티션된 데이터 세트의 멤버를 지정합니다.

### **이름**

완전한 RFS 파일 이름 또는 완전한 PDS 이름 및 멤버를 지정합니다.

### **MEM**

ddname 및 멤버 이름이 지정되었음을 표시합니다.

### **RFS**

RFS 파일이 지정되었음을 표시합니다.

### **PDS**

파티션된 데이터 세트 이름 및 멤버가 지정되었음을 표시합니다.

## 리턴 코드

### **0**

일반 리턴

### **1501**

올바르지 않은 명령

### **1502**

올바르지 않은 피연산자

### **1523**

EXECLOAD 정보를 저장하는 중에 오류 발생

### **1525**

EXECLOAD 정보를 검색하는 중에 오류 발생

### **1530**

CICPDS 루틴에 링크할 수 없음

### **1531**

CICPDS 루틴에서 오류가 리턴됨

### **1532**

RFS 읽기에서 오류가 리턴됨

### **1532**

PDS 빌드에서 오류가 리턴됨

### **1547**

GETMAIN 오류

### **1548**

사용 가능한 클라이언트가 없음

### **1599**

내부 오류

## 예

```
'EXECLOAD POOL1:\USERS\USER2\TEST.EXEC (RFS'
```

이 예제는 RFS에서 가상 스토리지로 `exec TEST.EXEC`을 로드합니다. `TEXT.EXEC`의 후속 호출은 로드된 사본을 사용합니다.

#### 참고:

1. `exec`은 가상 스토리지로 로드되는 경우 자동으로 모든 사용자에게 의해 공유됩니다.
2. `EXECLOAD`가 수행되어 스토리지에 있는 `exec`을 최신 사본으로 대체하는 경우 이전 사본의 새도우는 이러한 사본을 기반으로 하는 모든 활성 `exec`이 종료될 때까지 가상 스토리지에 보관됩니다. 이는 사용 계수를 통해 이루어집니다.
3. 가상 스토리지에 로드되는 `exec`이 항상 다른 `exec`보다 먼저 사용됩니다. 두 개의 `exec`이 같은 이름을 가지고 이 중 한 `exec`은 RFS 현재 디렉토리에 있고 다른 하나는 가상 스토리지에 로드되는 경우 RFS 사본을 실행할 수 없으므로 주의하여 프로그램의 이름을 지정하십시오.
4. `exec`이 가상 스토리지에 로드되는 경우 `exec`의 사본은 로드된 원래 위치에서 보안 특성을 상속받습니다. 예를 들어, 모든 사용자가 `CICEXEC` 데이터 세트에서 `exec`을 실행할 수 있고 `CICEXEC`에서 로드된 `exec`도 `REXX/CICS` 권한 부여 명령을 사용할 수 있습니다. [CICS 초기화 JCL](#) 수정을 참조하십시오.

## EXECMAP

`EXECMAP`은 `ddname` 및 멤버, 사용자 수, 디스크립터 테이블 시작(16진으로) 및 `EXECLOAD`를 사용하여 로드된 `exec`의 필요한 스토리지 양을 리턴합니다.

►► `EXECMAP` ◀◀

#### 리턴 코드

0

일반 리턴

1623

`EXECLOAD` 디렉토리를 찾을 수 없음

#### 예

```
'EXECMAP'
```

`exec POOL1:\USERS\USER1\TEST.EXEC`을 `EXECLOAD`를 사용하여 이전에 로드한 경우 결과로 생성되는 표시는 다음과 같습니다.

EXEC name	Use	Location	Size	Fully Qualified Name
TEST.EXEC	0	09369083	287	POOL1:\USERS\USER1\TEST.EXEC

## EXPORT

`EXPORT`는 RFS 파일을 MVS 데이터 세트에 내보냅니다.

►► `EXPORT` — *rfs\_fileid* — *dsn* ◀◀

#### 피연산자

*dsn*

완전한(따옴표 없음) MVS 실제 순차 또는 파티션된 조직 데이터 세트 이름을 지정합니다. 데이터 세트가 파티션된 경우(DSORG=PO) 멤버 이름을 소괄호 안에 제공해야 합니다.

*rfs\_fileid*

완전한 REXX 파일 시스템 파일 ID를 지정합니다.

#### 리턴 코드

0

일반 리턴

- 1701**  
올바르지 않은 명령
- 1702**  
올바르지 않은 피연산자
- 1723**  
RFS 쓰기 오류
- 1724**  
RFS 읽기 오류
- 1725**  
동적 할당에 실패함
- 1726**  
동적 해제에 실패함
- 1727**  
트랜지언트 데이터 큐 열기에 실패함
- 1728**  
멤버가 인큐되지 않음
- 1729**  
사용 중인 멤버
- 1730**  
레코드가 잘림
- 1733**  
내보내기를 위한 입력을 찾을 수 없음
- 1735**  
트랜지언트 데이터 오류
- 1736**  
예상치 못한 CICS 오류.
- 1737**  
올바르지 않은 요청
- 1738**  
올바르지 않은 데이터 세트 이름
- 1739**  
올바르지 않은 배치
- 1741**  
지원되지 않는 DSORG
- 1742**  
트랜지언트 데이터 풀을 빌드하는 중에 오류 발생
- 1743**  
REXX 트랜지언트 데이터 큐를 사용할 수 없음/찾을 수 없음
- 1744**  
사용자가 사인온하지 않음/데이터세트 액세스에 대해 권한을 부여받지 못함
- 1745**  
비어 있는 데이터 세트
- 1746**  
REXX 큐를 찾을 수 없음
- 1748**  
TIOT(Task Input/Output Table)에 ddname에 대한 항목이 없음
- 1749**  
여러 단위의 데이터세트에 내보낼 수 없음
- 1750**  
ddname에 연결된 데이터 세트가 둘 이상 있음

**예**

```
'EXPORT POOL1:\USERS\USER1\TEST.DATA USER1.TEST.DATA'
```

이 예제는 RFS 파일에서 MVS 순차 데이터 세트로 POOL1:\USERS\USER1\TEST.DATA를 복사합니다.

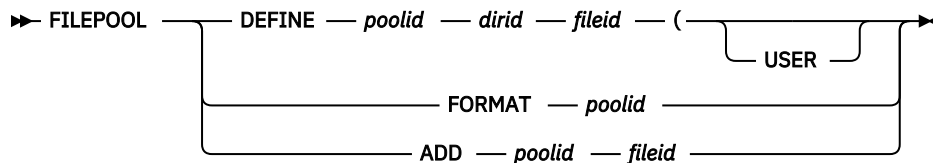
**참고:**

1. 이 명령은 SVC 99를 수행하므로 성능상의 문제로 자주 사용하는 것은 권장되지 않습니다.
2. REXX/CICS 제공 CICSECTX1 보안 엑시트가 호출되어 지정된 MVS 데이터 세트에 액세스할 수 있는 권한이 CICS 사인온 사용자 ID에 부여되어 있는지 확인합니다. 기본적으로 데이터 세트의 상위 레벨 접두부는 CICS 사인온 사용자 ID와 일치해야 합니다. CICS 리전에도 지정된 MVS 데이터 세트(예를 들어, 외부 보안이 사용되는 경우 외부 보안 관리자에 의해 지정됨)에 액세스하기 위한 권한이 있어야 합니다.
3. MVS 데이터 세트에 DISP=OLD가 할당됩니다.
4. dsn에 지정된 MVS 데이터가 이미 할당되어 있어야 합니다. 데이터 세트가 순차인 경우 rfs\_fileid에 지정된 RFS 파일의 콘텐츠로 대체됩니다. 데이터 세트가 파티션되고 지정된 멤버 이름이 이미 존재하면 RFS 파일의 콘텐츠로 대체됩니다.
5. CICS 리전으로 정의되는 MVS 데이터 세트로 내보내는 경우 CICS 리전이 재순환되기 전까지 데이터 세트의 배치는 OLD로 변경됩니다.

**FILEPOOL**

FILEPOOL은 RFS 파일 풀 관리 활동을 수행합니다.

다음은 권한 부여된 명령입니다.

**피연산자****DEFINE**

새 RFS 파일 풀을 정의합니다.

**poolid**

대상 파일 풀의 이름을 지정합니다.

**dirid**

파일 풀 디렉토리의 CICS 파일 ID를 지정합니다.

**fileid**

파일 풀에 대해 VSAM 파일의 CICS 파일 ID를 지정합니다.

**USER**

이는 사용자 파일 풀이고 RFS 보안을 사용하는 파일이며 \USERS 디렉토리가 자동으로 작성되게 함을 표시하는 선택적 키워드이므로 여러 사용자가 RFS 보안을 사용하여(CICS 보안에 비해) 이 파일 풀을 공유하고 디렉토리에 대한 액세스를 제어할 수 있습니다.

**FORMAT**

새 RFS 파일 풀에서 첫 번째 파일의 형식을 지정합니다.

**ADD**

추가 VSAM 파일을 기존 파일 풀에 추가합니다.

## 리턴 코드

**0**

일반 리턴

**1802**

올바르지 않은 피연산자

**1821**

올바르지 않은 파일 풀 하위 명령

**1822**

파일 풀 하위 명령이 지정되지 않음

**1823**

파일 풀 정보를 저장하는 중에 오류 발생

**1824**

파일 풀 ID가 지정되지 않음

**1825**

파일 풀 정보를 검색하는 중에 오류 발생

**1826**

올바르지 않은 파일 풀 ID

**1827**

올바르지 않은 파일 풀 데이터가 검색됨

**1828**

파일 풀이 정의되지 않음

**1829**

RFS가 파일 풀에 라이브러리를 추가할 수 없음

**1830**

RFS가 사용자 디렉토리를 작성할 수 없음

**1831**

파일 풀의 DDNAME을 지정해야 함

**1832**

올바르지 않은 DDNAME

**1833**

파일 풀 변수가 손상됨

**1834**

풀 ID가 이미 존재함

**1835**

이미 사용된 DDNAME임

**1836**

파일 풀을 형식화할 수 없음

**1837**

파일 풀을 먼저 형식화해야 함

**1838**

파일 풀 ADD 레코드가 가득 참

**1839**

파일 ID를 찾을 수 없음

## 예

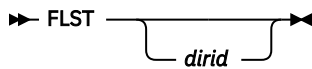
```
'FILEPOOL DEFINE POOL1 REXXDIR1 REXXLIB1 (USER'
```

이 예제는 파일 풀 POOL1을 정의하고 사용할 CICS 파일 정의가 REXXLIB1임을 RFS에 알립니다. 또한 \USERS 디렉토리를 빌드하기 위해 RFS MKDIR을 발행하도록 FILEPOOL FORMAT 명령에 대해 표시합니다.

**참고:** 이는 REXX/CICS 관리자 또는 시스템 프로그래머가 수행하는 권한 부여된 명령입니다..

# FLST

FLST는 파일에 대한 작업을 하기 위해 파일 목록 유틸리티를 호출합니다.



## 피연산자

***dirid***

파일 목록이 표시되는 선택적 전체 또는 부분 디렉토리 ID를 지정합니다. *dirid*를 지정하지 않는 경우 현재 작업 디렉토리로 기본 설정됩니다.

## 리턴 코드

FLST는 RFS에 의해 지정된 리턴 코드를 리턴합니다.

## 예

'FLST'

이 예제는 현재 작업 디렉토리의 멤버에 대해 파일 목록을 표시합니다.

## 참고:

1. FLST가 호출하려고 시도하는 기본 사용자 프로파일 매크로는 EXEC ID FLSTPROF입니다. FLSTPROF 매크로는 사용자(사용자 그룹)가 자신의 고유 기본값을 지정할 수 있도록 하는 기능을 제공합니다.
2. REXX 파일 시스템에 대한 자세한 정보는 [267 페이지의 『제 24 장 REXX/CICS 파일 시스템』](#)의 내용을 참조하십시오.

**FREE**

FREE는 데이터 정의 이름을 할당하지 않습니다.

➤ FREE — DDNAME — *ddname* ➤

## 피연산자

**DDNAME**

ddname으로 사용 가능해졌음을 나타내는 키워드입니다.

***ddname***

데이터 정의 이름을 지정합니다.

## 리턴 코드

### SVC 99에서 지정된 리턴 코드

자세한 정보는 z/OS MVS Programming: Authorized Assembler Services Guide의 내용을 참조하십시오.

## 1702

올바르지 않은 피연산자

## 예

```
'ALLOC USER1A USER1.REXX(TEST) '  
'FREE DDNAME USER1A'
```

이 예는 ddname USER1A로 데이터 세트 USER1.REXX의 멤버 TEST를 할당한 다음 기타 데이터 세트에 사용할 ddname USER1A를 사용 가능하게 합니다.



## GETVERS

GETVERS는 현재 REXX/CICS, 프로그램 이름, 버전과 컴파일 시간 정보를 검색하고, REXX 변수 VERSION에 배치합니다.

▶ GETVERS ◀

리턴된 정보는 *pgmname VxRyMmmmm mm/dd/yyyy hh:mm*의 형식이며 여기서,

***pgmname***

은 REXX/CICS Development System의 경우 CICREXD, REXX/CICS Runtime Facility의 경우 CICREXR를 지정합니다.

***x***

는 REXX/CICS 버전 번호를 지정합니다.

***y***

는 REXX/CICS 릴리스 번호를 지정합니다.

***mm/dd/yyyy***

는 REXX/CICS 기본 프로그램에 대해 컴파일 날짜를 지정합니다.

***hh:mm***

는 REXX/CICS 기본 프로그램에 대해 컴파일 시간을 지정합니다.

**리턴 코드**

**0**

정상 리턴

**1910**

요청 실패

**예**

```
'GETVERS'
```

이 예는 현재 REXX/CICS 버전과 컴파일 시간 정보를 검색하며 이를 REXX 변수 VERSION에 배치합니다. 해당 형식은 CICREXR V1R1M0000 14/12/2018 12:00와 같이 표시될 수 있습니다.

## HELP

HELP는 이 제품 정보 온라인을 찾아보고 검색합니다.

▶ HELP ◀  
└──────────┘  
  *search\_term*

**피연산자**

***search\_term***

찾으려는 문자열을 지정합니다.

**리턴 코드**

***n***

내부 RFS 또는 PANEL 명령에서 전달되는 리턴 코드

**0**

일반 리턴

## IMPORT

IMPORT는 MVS 순차 데이터 세트 또는 파티션된 데이터 세트 멤버를 RFS 파일로 가져옵니다.

## 피연산자

### ***dsn***

완전한(따옴표 없음) MVS 실제 순차 또는 파티션된 조직 데이터 세트 이름을 지정합니다. 데이터 세트가 파티션된 경우(DSORG=PO) 멤버 이름은 소괄호 안에 제공되어야 합니다.

### ***rfs\_fileid***

완전한 REXX 파일 시스템 파일 ID를 지정합니다.

## 리턴 코드

### **0**

일반 리턴

### **1701**

올바르지 않은 명령

### **1702**

올바르지 않은 피연산자

### **1723**

RFS 쓰기 오류

### **1724**

RFS 읽기 오류

### **1725**

동적 할당에 실패함

### **1726**

동적 해제에 실패함

### **1727**

트랜지언트 데이터 큐 열기에 실패함

### **1728**

멤버가 인큐되지 않음

### **1729**

사용 중인 멤버

### **1730**

레코드가 잠림

### **1733**

내보내기를 위한 입력을 찾을 수 없음

### **1735**

트랜지언트 데이터 오류

### **1736**

예상치 못한 CICS 오류.

### **1737**

올바르지 않은 요청

### **1738**

올바르지 않은 데이터 세트 이름

### **1739**

올바르지 않은 배치

### **1741**

지원되지 않는 DSORG

### **1742**

트랜지언트 데이터 풀을 빌드하는 중에 오류 발생

**1743**

REXX 트랜지언트 데이터 큐를 사용할 수 없음/찾을 수 없음

**1744**

사용자가 사인온하지 않음/데이터세트 액세스에 대해 권한을 부여받지 못함

**1745**

비어 있는 데이터 세트

**1746**

REXX 큐를 찾을 수 없음

**1799**

내부 오류

**예**

```
'IMPORT USER1.TEST.DATA POOL1:\USERS\USER1\TEST.DATA'
```

이 예제는 MVS 순차 데이터 세트에서 RFS 파일 POOL1:\USERS\USER1\TEST.DATA에 USER1.TEST.DATA를 복사합니다.

**참고:**

1. 이 명령은 SVC 99를 수행하므로 성능상의 문제로 자주 사용하는 것은 권장되지 않습니다.
2. REXX/CICS 제공 CICSECX1 보안 엑시트가 호출되어 지정된 MVS 데이터 세트에 액세스할 수 있는 권한이 CICS 사인온 사용자 ID에 부여되어 있는지 확인합니다. 기본적으로 데이터 세트의 상위 레벨 접두부는 CICS 사인온 사용자 ID와 일치해야 합니다. CICS 리전에도 지정된 MVS 데이터 세트(예를 들어, 외부 보안이 사용되는 경우 외부 보안 관리자에 의해 지정됨)에 액세스하기 위한 권한이 있어야 합니다.
3. MVS 데이터 세트는 DISP=SHR과 함께 할당됩니다.
4. *rfs\_fileid*에서 지정된 RFS 파일이 이미 존재하는 경우 이 파일은 MVS 데이터 세트의 콘텐츠에 의해 대체됩니다.

**LISTCMD**

LISTCMD는 REXX 명령 정의 정보(이전에 DEFCMD에 의해 지정됨)를 나열합니다.

```
➤ LISTCMD ┌──────────┴──────────┐
           │ envname │ cmdname │
           └──────────┴──────────┘
```

**피연산자****envname**

DEFCMD 또는 DEFSCMD를 사용하여 정의된 명령 환경의 이름을 지정합니다.

**cmdname**

DEFCMD 또는 DEFSCMD에 지정된 명령의 이름을 지정합니다.

**리턴 코드****0**

일반 리턴

**821**

올바르지 않은 환경 이름

**822**

올바르지 않은 명령 이름

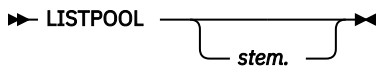
**예**

```
'LISTCMD EDITSVR MYCMD'
```

이 예제는 하위 명령 MYCMD에 대해 DEFCMD에서 정의된 정보를 리턴합니다.

## LISTPOOL

LISTPOOL은 RFS 파일 풀 정보를 터미널에 표시하거나 스템이 지정된 경우에는 지정된 스템 배열에 표시됩니다.



### 피연산자

#### stem.

스템의 이름을 지정합니다. 스템은 마침표로 종료되어야 합니다. [144 페이지의 『스템』](#)의 내용을 참조하십시오. 세 개의 정보 열이 리턴됩니다.

- 풀 ID
- 파일 풀 첫 번째 파일의 ddname
- 사용자 디렉토리를 포함하는지 여부

### 리턴 코드

0

일반 리턴

2225

파일 풀 정보를 검색하는 중에 오류 발생

2226

올바르지 않은 스템 변수 이름

### 예

```
'LISTPOOL LST.'
```

이 예제는 RFS 파일 풀에 대한 정보를 REXX 복합 변수 LST.1 - LST.n에 배치합니다. LST.0는 리턴되는 요소의 수를 포함합니다.

**참고:** 이는 모든 정의된 RFS 파일 풀을 표시하는 일반 사용자 명령입니다.

## LISTTRNID

LISTTRNID는 DEFTRNID 명령을 통해 작성된 현재 트랜잭션 ID 정의를 나열합니다.

이는 권한 부여된 명령입니다.

➡ LISTTRNID ➡

### 리턴 코드

0

일반 리턴

2325

트랜잭션 테이블(trantable) 정보를 검색하는 중에 오류 발생

### 예

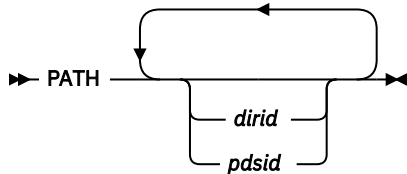
```
'LISTTRNID'
```

ICSTART exec은 기본 트랜잭션과 해당 EXEC 이름을 정의합니다. 결과는 다음과 같이 표시됩니다.

TRNID	EXEC name
REXX	CICRXTRY
EDIT	CICEDIT
FLST	CICFLST
*SIB	CICOVSIB
End of Transaction table list.	

## PATH

PATH는 REXX exec에 대한 검색 경로를 정의합니다.



**참고:** SET 명령(매개변수 없이)은 현재 경로 설정을 검색하는 데 사용됩니다.

### 피연산자

#### *dirid*

실행될 exec을 찾으려고 하면 검색되는 하나 이상의 완전한 REXX 파일 시스템 디렉토리를 지정합니다.

전체 RFS 디렉토리 ID는 풀 ID로 시작되고 양식은 다음과 같습니다. P00L1:\dirid1\...\diridn

둘 이상의 디렉토리 ID를 지정할 때 이를 분리하는 데 공백이 사용됩니다.

#### *pdsid*

하나 이상의 MVS 파티션된 데이터 세트 이름을 지정합니다.

### 리턴 코드

**0**

일반 리턴

**625**

경로 정보를 검색하는 중에 오류 발생

**626**

올바르지 않은 RFS 디렉토리 이름

**627**

올바르지 않은 PDS 이름

**628**

RESULT 값을 설정하는 중에 오류 발생

**629**

올바르지 않은 데이터 세트 이름

**630**

경로 정보를 저장하는 중에 오류 발생

**631**

현재 정의된 경로가 없음

**632**

결과로 생성된 PATH에 RFS 디렉토리 또는 PDS 이름이 없음

### 예제

```
'PATH P00L1:\USERS\USER2 P00L2:\USERS\USER2\PROJECT1'
```

이 예제는 이 목록의 디렉토리가 지정된 순서대로(왼쪽에서 오른쪽) 검색됨을 보여줍니다.

```
'PATH MYUSERID.REXX1.EXECS MYUSERID.REXX2.EXECS'
```

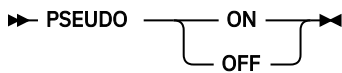
이 예제는 첫 번째 파티션된 데이터 세트에서 검색이 시작됨을 보여줍니다.

#### 참고:

1. *dirid* 및 *pdsid*는 하나의 PATH문에 혼합될 수 있습니다.
2. 문자열을 변수로 연결한 다음 이 변수를 PATH 명령에 지정하여 매우 긴 PATH 디렉토리 목록을 작성할 수 있습니다.
3. PATH 명령 정의는 CICS 다시 시작 시 전달되지 않습니다. PATH 정의를 영구적으로 변경하려면 PATH 명령을 사용자 ID의 기본 디렉토리에 있는 PROFILE exec에 삽입하십시오.
4. PATH 명령은 누적되지 않습니다. 즉, 마지막 PATH 명령이 이전 PATH 정의를 대체합니다.
5. *dirid* 또는 *pdsid*를 지정하지 않으면 현재 작업 경로의 사용자가 검색되고 REXX 특수 변수 RESULT에 배치됩니다.
6. 0이 아닌 리턴 코드를 수신하는 경우 RESULT 특수 변수의 콘텐츠는 예측 불가능합니다.

## PSEUDO

PSEUDO는 의사 대화식 모드를 켜거나 끕니다.



### 피연산자

#### ON

대화식 터미널 읽기가 즉시 발생하는 대신 현재 exec에서 다음 조건 중 하나가 발생하는 경우 터미널 입력이 발생한 다음 터미널 읽기가 발생할 때까지 exec을 일시중단하기 위해 EXEC CICS RETURN TRANSID가 사용되도록 자동 의사 대화식 지원을 사용합니다.

- REXX PULL 명령어
- REXX/CICS WAITREAD 명령
- PANEL RECEIVE 명령
- 화면이 가득 차고 화면의 하단 오른쪽 모서리에 MORE가 표시됩니다(화면이 지워질 때까지 대기하는 내재적 READ).

#### OFF

자동 의사 대화식 지원을 사용 안함으로 설정합니다(끝니다).

### 리턴 코드

#### 0

일반 리턴

#### 2502

올바르지 않은 피연산자

#### 2521

피연산자가 지정되지 않음

### 예

```
'PSEUDO ON'
```

이 예제는 의사 대화식 모드를 켭니다.

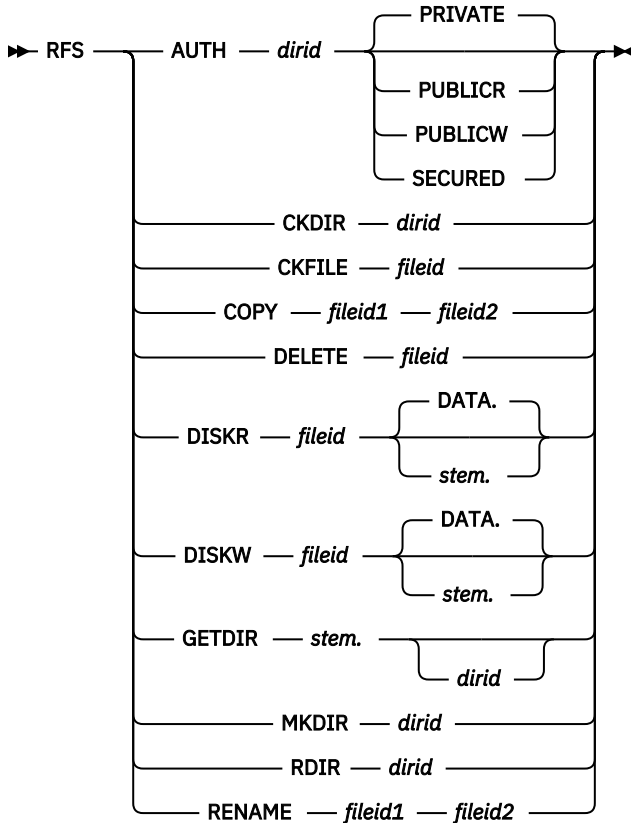
**참고:** PSEUDO ON/OFF 설정은 임시입니다. 의사 설정은 현재 exec이 실행되는 동안 존재합니다. 중첩된 모든 exec은 PSEUDO의 현재 설정을 상속받습니다. 현재 exec이 종료되는 경우 exec에 대한 입력 시 의사의 값이 복원됩니다. SETSYS PSEUDO 명령은 PSEUDO의 시스템 기본 설정을 정의합니다.



**경고:** PSEUDO ON이면 REXX exec은 즉시 의사 대화식이 됩니다. 이 경우 CICS는 다음 의사 대화식 터미널 읽기 시에 모든 파일 변경사항을 커미트하고 비공유 GETMAINed 영역을 해제하게 됩니다.

## RFS

RFS는 REXX 파일 시스템에 파일 입출력(I/O)을 수행합니다.



### 피연산자

#### AUTH

RFS 디렉토리에 대한 액세스에 권한을 부여하는 명령입니다. 지정된 액세스는 이 디렉토리의 모든 파일에 적용됩니다.

#### dirid

REXX 파일 시스템 디렉토리 ID를 지정합니다. 이는 부분 또는 완전한 ID입니다. 자세한 정보는 CD 명령 [333 페이지](#)의 『CD』의 내용을 참조하십시오.

#### PRIVATE

디렉토리의 소유자만이 이 파일에 대한 읽기/쓰기 권한을 가짐을 지정합니다. 이는 기본값입니다.

#### PUBLICR

모든 사용자에게 해당 디렉토리의 파일에 대한 읽기 전용 액세스 권한이 있음을 지정합니다.

#### PUBLICW

모든 사용자에게 해당 디렉토리의 파일에 읽기/쓰기 액세스 권한이 있음을 지정합니다.

#### SECURED

외부 보안 관리자에게 해당 디렉토리의 파일에 대한 액세스 권한이 부여됨을 지정합니다.

#### CKDIR

기존 RFS 디렉토리 레벨을 확인하는 명령입니다.

**CKFILE**

부분적으로 또는 완전한 파일 ID가 존재하는지를 확인하기 위해 검사하는 명령입니다.

**fileid**

파일 ID를 지정합니다.

**COPY**

기존 파일도 대체하면서, 파일을 복사하는 명령입니다.

**fileid1**

소스 파일 ID를 지정합니다. 완전히 또는 부분적으로 검증된 디렉토리와 파일 ID일 수 있습니다.

**fileid2**

대상 파일 ID를 지정합니다. 완전히 또는 부분적으로 검증된 디렉토리와 파일 ID일 수 있습니다.

**DELETE**

RFS 디렉토리 레벨이나 RFS 파일을 삭제하는 명령입니다.

**fileid**

소스 파일 ID를 지정합니다. 완전한 또는 부분적인 ID일 수 있습니다.

**DISKR**

RFS 파일에서 레코드를 읽는 명령입니다.

**stem**

스탬의 이름을 지정합니다. 스탬은 마침표로 종료되어야 합니다. [144 페이지의 『스탬』](#)의 내용을 참조하십시오. 기본 스탬은 DATA.입니다.

**DISKW**

RFS 파일에 레코드를 기록하는 명령입니다.

**GETDIR**

현재 또는 지정된 디렉토리의 콘텐츠 목록을 지정된 REXX 배열로 리턴하는 명령입니다.

**MKDIR**

새 RFS 디렉토리 레벨을 작성하는 명령입니다.

**RDIR**

지정된 RFS 디렉토리를 제거하는 명령입니다.

**RENAME**

RFS 파일의 이름을 새 이름으로 변경하는 명령입니다.

**fileid1**

소스 파일 ID를 지정합니다. 완전히 또는 부분적으로 검증된 디렉토리와 파일 ID일 수 있습니다.

**fileid2**

소스 대상 파일 ID를 지정합니다. 완전히 또는 부분적으로 검증된 디렉토리와 파일 ID일 수 있습니다.

**리턴 코드****0**

정상 리턴

**101**

올바르지 않은 명령

**102**

올바르지 않은 피연산자

**103**

파일이 없음

**104**

권한이 부여되지 않음

**105**

파일이 이미 존재함

**107**

파일 풀에 충분하지 않은 공간

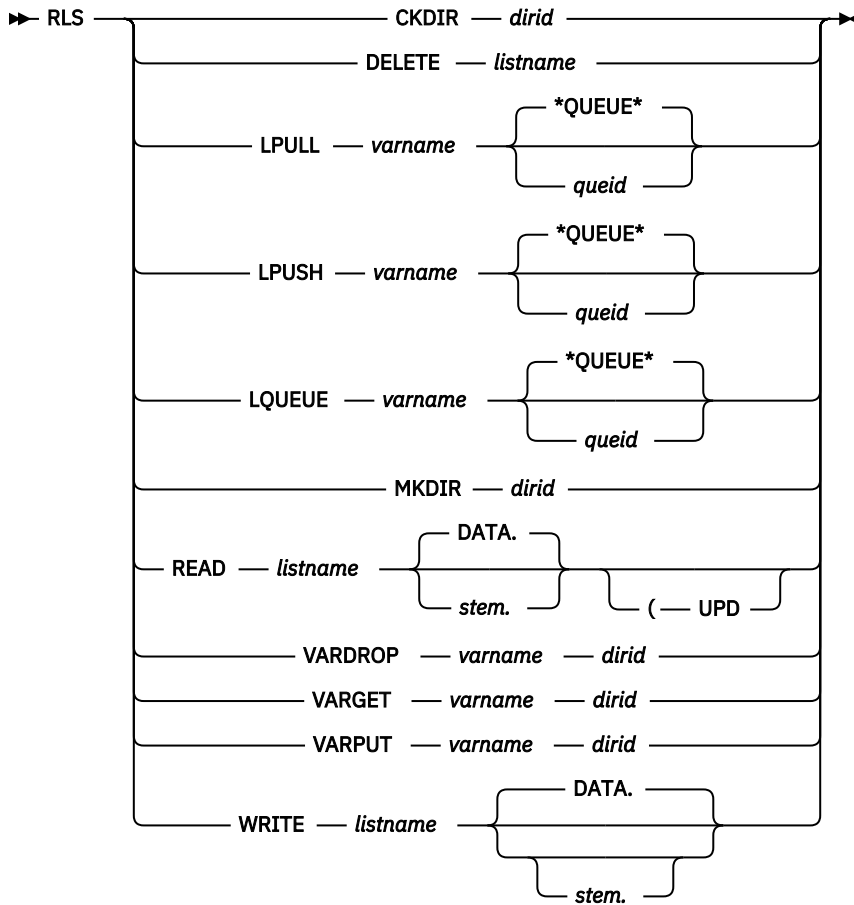


- 110**  
요청 실패
- 111**  
올바르지 않은 파일 ID
- 113**  
디렉토리를 찾을 수 없음
- 115**  
디렉토리가 이미 존재함
- 116**  
디렉토리가 지정되어 있지 않음
- 121**  
파일 손상됨
- 122**  
올바르지 않거나 범위를 벗어난 stem.0
- 126**  
경로 오류
- 127**  
CICS I/O 오류
- 128**  
이 위치로부터 올바르지 않은 명령
- 130**  
디렉토리가 비어 있지 않음
- 131**  
누락된 피연산자
- 132**  
누락된 파일 풀 데이터 레코드. 파일 풀은 형식이 지정되지 않습니다.
- 199**  
내부 오류

**참고:** 파일 액세스 보안 검사는 파일 레벨 대신 디렉토리 레벨에서 수행됩니다. 지정된 파일 ID가 완전한 ID가 아닌 경우, 현재 디렉토리나 PATH 디렉토리는 부분 이름을 완전한 이름으로 해석하기 위한 시도로 사용됩니다. 이 경우에 추가 검사는 필요하지 않습니다. 완전한 파일 ID를 사용하는 경우, 파일이 상주하는 디렉토리는 파일을 열 때 적절한 액세스 권한 부여를 위해 확인됩니다.

## RLS

RLS는 REXX 목록 시스템에 대한 입출력(I/O)을 나열합니다.



## 피연산자

### CKDIR

기존 RLS 디렉토리 레벨을 검사하는 명령입니다.

### dirid

REXX 목록 시스템 디렉토리 레벨 ID를 지정합니다. 부분적이거나 완전합니다. 자세한 정보는 CLD 명령 [349](#) 페이지의 『CLD』의 내용을 참조하십시오.

### DELETE

RLS 디렉토리 레벨 또는 RLS 목록을 삭제하는 명령입니다.

### listname

REXX 목록 시스템 목록 ID를 지정합니다. 부분적이거나 완전합니다.

### LPULL

큐의 맨 위에서 레코드를 가져오는 명령입니다.

### varname

단순 REXX 변수 이름을 지정합니다. 이 변수는 변수 이름을 스템 이름과 구분하는 마침표로 종료되지 않습니다.

### \*QUEUE\*

특수 기본 이름을 지정합니다.

### queid

LPULL, LPUSH 또는 LQUEUE를 통해 액세스하는 특수 RLS 목록 유형의 ID입니다.

### LPUSH

레코드를 큐의 맨 위로 푸시하는 명령입니다(LIFO).

### LQUEUE

레코드를 큐의 끝에 추가하는 명령입니다(FIFO).

**MKDIR**

새 RLS 디렉토리 레벨을 작성하는 명령입니다.

**READ**

RLS 목록에서 스템으로 레코드를 읽는 명령입니다.

**listname**

목록 ID를 지정합니다.

**stem.**

스템의 이름을 지정합니다. 스템은 마침표로 종료되어야 합니다. [144 페이지의 『스템』](#)의 내용을 참조하십시오. 기본 스템은 DATA. 입니다.

**UPD**

업데이트를 위해 목록에서 큐에 넣습니다.

**VARDROP**

RLS 변수가 삭제됨을 나타내는 키워드입니다.

**VARGET**

RLS 변수를 취하여 이를 동일한 이름의 REXX 변수에 복사하는 명령입니다.

**VARPUT**

REXX 변수를 취하여 이를 동일한 이름의 RLS 변수에 복사하는 명령입니다.

**WRITE**

스템에서 RLS 목록에 레코드를 쓰는 명령입니다.

**리턴 코드****0**

일반 리턴

**701**

올바르지 않은 명령

**702**

올바르지 않은 피연산자

**713**

디렉토리를 찾을 수 없음

**715**

디렉토리가 이미 있음

**716**

디렉토리가 지정되지 않음

**723**

목록을 찾을 수 없음

**726**

목록이 지정되지 않음

**728**

목록이 업데이트 모드임

**729**

목록이 업데이트 모드가 아님

**730**

사용자가 사인온하지 않음

**732**

큐가 비어 있음

**733**

이름 지정된 큐를 찾을 수 없음

**736**

스템 또는 변수가 지정되지 않음

**737**

스텝 또는 변수 이름이 너무 김

**738**

스텝 또는 변수 개수가 올바르지 않음

**743**

블록을 찾을 수 없음

**746**

CICGETV 오류

**747**

GETMAIN 오류

**748**

FREEMAIN 오류

**749**

ENQ 오류

**750**

DEQ 오류

**751**

동적 영역 GETMAIN 오류

**752**

저장된 변수 데이터에 오류가 있음

**753**

저장된 변수를 찾을 수 없음

**754**

사용자가 목록의 소유자가 아님

## SCRNINFO

SCRNINFO는 변수 SCRNHT에 두 자리 화면 높이(행 수 단위)를 리턴하고 변수 SCRNWD에 3자리 10진수 화면 너비(열 수 단위)를 리턴합니다.

►► SCRNINFO ◀◀

### 리턴 코드

***n***

오류가 발견되면 CICS에 의해 다시 전달되는 리턴 코드입니다.

**0**

일반 리턴

**-499**

내부 오류

### 예

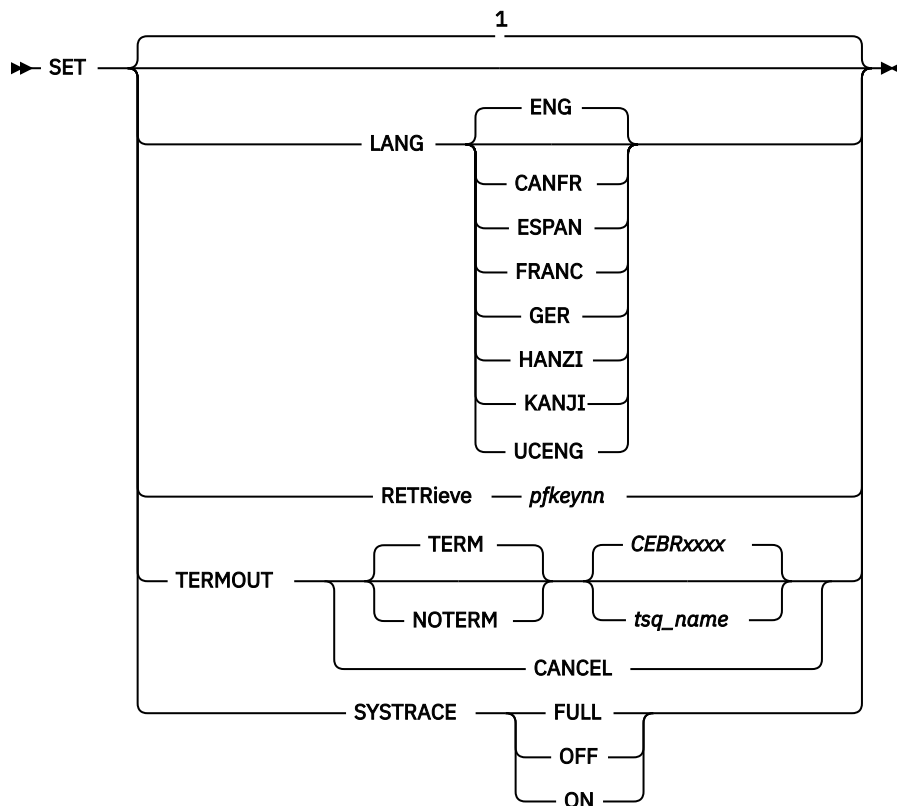
```
'SCRNINFO'
```

### 참고:

1. 리턴되는 화면 정보는 EXEC CICS ASSIGN SCRNHT(*scrnht*) SCRNWD(*scrnwd*)를 수행하여 얻어집니다.
2. 접속된 터미널이 없는 경우 화면 높이 및 화면 너비의 값은 0입니다.

## SET

SET은 현재 사용자에게 대해 REXX/CICS 처리 옵션을 설정합니다. 필요한 SET 명령은 사용자 프로파일 exec에 배치하는 것이 좋습니다.



참고:

<sup>1</sup> SET 명령에 매개변수가 전달되지 않으면 SET은 스템 변수(SET.)를 작성합니다. 이 변수는 SET, SETSYS 또는 PATH 명령을 통해 작성된 사용자에게 대한 모든 처리 옵션을 포함합니다.

## 피연산자

### LANG

REXX 런타임 환경이 메시지와 날짜에 대해 사용하는 언어를 지정합니다.

여러 REXX/CICS 사용자가 동일한 리전에서 동시에 다른 언어를 사용할 수 있습니다.

#### ENG

영어. 이는 기본값입니다.

#### CANFR

캐나다 프랑스어

#### ESPAÑOL

스페인어

#### FRANCO

프랑스어

#### GER

독일어

#### HANZI

대만어

#### KANJI

간지

#### UCENG

대문자 영어

### RETRIEve

입력된 마지막 행을 검색하기 위한 PF 키를 지정합니다.

**pfkeynn**

PF 키 번호를 지정합니다.

**TERMOUT**

터미널에 접속한 상태라도 CICS 임시 스토리지 큐에 터미널 행 모드 출력(예: SAY 및 TRACE 출력)을 전송합니다.

**CANCEL**

행 모드 출력이 CICS 임시 스토리지 큐에 전송되지 않도록 지정합니다.

**NOTERM**

터미널 행 모드 출력이 터미널에 표시되지 않도록 지정합니다.

**TERM**

행 모드 출력이 터미널에 전송되도록 지정합니다.

**tsq\_name**

CICS 임시 스토리지 큐 이름을 지정합니다. 기본 *tsq\_name*은 CEBRxxxx이고 여기서, xxxx는 터미널 ID입니다. (TSQ 이름을 지정하지 않고 CEBR 트랜잭션을 입력하는 경우 이 이름이 사용되는 기본 이름입니다.)

**SYSTRACE**

REXX/CICS 시스템 추적이 켜져 있는지 여부를 지정합니다. 이 옵션은 IBM 서비스의 안내에 따라서만 사용하십시오.

REXX/CICS 시스템 추적은 CICS 사용자 추적도 활성화인 경우에만 생성됩니다.

**FULL**

예외 추적 외에도 기본 REXX/CICS 시스템 추적(입력 및 리턴) 및 보조 추적이 켜지도록 지정합니다.

**OFF**

REXX/CICS 시스템 추적이 꺼져 있도록 지정합니다. 예외 추적만 켜집니다.

**ON**

예외 추적 외에도 기본 REXX/CICS 시스템 추적이 켜지도록 지정합니다.

**리턴 코드****0**

일반 리턴

**421**

올바르지 않은 SET 하위 명령

**422**

변수를 저장하는 중에 오류 발생

**423**

올바르지 않은 언어

**426**

올바르지 않거나 누락된 RETRIEVE PFkey 피연산자

**427**

올바르지 않은 TERMOUT 피연산자

**428**

올바르지 않거나 누락된 SYSTRACE 피연산자

**예**

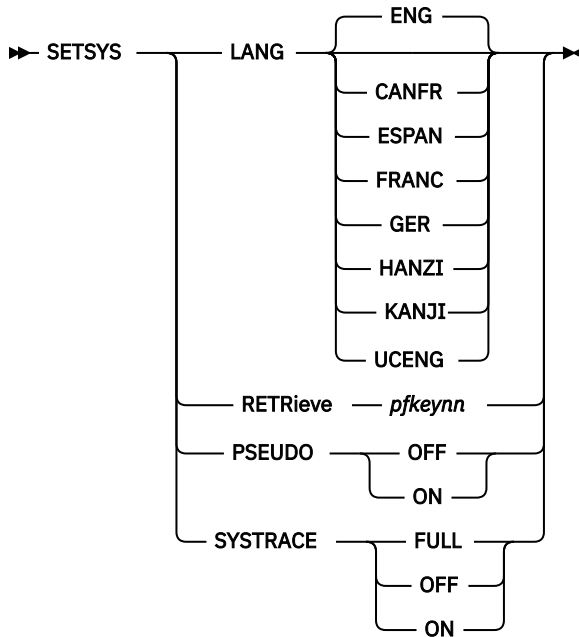
```
'SET TERMOUT TERM TSQ1'
```

이 예제는 터미널 행 모드 출력을 임시 스토리지 큐 TSQ1에 전송하기 위한 처리 옵션을 설정합니다.

## SETSYS

SETSYS는 시스템에 대한 REXX/CICS 처리 옵션을 설정합니다. CICSTART exec에 시스템 전체 SETSYS 명령을 배치하는 것이 좋습니다.

이는 권한 부여된 명령입니다.



### 피연산자

#### LANG

REXX 런타임 환경이 메시지와 날짜에 대해 사용하는 언어를 지정합니다.

#### ENG

영어. 이는 기본값입니다.

#### CANFR

캐나다 프랑스어

#### ESPAN

스페인어

#### FRANC

프랑스어

#### GER

독일어

#### HANZI

대만어

#### KANJI

간지

#### UCENG

대문자 영어

#### RETRieve

입력된 마지막 행을 검색하기 위한 PF 키를 지정합니다.

#### pfkeynn

PF 키 번호를 지정합니다.

## PSEUDO

기본 리전 전체 범위 REXX/CICS 자동 의사 대화식 설정을 지정합니다. [376 페이지의 『PSEUDO』](#)의 내용을 참조하십시오.

이 옵션이 아직 설정되지 않은 경우 자동 의사 대화식 설정이 켜집니다.

### OFF

자동 의사 대화식 설정이 꺼지도록 지정합니다.

### ON

자동 의사 대화식 설정이 켜지도록 지정합니다.

## SYSTRACE

REXX/CICS 시스템 추적이 켜지는지 여부를 지정합니다. 이 옵션은 IBM 서비스의 안내에 따라서만 사용합니다.

REXX/CICS 시스템 추적은 CICS 사용자 추적도 활성화인 경우에만 생성됩니다. 이 옵션이 아직 설정되지 않은 경우 REXX/CICS 시스템 추적은 꺼집니다.

### FULL

예외 추적 외에도 기본 REXX/CICS 시스템 추적(입력 및 리턴)과 보조 추적이 켜지도록 지정합니다.

### OFF

REXX/CICS 시스템 추적이 꺼지도록 지정합니다. 예외 추적만 켜집니다.

### ON

예외 추적 외에도 기본 REXX/CICS 시스템 추적이 켜지도록 지정합니다.

## 리턴 코드

### 0

일반 리턴

### 2721

올바르지 않은 SETSYS 하위 명령

### 2722

변수를 저장하는 중에 오류 발생

### 2723

올바르지 않은 언어

### 2726

올바르지 않거나 누락된 RETRIEVE PFkey 피연산자

### 2728

올바르지 않거나 누락된 SYSTRACE 피연산자

### 2732

올바르지 않거나 누락된 PSEUDO 피연산자

## 예

```
'SETSYS RETRIEVE 12'
```

이 예제는 PF 키 12를 검색 키로 설정합니다.

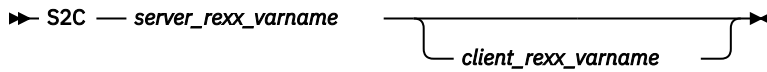
```
SETSYS SYSTRACE ON
```

이 예제는 모든 사용자에게 기본 REXX/CICS 시스템 추적을 설정합니다.

## S2C

S2C는 서버 REXX 변수를 클라이언트 REXX 변수에 복사합니다.





## 피연산자

### **server\_rexx\_varname**

복사 원본인 서버 REXX 변수의 이름입니다.

### **client\_rexx\_varname**

복사 대상인 클라이언트 REXX 변수의 선택적 이름입니다. 지정하지 않으면 *server\_rexx\_varname*과 함께 기본 설정됩니다.

## 리턴 코드

**0**

일반 리턴

**2840**

변수 이름이 지정되지 않음

**2841**

변수를 검색하는 중에 오류 발생

**2842**

변수를 저장하는 중에 오류 발생

**2848**

사용 가능한 클라이언트가 없음

## 예

```
'S2C VARA VARB'
```

이 예제는 클라이언트 REXX 변수 VARB로 복사하는 서버 REXX 변수 VARA의 콘텐츠를 보여줍니다. VARB의 길이는 VARA의 길이와 같습니다.

## 참고:

1. 이 명령의 *varname*에 대해 허용되는 최대 길이는 250자입니다. 더 긴 이름이 지정되면 처음 250자만 사용됩니다.
2. S2C에 의해 복사되는 변수의 최대 길이는 6000바이트입니다.
3. 서버 exec으로부터만의 S2C 명령을 사용할 수 있습니다.

## TERMID

TERMID는 변수 TERMID에 CICS 필드 EIBTRMID로부터 4자로 된 CICS 터미널 ID를 리턴합니다.

➡ TERMID ➡

## 리턴 코드

**0**

일반 리턴

**2921**

터미널 ID를 얻는 중에 오류 발생

**2928**

TERMID 값을 설정하는 중에 오류 발생

예

```
'TERMID'
```

이 예제는 변수 TERMID에 CICS 필드 EIBTRMID로부터의 CICS 터미널 ID를 배치합니다.

## WAITREAD

WAITREAD는 전체 화면 터미널 입력을 수행하고 결과를 복합 변수에 배치합니다.

►► WAITREAD ◀◀

복합 변수의 결과는 다음과 같습니다.

### WAITREAD.0

리턴된 요소의 수를 포함합니다.

### WAITREAD.1

AID 설명을 포함합니다.

### WAITREAD.2

커서 위치를 포함합니다.

### WAITREAD.3 - WAITREAD.n

수정된 나머지 3270개 필드를 포함합니다.

### 리턴 코드

0

일반 리턴

3021

터미널이 접속되지 않음

3099

내부 오류

예

```
'WAITREAD'
```

이 예제는 터미널 화면에서 입력을 읽고 결과를 REXX 복합 변수 WAITREAD.1 - WAITREAD.n에 배치합니다.

**참고:** 수정된 읽기는 터미널에 대해 수행되고 배열은 이 읽기에서 얻은 결과와 함께 리턴됩니다. 이러한 요소의 형식은 다음과 같습니다. *field\_row field\_column data*

## WAITREQ

WAITREQ는 REXX 서버에서만 사용되며 서버가 요청을 대기하도록 합니다. 요청은 수신된 후에 REXX 변수 REQUEST에 배치됩니다.

►► WAITREQ ◀◀

### 리턴 코드

0

일반 리턴

3121

WAITREQ가 사용 가능하지 않음

3122

Exec이 서버가 아님

**3123**

요청 변수를 저장하는 중에 오류 발생

**3199**

내부 오류

클라이언트 프로그램에 반영된 리턴 코드는 WAITREQ 명령에 입력 시 또는 서버 exec의 종료 시 REXX 서버 변수의 값입니다.

**예**

```
'WAITREQ'
```

이 예제의 결과로 서버에 대한 다른 요청이 발생할 때까지 서버 exec은 일시중단됩니다. 요청이 발생하면 서버 exec은 새 요청 값으로 사용하여 이전에 일시 중단되었던 상태로 복원됩니다.



## 제 31 장 오류 번호 및 메시지

오류 코드 및 연관 CICS 메시지가 나열됩니다.

언어 프로세서가 제어를 얻기 전 또는 제어가 언어 프로세서를 남긴 후, 언어 프로세서에 대한 외부 인터페이스가 세 개의 오류 메시지를 생성할 수 있습니다. 그러므로 SIGNAL ON SYNTAX는 다음 오류를 트래핑할 수 없습니다.

- 3 및 5(스토리지에 대한 초기 요구사항을 충족할 수 없는 경우)
- 26(종료 시 리턴된 문자열이 유효한 리턴 코드를 형성하도록 변환될 수 없는 경우)

오류 4는 SIGNAL ON HALT 또는 CALL ON HALT에만 트래핑될 수 있습니다.

레이블 SYNTAX가 오류가 있는 절보다 이전 프로그램에서 표시되지 않으면 언어 프로세서가 감지하는 다음 5개의 오류를 SIGNAL ON SYNTAX에서 트래핑할 수 없습니다.

- 6
- 12
- 13
- 22
- 30

표 5. 오류 코드 및 CICS 메시지

오류 코드	CICS 메시지	오류 코드	CICS 메시지
번호 없음	CICREX255T	오류 26	CICREX466E
오류 3	CICREX451E	오류 27	CICREX467E
오류 4	CICREX452E	오류 28	CICREX486E
오류 5	CICREX450E	오류 29	CICREX487E
오류 6	CICREX453E	오류 30	CICREX468E
오류 7	CICREX454E	오류 31	CICREX469E
오류 8	CICREX455E	오류 32	CICREX492E
오류 9	CICREX456E	오류 33	CICREX488E
오류 10	CICREX457E	오류 34	CICREX470E
오류 11	CICREX458E	오류 35	CICREX471E
오류 12	CICREX459E	오류 36	CICREX472E
오류 13	CICREX460E	오류 37	CICREX473E
오류 14	CICREX461E	오류 38	CICREX489E
오류 15	CICREX462E	오류 39	CICREX474E
오류 16	CICREX463E	오류 40	CICREX475E
오류 17	CICREX465E	오류 41	CICREX476E
오류 18	CICREX491E	오류 42	CICREX477E
오류 19	CICREX482E	오류 43	CICREX478E
오류 20	CICREX483E	오류 44	CICREX479E
오류 21	CICREX464E	오류 45	CICREX480E

표 5. 오류 코드 및 CICS 메시지 (계속)

오류 코드	CICS 메시지	오류 코드	CICS 메시지
오류 22	CICREX449E	오류 46	CICREX218E
오류 23	CICREX1106E	오류 47	CICREX219E
오류 24	CICREX484E	오류 48	CICREX490E
오류 25	CICREX485E	오류 49	CICREX481E

이 메시지에서 용어 언어 프로세서는 REXX/CICS 해석기를 참조합니다.

다음 섹션에서 오류 메시지 외에 언어 프로세서는 다음 터미널 메시지(복구 불가능한)392 페이지의 『CICREX255T』를 실행합니다.

## CICREXnnn 오류 메시지

### CICREX218E 오류 46 올바르지 않은 변수 참조

#### 설명

ARG, DROP, PARSE, PULL 또는 PROCEDURE 명령어 내, 변수 참조의 구문(값이 사용될 변수로, 해당 이름이 소괄호로 둘러싸여 표시되는)이 올바르지 않습니다. 변수 이름 바로 다음에 와야 하는 오른쪽 괄호가 누락될 수 있습니다.

#### 시스템 조치

실행이 중지됩니다.

#### 사용자 응답

필요한 정정을 수행합니다.

### CICREX219E 오류 47 예상치 않은 레이블

#### 설명

INTERPRET 명령어에 대해 평가받는 표현식 또는 대화식 디버그 중 입력된 표현식에서 올바르지 않게 사용되는 레이블이 작성됩니다.

#### 시스템 조치

실행이 중지됩니다.

#### 사용자 응답

이 표현식에 레이블을 사용하지 마십시오.

### CICREX255T Exec 해석기에 충분하지 않은 스토리지

#### 설명

언어 프로세서가 그 자체를 초기화하기에 충분하지 않은 스토리지가 있습니다.

#### 시스템 조치

오류 위치에서 실행이 종료됩니다.

#### 사용자 응답

스토리지를 재정의하고 명령을 다시 실행합니다.

### CICREX449E fn ft, nn행 실행 오류 22: 올바르지 않은 문자열

#### 설명

적용 중인 OPTIONS ETMODE로 스캔된 문자열은 다음 중 하나를 포함합니다.

- 일치하지 않는 SO(shift-out) 및 SI(shift-in) 제어 문자
- SO(shift-out) 및 SI(shift-in) 문자 사이 홀수의 바이트

#### 시스템 조치

실행이 중지됩니다.

#### 사용자 응답

EXEC 파일에서 올바르지 않은 문자열을 수정합니다.

### CICREX450E fn ft, nn행 실행 오류 5: 사용자 스토리지가 고갈되거나 요청이 한계를 초과함

#### 설명

프로그램을 처리하려고 하는 동안, 언어 프로세서는 계속 할 필요가 있는 자원을 얻을 수 없습니다.(예를 들어 작업 영역이나 변수에 필요한 공간을 얻을 수 없습니다.) 언어 프로세서를 호출한 프로그램은 사용 가능한 스토리지의 대부분을 소모하거나 스토리지에 대한 요청이 구현 최대값보다 더 많을 수 있습니다.

#### 시스템 조치

실행이 중지됩니다.

#### 사용자 응답

자체적으로 exec 또는 매크로를 실행하거나 적절하게 종료되지 않은 가능한 루프에 대해 NUCXLOAD를 실행하는

프로그램을 확인하십시오. 추가 스토리지 요구사항은 시스템 관리자를 참조하십시오.

---

**CICREX451E** *fn ft*를 실행하는 오류 3: 프로그램을 읽을 수 없음

---

**설명**

REXX 프로그램을 읽을 수 없습니다. exec 파일의 잘못된 데이터 또는 I/O 오류 때문일 수 있습니다.

**시스템 조치**

실행이 중지됩니다.

**사용자 응답**

exec 파일을 검사하여 수정합니다.

**참고:** 순서 번호는 REXX exec의 73 ~ 80열에는 허용되지 않습니다.

---

**CICREX452E** *fn ft, nn*행 실행 오류 4: 프로그램이 중단됨

---

**설명**

시스템이 사용자 REXX 프로그램의 실행을 중단했습니다. 비참한 오류 조건을 감지하는 경우 특정 유틸리티 모듈은 이 조건을 강제 실행할 수 있습니다.

**시스템 조치**

실행이 중지됩니다.

**사용자 응답**

exec 또는 매크로에서 호출된 유틸리티 모듈의 문제점을 찾으십시오.

---

**CICREX453E** *fn ft, nn*행 실행 오류 6: 일치하지 않는 \*/ 또는 따옴표

---

**설명**

주석 또는 리터럴 문자열이 시작되지만 완료되지 않았습다. 언어 프로세서가 인식하기 때문일 수 있습니다.

- 주석에 대해 닫는 \*/ 또는 리터럴 문자열에 대해 닫는 따옴표를 찾지 않고 파일의 끝(또는 INTERPRET 명령문에서 데이터의 끝)
- 리터럴 문자열을 위한 행의 끝.

**시스템 조치**

실행이 중지됩니다.

**사용자 응답**

exec를 편집하고 닫는 \*/ 또는 따옴표를 추가하십시오. 프로그램의 맨 이에 TRACE SCAN 명령문을 삽입하고 다시

실행할 수도 있습니다. 결과 출력은 오류가 존재하는 위치를 표시해야 합니다.

---

**CICREX454E** *fn ft, nn*행 실행 오류 7: WHEN 또는 OTHERWISE가 예상됨

---

**설명**

언어 프로세서는 SELECT문에서 일련의 WHEN 및 OTHERWISE를 예상합니다. 다른 명령어를 찾거나 모든 WHEN 표현식이 false로 밝혀지고 OTHERWISE가 없는 경우 이 메시지가 실행됩니다. WHEN 다음에 오는 명령어 목록 주위의 DO 및 END 명령어를 잊어 오류가 자주 발생합니다. 예를 들어, 다음과 같습니다.

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

**시스템 조치**

실행이 중지됩니다.

**사용자 응답**

필요한 정정을 수행합니다.

---

**CICREX455E** *fn ft, nn*행 실행 오류 8: 예상치 못한 THEN 또는 ELSE

---

**설명**

언어 프로세서는 해당하는 IF 절과 일치하지 않는 THEN 또는 ELSE를 찾습니다. 이 상황은 자주 복합 IF-THEN-ELSE 구성의 THEN 파트에서 올바르게 않은 DO-END를 사용하여 야기됩니다. 예를 들어, 다음과 같습니다.

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

**시스템 조치**

실행이 중지됩니다.

**사용자 응답**

필요한 정정을 수행합니다.

---

**CICREX456E** *fn ft, nn*행 실행 오류 9: 예상치 못한 WHEN 또는 OTHERWISE

---

## 설명

언어 프로세서는 SELECT 구성 외부에서 WHEN 또는 OTHERWISE 명령어를 찾습니다. END 명령어를 종료하여 DO-END 구성에서 명령어를 실수로 둘러싸거나 SIGNAL 명령문으로 분기하려고 할 수 있습니다(SELECT가 종료되기 때문에 작업할 수 없음).

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

**CICREX457E** *fn ft, nn*행 실행 오류 10: 예상치 못한 또는 일치하지 않는 END

## 설명

언어 프로세서는 DO 또는 SELECT 보다 많은 END를 사용자 프로그램에서 발견했거나 DO 또는 SELECT와 일치하지 않도록 END가 배치되었습니다. 반복적 루프를 닫는 END에 제어 변수의 이름을 넣으면 이 유형의 오류를 찾는 데 도움이 될 수 있습니다.

루프 중앙으로 신호하려고 하는 경우 이 메시지가 야기될 수 있습니다. 이 경우, 이전 DO가 실행되지 않을 것이기 때문에 END는 예상하지 못합니다. SIGNAL이 현재 루프를 종료하므로 루프 내 한 곳에서 다른 곳으로 제어를 전송하는 데 사용될 수 없다는 점을 기억하십시오.

THEN 또는 ELSE 구성 바로 다음에 END를 두거나 DO 다음에 오는 *name*과 일치하지 않는 END 키워드에 *name*을 지정한 경우 이 메시지도 야기될 수 있습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다. TRACE Scan을 사용하여 프로그램의 구조를 표시하는 데 유용하며 쉽게 오류를 찾을 수 있습니다. 반복적 루프를 닫는 END에 제어 변수의 이름을 입력하면 이 종류의 오류를 찾을 수도 있습니다.

**CICREX458E** *fn ft, nn*행 실행 오류 11: 제어 스택이 가득 참

## 설명

제어 구조(DO-END, IF-THEN-ELSE 등)의 중첩 레벨 250 한계를 초과하거나 사용자 스토리지 한계에 도달한 경우 이 메시지가 실행됩니다.

이 메시지는, INTERPRET 명령어 루프로 야기될 수 있습니다.

```
line='INTERPRET line'
INTERPRET line
```

중첩 레벨 한계를 초과했고 이 메시지가 실행될 때까지 이러한 행은 루프됩니다. 마찬가지로, 이 메시지가 표시될 때까지 종료되지 않는 반복적 서브루틴이 올바르게 루프될 수 있습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

**CICREX459E** *fn ft, nn*행 실행 오류 12: 절이 너무 김

## 설명

절에 대한 내부 표현의 길이 한계를 초과했습니다. 실제 한계는 단일 요청에서 얻어질 수 있는 스토리지의 크기입니다.

이 메시지의 원인이 명백하지 않은 경우, 많은 행이 긴 하나의 문자열에 포함되게 하는 누락된 따옴표 때문일 수 있습니다. 이 경우, 절 역 추적에 포함된 데이터 시작에서 오류가 발생할 수 있습니다(콘솔에서 +++로 플래그 지정된).

절의 내부 표시는 문자열 외부에 있는 다중 공백이나 주석을 포함하지 않습니다. 기호(이름)이나 문자열이 내부 표시의 길이에서 두 문자를 얻는다는 것에 또한 주목하십시오.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

**CICREX460E** *fn ft, nn*행 실행 오류 13: 프로그램에서 올바르지 않은 문자

## 설명

언어 프로세서는 리터럴(인용된) 문자열 외부의 올바르지 않은 문자를 찾습니다. 올바른 문자는 다음과 같습니다.

- A-Z a-z 0-9(영숫자 문자)
- @ # \$ % . ? ! \_ (이름 문자)
- & \* ( ) - + = \ ~ ' " ; : < , > / | (특수 문자)

X'0E'(SO)와 X'0F'(SI)로 둘러싸인 경우 및 ETMODE가 사용된 경우, 다음도 올바른 문자입니다.

- X'41' - X'FE'(DBCS 문자)

이 오류의 일부 원인은 다음과 같습니다.

1. 기호에서 악센트 부호가 있는 언어 및 다른 언어 특정 문자 사용.
2. 적용 중인 ETMODE가 없는 DBCS 문자 사용.

**참고:** 다음 문자는 에뮬레이터 구성에 사용된 코드 페이지에 따라 REXX 온라인 도움말에서 다르게 표시될 수 있습니다.



니다. @ # \$ %. CICS 문서에서 사용하는 규칙과 용어도 참조하십시오.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

#### CICREX461E *fn ft, nn*행 실행 오류 14: 불완전 DO/SELECT/IF

##### 설명

언어 프로세서는 파일 끝에 도달했고(또는 INTERPRET 명령어의 데이터 끝) 일치하는 END 없이 DO 또는 SELECT가 있거나 THEN 절 다음에 오지 않는 IF를 찾습니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다. TRACE Scan을 사용하여 프로그램의 구조를 표시하며 누락된 END 또는 THEN이 있어야 하는 위치를 쉽게 찾을 수 있습니다. 반복적 루프를 닫는 END에 제어 변수의 이름을 입력하면 이 종류의 오류를 찾을 수도 있습니다.

#### CICREX462E *fn ft, nn*행 실행 오류 15: 올바르지 않은 16진 또는 2진 문자열

##### 설명

2진 문자열은 REXX에 새롭게 사용되며 언어 프로세서는 사용자의 의도가 없는 경우 이제 사용자 명령문에서 해당 문자열을 2진으로 간주할 수 있습니다.

언어 프로세서의 경우 16진수 문자열은 선행이나 후행 공백이 없을 수 있으며 바이트 경계에서만 공백을 삽입할 수 있습니다. 숫자 0-9 및 문자 a-f 및 A-F만이 허용됩니다. 마찬가지로, 2진 문자열은 4개의 2진 그룹 경계에만 공백이 있을 수 있으며 숫자 0과 1만이 허용됩니다.

다음은 모두 유효한 16진 또는 2진 상수입니다.

'13'x	'0101 1100'b
'A3C2 1c34'x	'001100'B
'1de8'x	"0 11110000"b

0 대신 문자 o 입력과 같이 숫자 중 하나를 잘못 입력했을 수 있습니다. 또는 문자열이 16진이나 2진 스펙으로 의도되지 않은 경우 리터럴 문자열 다음에 1자 기호 X, x, B 또는 b(각각 변수 X 또는 B의 이름)를 둘 수 있습니다. 이 경우, 명시적 연결 연산자(|)를 사용하여 문자열을 기호 값에 연결합니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

#### CICREX463E *fn ft, nn*행 실행 오류 16: 레이블을 찾을 수 없음

##### 설명

언어 프로세서는 SIGNAL 명령어에서 지정된 레이블 또는 해당(트래핑된) 이벤트 발생 시 사용 가능한 조건을 일치하는 레이블을 찾을 수 없습니다. 레이블을 잘못 입력하거나 포함하는 것을 잊거나 또는 대문자여야 하는 경우 대소문자가 혼용된 상태로 입력했을 수 있습니다.

### 시스템 조치

실행이 중지됩니다. 누락된 레이블의 이름은 오류 역 추적에 포함됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

#### CICREX464E *fn ft, nn*행 실행 오류 21: 절 끝에 올바른지 않은 데이터

##### 설명

주석이 아닌 일부 데이터로 SELECT 또는 NOP와 같이 절 다음에 옵니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

#### CICREX465E *fn ft, nn*행 실행 오류 17: 예상치 못한 PROCEDURE

##### 설명

언어 프로세서는 올바르지 않은 위치에서 PROCEDURE 명령어를 만납니다. PROCEDURE 명령어가 이미 내부 루틴 속에 있기 때문이거나 PROCEDURE 명령어가 CALL이나 함수 호출 다음에 실행된 첫 번째 명령어가 아니기 때문에, 내부 루틴도 활성이 아니므로 발생할 수 있었습니다. 이 오류는 CALL이나 함수 호출로 호출하는 대신, 내부 루틴에 "못쓰게 되어" 초래될 수 있습니다.

### 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX466E** *fn ft, nn*행 실행 오류 26: 올바르지 않은 정수

### 설명

언어 프로세서는 NUMERIC 명령어, 구문 분석 위치 패턴 또는 정수로 평가하지 않는 지수(\*\*) 연산자의 오른쪽 조건 또는 이 사용에 대해 999999999의 한계보다 큰 표현식을 찾습니다.

EXIT 또는 RETURN 명령어(REXX 프로그램이 명령으로 호출된 경우)에서 다시 전달된 리턴 코드가 정수이거나 일반 레지스터에 맞지 않는 경우 이 메시지도 실행될 수 있습니다. 이 오류는 기호 이름을 잘못 입력하여 발생할 수 있으므로 이 명령문에서 표현식의 변수 이름이 아닙니다. 예를 들어, "EXIT RC" 대신에 "EXIT CR"을 입력하는 경우 이는 true일 수 있습니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX467E** *fn ft, nn*행 실행 오류 27: 올바르지 않은 DO 구문

### 설명

언어 프로세서는 DO 명령문에서 구문 오류를 발견했습니다. BY, TO, FOR, WHILE, OR UNTIL를 두 번 사용했거나 WHILE 및 UNTIL을 사용했을 수 있습니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX468E** *fn ft, nn*행 실행 오류 30: 이름 또는 문자열 > 250자

### 설명

언어 프로세서는 한계보다 긴 변수나 리터럴(인용된) 문자열을 찾습니다.

대체 후 이름 한계는 250자입니다. 이 오류에 대해 가능한 원인은 이름에서 마침표(.) 사용으로, 예상치 않게 대체를 발생시킵니다.

리터럴 문자열에 대한 한계는 250자입니다. 여러 절이 문자열에 포함될 수 있기 때문에 닫는 따옴표를 종료하여(또는 문자열에 작은따옴표를 두어) 이 오류가 발생할 수 있습니다.

니다. 예를 들어 문자열 'don't'는 'don''t' 또는 "don't"로 작성되어야 합니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX469E** *fn ft, nn*행 실행 오류 31: 이름이 수로 시작하거나 또는

### 설명

언어 프로세서는 이름이 숫자나 마침표(.)로 시작하는 기호를 발견했습니다. 숫자 상수를 재정의한 다음 심각할 수 있기 때문에 REXX 언어 규칙을 사용하면 이름이 숫자나 마침표로 시작하는 기호에 값을 지정할 수 없습니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

올바르게 변수의 이름을 변경하십시오. 영문자로 변수 이름을 시작하는 것이 가장 좋지만 일부 다른 문자가 허용됩니다.

---

**CICREX470E** *fn ft, nn*행 실행 오류 34: 논리 값이 0 또는 1이 아님

### 설명

언어 프로세서는 결과가 0 또는 1이 아닌 IF, WHEN, DO WHILE 또는 DO UNTIL 구문에서 표현식을 찾았습니다. 논리 연산자(¬, \, |, & 또는 &&)로 조작된 값은 결과가 0 또는 1이어야 합니다. 예를 들어, 결과가 0 또는 1이 아닌 값인 경우 구문 "If result then exit rc"는 실패합니다. 그러므로 구문은 If result¬=0 then exit rc로 작성하는 것이 좋습니다.

### 시스템 조치

실행이 중지됩니다.

### 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX471E** *fn ft, nn*행 실행 오류 35: 올바르지 않은 표현식

### 설명

언어 프로세서는 표현식에서 문법적 오류를 발견했습니다. 다음 때문일 수 있습니다.

- 연산자로 표현식을 끝냈습니다.

- 표현식에서 두 사이에 아무 것도 없는 서로 옆 두 연산자를 지정했습니다.
- 표현식이 필요한 경우 지정하지 않았습니다.
- 오른쪽 소괄호가 필요한 경우 지정하지 않았습니다.
- 따옴표로 둘러싸지 않고 의도된 문자 표현식에서 특수 문자(예: 연산자)를 사용했습니다.

마지막 경우의 예는 `LISTFILE ***`가 `LISTFILE '* * *'`(`LISTFILE`이 변수가 아닌 경우) 또는 `'LISTFILE * * *'`로 작성되어야 합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX472E** *fn ft, nn*행 실행 오류 36: 일치하지 않음(표현식에서)

## 설명

언어 프로세서는 표현식에서 일치하지 않는 소괄호를 발견했습니다. 따옴표로 둘러싸지 않고 명령에 단일 소괄호를 포함하는 경우 이 메시지가 발생합니다. 예를 들어, `COPY A B C A B D ' ('` `REP`로서 `COPY A B C A B D (REP`를 씁니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX473E** *fn ft, nn*행 실행 오류 37: 예상치 못함 또는

## 설명

언어 프로세서는 표현식에서 루틴 호출 밖 쉼표(,)를 발견하거나 표현식에서 너무나 많은 오른쪽 소괄호를 발견했습니다. 따옴표에 넣는 것 없이 쉼표를 문자 표현식에 포함시키면 이 메시지가 발생합니다. 예를 들어 다음 명령어:

```
Say Enter A, B, or C
```

should be written as:

```
Say 'Enter A, B, or C'
```

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX474E** *fn ft, nn*행 실행 오류 39: 평가 스택이 오버플로우됨

## 설명

언어 프로세서는 너무 복잡하기 때문에(여러 중첩된 소괄호와 함수) 표현식을 평가할 수 없었습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

하위 표현식을 임시 변수에 지정하여 표현식을 해체하십시오.

---

**CICREX475E** *fn ft, nn*행 실행 오류 40: 루틴에 대한 올바르지 않은 호출

## 설명

언어 프로세서는 루틴에 대해 올바르지 않게 사용된 호출을 발견했습니다. 일부 가능한 원인은 다음과 같습니다.

- 올바르지 않은 데이터(인수)를 기본 제공 또는 외부 루틴에 전달하였습니다(실제 루틴에 따라 다릅니다).
- 또한 많은 인수를 기본 제공, 외부 또는 내부 루틴에 전달하였습니다.
- 호출된 모듈은 언어 프로세서로 호환되지 않았습니다.

루틴을 호출하려고 하지 않은 경우, 공백이나 연산자로 분리됨을 의미할 때 소괄호 (예 인접한 기호나 문자열이 있을 수 있습니다. 이 때문에 함수 호출로 보여집니다. 예를 들어, `TIME(4+5)`는 `TIME*(4+5)`로 작성되어야 합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX476E** *fn ft, nn*행 실행 오류 41: 잘못된 산술 변환

## 설명

언어 프로세서는 올바른 수이거나 -999999999에서 +999999999까지의 허용된 범위 밖에 지수가 있는 산술 연산식에서 조건을 발견했습니다.

변수 이름을 잘못 입력했거나 따옴표에 넣는 것 없이 문자 표현식에 산술 연산자를 포함했을 수 있습니다. 예를 들면 명령 `MSG * Hi!`는 `'MSG * Hi!'`로 작성되어야 하며, 그렇지 않은 경우 언어 프로세서는 `"MSG"`를 `"Hi!"`에 곱하려고 합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX477E** *fn ft, nn*행 실행 오류 42: 산술 오버플로우/언더플로우

## 설명

언어 프로세서는 9자리(999999999 이상 또는 -999999999 미만)의 한계보다 더 큰 지수를 요구한 산술 연산의 결과를 발견했습니다.

표현식의 평가(0으로 수를 나누려는 결과로) 또는 DO 루프 제어 변수의 스텝핑 중 이 오류가 발생할 수 있습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX478E** *fn ft, nn*행 실행 오류 43: 루틴을 찾을 수 없음

## 설명

언어 프로세서는 프로그램에서 호출된 루틴을 찾을 수 없었습니다. 표현식 내 또는 CALL로 호출된 서브루틴에서 함수를 호출했지만 지정된 레이블이 프로그램에 없거나, 기본 제공 함수의 이름이 아니고 REXX/CICS가 외부적으로 찾을 수 없습니다.

이 오류의 가장 간단하면서 가능한 공통된 원인은 잘못된 입력된 이름입니다. 다른 가능성은 표준 함수 패키지 중 하나가 사용할 수 없다는 것일 수 있습니다.

루틴을 호출하려고 하지 않은 경우, 공백이나 연산자로 분리됨을 의미할 때 "("에 인접한 기호나 문자열이 있을 수 있습니다. 언어 프로세서는 함수 호출로서 보일 수 있습니다. 예를 들면 문자열 3(4+5)는 3\*(4+5)로 작성되어야 합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX479E** *fn ft, nn*행 실행 오류 44: 함수는 데이터를 리턴하지 못함

## 설명

언어 프로세서는 표현식 내 외부 루틴을 호출했습니다. 루틴은 오류 없이 종료하는 것 같이 보이지만 표현식에서 사용하도록 데이터를 리턴하지 않았습니다.

REXX 함수로 사용하도록 의도되지 않은 모듈의 이름을 지정하여 야기될 수 있습니다. 명령어나 서브루틴으로 호출되어야 합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX480E** *fn ft, nn*행 실행 오류 45: 함수 RETURN에 지정된 데이터가 없음

## 설명

REXX 프로그램은 함수로 호출되지만 데이터를 다시 전달하지 않고 리턴하려는 시도가 있습니다(RETURN; 명령어로). 마찬가지로, 함수로 호출된 내부 루틴은 표현식을 지정하는 RETURN 명령문으로 끝나야 합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX481E** *fn ft, nn*행 실행 오류 49: 언어 프로세서 실패

## 설명

언어 프로세서는 많은 내부 자체 일관성 검사를 수행합니다. 심각한 오류가 발생하면 이 메시지를 실행합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

IBM 담당자에 이 메시지의 발생을 보고하십시오.

---

**CICREX482E** *fn ft, nn*행을 실행 오류 19: 문자열이나 기호가 예상됨

## 설명

언어 프로세서는 CALL 또는 SIGNAL 명령어 다음에 오는 기호를 예상했지만 아무 것도 발견하지 못했습니다. 문자열이나 기호를 생략했거나 특수 문자(예: 소괄호)를 삽입했을 수 있습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX483E** *fn ft, nn*행 실행 오류 20: 기호 예상됨

### 설명

언어 프로세서는 CALL ON, CALL OFF, END, ITERATE, LEAVE, NUMERIC, PARSE, PROCEDURE, SIGNAL ON 또는 SIGNAL OFF 키워드 다음에 기호를 예상하거나 DROP, UPPER 또는 PROCEDURE(EXPOSE 옵션이 있는) 키워드 다음에 오는 기호나 변수 참조 목록을 예상했습니다. 기호가 필요한 경우 기호가 없거나 일부 다른 문자가 발견되었습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX484E** *fn ft, nn*행 실행 오류 24: 올바르지 않은 TRACE 요청

### 설명

언어 프로세서는 다음의 경우 이 메시지를 실행합니다.

- TRACE 명령어나 TRACE 기본 제공 함수에 대한 인수에서 지정된 조치는 올바른 영문자 옵션 중 하나와 일치하지 않는 문자로 시작합니다. 올바른 옵션은 A, C, E, F, I, L, N, O, R 또는 S입니다.
- 제어 구성 내부 또는 대화식 디버그 중 TRACE Scan을 요청하는 시도가 있음
- 대화식 추적에서, 정수가 아닌 수를 입력합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX485E** *fn ft, nn*행 실행 오류 25: 올바르지 않은 서브 키워드 찾음

### 설명

언어 프로세서는 명령어의 이 위치에서 특정 서브 키워드를 예상했지만 다른 것이 발견되었습니다. 예를 들면, NUMERIC 명령어 다음에 서브 키워드 DIGITS, FUZZ 또는 FORM이 와야 합니다. NUMERIC 다음에 다른 것이 오는 경우 이 메시지가 실행됩니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX486E** *fn ft, nn*행 실행 오류 28: 올바르지 않은 LEAVE 또는 ITERATE

### 설명

언어 프로세서는 올바르지 않은 LEAVE 또는 ITERATE 명령어를 발견했습니다. 다음 중 하나 때문에 명령어가 올바르지 않았습니다.

- 활성인 루프가 없습니다.
- 명령어에 지정된 이름은 활성 루프의 제어 변수와 일치하지 않습니다.

내부 루틴 호출 및 INTERPRET 명령어는 이를 비활성에게 하여 DO 루프를 보호함에 주의하십시오. 그러므로 예를 들어 서브루틴의 LEAVE 명령어는 호출 루틴의 DO 루프에 영향을 줄 수 없습니다.

SIGNAL 명령어를 사용하여 루프 내부 또는 안으로 제어를 전송하는 경우 이 메시지가 실행될 수 있습니다. SIGNAL 명령어는 모든 활성 루프를 종료하며 ITERATE 또는 LEAVE 명령어가 실행된 다음 이 메시지가 실행됩니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX487E** *fn ft, nn*행 실행 오류 29: 환경 이름이 너무 길

### 설명

언어 프로세서는 8자의 한계보다 더 긴 ADDRESS 명령어에 지정된 환경 이름을 발견했습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

올바르게 환경 이름을 지정하십시오.

---

**CICREX488E** *fn ft, nn*행 실행 오류 33: 올바르지 않은 표현식 결과

## 설명

언어 프로세서는 해당 특정 컨텍스트에서 올바르지 않은 표현식 결과를 발견했습니다. 결과는 다음 중 하나에서 올바르지 않을 수 있습니다.

- ADDRESS VALUE 표현식
- NUMERIC DIGITS 표현식
- NUMERIC FORM VALUE 표현식
- NUMERIC FUZZ 표현식
- OPTIONS 표현식
- SIGNAL VALUE 표현식
- TRACE VALUE 표현식.

(FUZZ는 DIGITS보다 작아야 합니다.)

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX489E** *fn ft, nn*행 실행 오류 38: 올바르지 않은 템플릿 또는 패턴

## 설명

언어 프로세서는 구문 분석 템플릿 내 올바르지 않은 특수 문자(예: %)를 발견했거나 변수 트리거의 구문은 올바르지 않습니다(왼쪽 소괄호 다음 발견된 기호가 없음). WITH 서브 키워드가 PARSE VALUE 명령어에서 생략된 경우 이 메시지도 실행됩니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

필요한 정정을 수행합니다.

---

**CICREX490E** *fn ft, nn*행 실행 오류 48: 시스템 서비스에서 실패

## 설명

사용자 입력이나 출력 또는 콘솔 스택의 조작과 같이 일부 시스템 서비스가 올바르게 작동하지 않기 때문에 언어 프로세서는 프로그램의 실행을 정지합니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

입력이 정확하고 프로그램이 올바르게 작동 중인지 확인하십시오. 문제가 지속되면 시스템 지원 담당자에게 통지하십시오.

---

**CICREX491E** *fn ft, nn*행 실행 오류 18: THEN이 예상됨

## 설명

모든 REXX IF 및 WHEN 절 다음에 THEN 절이 와야 합니다. THEN 명령문을 발견하기 전에 다른 절이 발견되었습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

IF 또는 WHEN 절 및 다음 절 사이에 THEN 절을 삽입합니다.

---

**CICREX492E** *fn ft, nn*행 실행 오류 32: 올바르지 않은 스템 사용

## 설명

스템인 기호의 값을 변경하기 위해 시도된 REXX 프로그램입니다. (스템은 첫 번째 마침표까지의 기호 일부입니다. 해당 스템으로 시작하는 모든 변수에 영향을 주려고 하는 경우 스템을 사용합니다.) 이 경우 조치가 알 수 없으므로 오류가 발생하는 UPPER 명령어에 있을 수 있습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

스템의 값을 변경하려고 시도하지 않도록 프로그램을 변경하십시오.

---

**CICREX1106E** *fn ft, nn*행 실행 오류 23: 올바르지 않은 SBCS/DBCS 혼용 문자열

## 설명

일치하지 않는 SO-SI 쌍(즉, SI가 없는 SO)이 있는 문자열 또는 SO-SI 문자 사이의 홀수 바이트는 적용 중인 OPTIONS EXMODE로 처리되었습니다.

## 시스템 조치

실행이 중지됩니다.

## 사용자 응답

올바르지 않은 문자열을 정정하십시오.

## 제 32 장 리턴 코드

REXX/CICS 리턴 코드가 나열됩니다.

명령은 414 페이지의 『특정 명령과 연관되지 않은 리턴 코드』에 나열된 리턴 코드를 리턴할 수도 있습니다.

### 패널 기능 리턴 코드

- 4 경고. 패널 기능이 처리를 계속합니다.
- 8 프로그래머 오류
- 10 상태 정보가 포함된 프로그래머 오류
- 12 CICS 명령 오류
- 14 RFS 오류: 이유 코드에 RFS 리턴 코드가 포함됩니다.
- 16 내부 시스템 오류

추가 코드(예: 상태 및 이유 코드)는 305 페이지의 『제 29 장 REXX/CICS 패널 기능』의 내용을 참조하십시오.

### SQL 리턴 코드

- n* SQL문이 오류 또는 경고가 되는 경우 SQLCODE
- 0 SQL문은 EXECSQL 환경에서 처리되었습니다.
- 30 SQLDSECT 변수를 빌드하기에 메모리가 부족했습니다.
- 31 SQL문 영역을 빌드하기에 메모리가 부족했습니다.
- 32 SQLDA 변수를 빌드하기에 메모리가 부족했습니다.
- 33 SELECT문에 대한 결과 영역을 빌드하기에 메모리가 부족했습니다.

### Db2 리턴 코드

- n* Db2 IFI(Instrumentation Facility Interface)에 대한 호출의 결과를 표시하는 양의 값입니다. Db2 IFI에서 RC가 0(영)이 아닌 경우, REXX 변수 DB2\_RC2는 Db2 IFI 이유 코드를 포함합니다. DB2\_RC2 값은 오류 판별을 위해 z/OS용 Db2 제품 문서의 Db2 코드를 사용하여 RC와 함께 사용됩니다.
- 0 Db2 명령은 Db2 IFI에서 처리되었습니다.
- 50 지정된 Db2 명령은 Db2 IFI에서 처리되도록 허용되기에 너무 짧거나 너무 길입니다. Db2 명령은 6보다 적거나 4092 문자 길이보다 큼니다.
- 51 Db2 IFI에 대해 출력 영역을 빌드하기에 메모리가 부족했습니다.

- 52** Db2 IFI에 대해 통신 영역을 빌드하기에 메모리가 부족했습니다.
- 53** Db2 IFI에 대해 리턴 영역을 빌드하기에 메모리가 부족했습니다.
- 54** DB2\_RC2 REXX 변수는 빌드될 수 없습니다.
- 55** DB2\_BNM REXX 변수는 빌드될 수 없습니다.
- 56** DB2\_OUTPUT.*n* REXX 변수는 빌드될 수 없습니다.
- 57** DB2\_OUTPUT.0 REXX 변수는 빌드될 수 없습니다.

## **RFS 및 FLST**

- 0** 일반 리턴
- 101** 올바르지 않은 명령
- 102** 올바르지 않은 피연산자
- 103** 파일을 찾을 수 없음
- 104** 권한이 부여되지 않음
- 105** 파일이 이미 있음
- 107** 파일 폴에 공간이 충분하지 않음
- 110** 요청이 실패함
- 111** 올바르지 않은 파일 ID
- 113** 디렉토리를 찾을 수 없음
- 115** 디렉토리가 이미 있음
- 116** 디렉토리가 지정되지 않음
- 121** 파일이 손상됨
- 122** 올바르지 않거나 범위를 벗어난 stem.0
- 126** 경로 오류
- 127** CICS I/O 오류
- 128** 이 위치에서 명령이 올바르지 않음



- 130**  
디렉토리가 비어 있지 않음
- 131**  
누락된 피연산자
- 132**  
누락된 파일 폴 데이터 레코드. 파일 폴이 형식화되지 않았을 수 있습니다.
- 199**  
내부 오류

## EDITOR 및 EDIT

- 0**  
일반 리턴
- 201**  
올바르지 않은 명령
- 202**  
올바르지 않은 피연산자
- 203**  
파일을 찾을 수 없음
- 204**  
권한이 부여되지 않음
- 207**  
파일 폴에 공간이 충분하지 않음
- 210**  
요청이 실패함
- 211**  
올바르지 않은 파일 ID
- 223**  
검색 인수를 찾을 수 없음
- 226**  
파일이 현재 편집되고 있음
- 229**  
숫자가 범위 밖에 있음
- 230**  
커서가 파일 영역에 없음
- 231**  
가상 스토리지 부족
- 232**  
접두부 명령 충돌
- 236**  
정의되지 않음
- 299**  
내부 오류
- 1748**  
TIOT(Task Input/Output Table)에 ddname에 대한 항목이 없음
- 1749**  
여러 단위의 데이터세트에 내보낼 수 없음
- 1750**  
ddname에 연결된 데이터 세트가 둘 이상 있음

## DIR

0

일반 리턴

321

현재 RFS 디렉토리 정보에 액세스할 수 없음

322

올바르지 않은 스템 이름

325

RFS 디렉토리 검색 중 오류 발생

## SET

0

일반 리턴

421

올바르지 않은 SET 하위 명령

422

변수를 저장하는 중에 오류 발생

423

올바르지 않은 언어

426

올바르지 않은 RETRIEVE PFkey 피연산자

427

올바르지 않은 TERMOUT 피연산자

## CD

0

일반 리턴

521

파일 폴 정의를 검색하는 중에 오류 발생

522

기본 RFS 디렉토리를 작성하는 중에 오류 발생

523

현재 RFS 디렉토리 정보를 저장하는 중에 오류 발생

524

RFS 디렉토리가 없거나 액세스 권한이 부여되지 않음

525

디렉토리 정보를 검색하는 중에 오류 발생

526

올바르지 않은 파일 폴/디렉토리

527

지난 루트 디렉토리로 돌아갈 수 없음

528

결과 값을 설정하는 중에 오류 발생

## PATH

0

일반 리턴

- 625**  
경로 정보를 검색하는 중에 오류 발생
- 626**  
올바르지 않은 RFS 디렉토리 이름
- 627**  
올바르지 않은 PDS 이름
- 628**  
RESULT 값을 설정하는 중에 오류 발생
- 629**  
올바르지 않은 데이터 세트 이름
- 630**  
경로 정보를 저장하는 중에 오류 발생
- 631**  
현재 정의된 경로가 없음
- 632**  
결과로 생성된 PATH에 RFS 디렉토리 또는 PDS 이름이 없음

## RLS

- 0**  
일반 리턴
- 701**  
올바르지 않은 명령
- 702**  
올바르지 않은 피연산자
- 713**  
디렉토리를 찾을 수 없음
- 715**  
디렉토리가 이미 있음
- 716**  
디렉토리가 지정되지 않음
- 723**  
목록을 찾을 수 없음
- 726**  
목록이 지정되지 않음
- 728**  
목록이 업데이트 모드임
- 729**  
목록이 업데이트 모드가 아님
- 730**  
사용자가 사인온하지 않음
- 732**  
큐가 비어 있음
- 733**  
이름 지정된 큐를 찾을 수 없음
- 736**  
스텝 또는 변수가 지정되지 않음
- 737**  
스텝 또는 변수 이름이 너무 김

- 738**  
스텝 또는 변수 개수가 올바르지 않음
- 743**  
블록을 찾을 수 없음
- 746**  
CICGETV 오류
- 747**  
GETMAIN 오류
- 748**  
FREEMAIN 오류
- 749**  
ENQ 오류
- 750**  
DEQ 오류
- 751**  
동적 영역 GETMAIN 오류
- 752**  
저장된 변수 데이터에 오류가 있음
- 753**  
저장된 변수를 찾을 수 없음
- 754**  
사용자가 목록의 소유자가 아님

## LISTCMD

- 0**  
일반 리턴
- 821**  
올바르지 않은 환경 이름
- 822**  
올바르지 않은 명령 이름

## CLD

- 0**  
일반 리턴
- 923**  
현재 RLS 디렉토리 정보를 저장하는 중에 오류 발생
- 924**  
RLS 디렉토리가 없거나 액세스 권한이 부여되지 않음
- 925**  
디렉토리 정보를 검색하는 중에 오류 발생
- 926**  
올바르지 않은 디렉토리
- 927**  
지난 루트 디렉토리로 돌아갈 수 없음
- 928**  
결과 값을 설정하는 중에 오류 발생

## DEFCMD

- 0  
일반 리턴
- 1001  
올바르지 않은 명령
- 1021  
프로그램을 로드할 수 없음
- 1023  
항목을 찾을 수 없음
- 1048  
사용 가능한 클라이언트가 없음
- 1099  
내부 오류

## DEFSCMD

- 0  
일반 리턴
- 1101  
올바르지 않은 명령
- 1121  
프로그램을 로드할 수 없음
- 1123  
항목을 찾을 수 없음
- 1148  
사용 가능한 클라이언트가 없음
- 1199  
내부 오류

## DEFTRNID

- 0  
일반 리턴
- 1202  
올바르지 않은 피연산자
- 1222  
올바르지 않은 옵션
- 1223  
트랜잭션 테이블(trantable) 정보를 저장하는 중에 오류 발생
- 1225  
트랜잭션 테이블(trantable) 정보를 검색하는 중에 오류 발생
- 1226  
Exec 이름 길이 오류
- 1228  
트랜잭션 테이블(trantable) 값을 설정하는 중에 오류 발생
- 1233  
테이블에서 트랜잭션을 찾을 수 없음

## EXECDROP

0

일반 리턴

1401

올바르지 않은 명령

1402

올바르지 않은 피연산자

1423

EXECLOAD 정보를 저장하는 중에 오류 발생

1425

EXECLOAD 정보를 검색하는 중에 오류 발생

1448

사용 가능한 클라이언트가 없음

## EXECLOAD

0

일반 리턴

1501

올바르지 않은 명령

1502

올바르지 않은 피연산자

1523

EXECLOAD 정보를 저장하는 중에 오류 발생

1525

EXECLOAD 정보를 검색하는 중에 오류 발생

1530

CICPDS 루틴에 링크할 수 없음

1531

CICPDS 루틴에서 오류가 리턴됨

1532

RFS 읽기에서 오류가 리턴됨

1533

PDS 빌드에서 오류가 리턴됨

1547

GETMAIN 오류

1548

사용 가능한 클라이언트가 없음

1599

내부 오류

## EXECMAP

0

일반 리턴

1623

EXECLOAD 디렉토리를 찾을 수 없음

## ALLOC 및 FREE

**SVC 99에서 지정된 리턴 코드**

자세한 정보는 [z/OS MVS Programming: Authorized Assembler Services Guide](#)의 내용을 참조하십시오.

**1702**

올바르지 않은 피연산자

## EXPORT 및 IMPORT

**0**

일반 리턴

**1701**

올바르지 않은 명령

**1702**

올바르지 않은 피연산자

**1723**

RFS 쓰기 오류

**1724**

RFS 읽기 오류

**1725**

동적 할당에 실패함

**1726**

동적 해제에 실패함

**1727**

임시 데이터 큐 열기에 실패함

**1728**

멤버가 인큐되지 않음

**1729**

사용 중인 멤버

**1730**

레코드가 잘림

**1733**

내보내기를 위한 입력을 찾을 수 없음

**1735**

트랜지언트 데이터 오류

**1736**

예상치 못한 CICS 오류

**1737**

올바르지 않은 요청

**1738**

올바르지 않은 데이터 세트 이름

**1739**

올바르지 않은 배치

**1741**

지원되지 않는 DSORG

**1742**

트랜지언트 데이터 풀을 빌드하는 중에 오류 발생

**1743**

REXX 트랜지언트 데이터 큐를 사용할 수 없음/찾을 수 없음

**1744**

사용자가 사인온하지 않음/데이터세트 액세스에 대해 권한을 부여받지 못함

**1745**

비어 있는 데이터 세트

**1746**

REXX 큐를 찾을 수 없음

**1748**

TIOT(Task Input/Output Table)에 ddname에 대한 항목이 없음

**1749**

여러 단위의 데이터세트에 내보낼 수 없음

**1750**

ddname에 연결된 데이터 세트가 둘 이상 있음

**1799**

내부 오류

## FILEPOOL

**0**

일반 리턴

**1802**

올바르지 않은 피연산자

**1821**

올바르지 않은 파일 풀 하위 명령

**1822**

파일 풀 하위 명령이 지정되지 않음

**1823**

파일 풀 정보를 저장하는 중에 오류 발생

**1824**

파일 풀 ID가 지정되지 않음

**1825**

파일 풀 정보를 검색하는 중에 오류 발생

**1826**

올바르지 않은 파일 풀 ID

**1827**

올바르지 않은 파일 풀 데이터가 검색됨

**1828**

파일 풀이 정의되지 않음

**1829**

RFS가 파일 풀에 라이브러리를 추가할 수 없음

**1830**

RFS가 사용자 디렉토리를 작성할 수 없음

**1831**

파일 풀의 DDNAME을 지정해야 함

**1832**

올바르지 않은 DDNAME

**1833**

파일 풀 변수가 손상됨

**1834**

풀 ID가 이미 존재함

**1835**

이미 사용된 DDNAME임



**1836**

파일 풀을 형식화할 수 없음

**1837**

파일 풀을 먼저 형식화해야 함

**1838**

파일 풀 ADD 레코드가 가득 참

**1839**

파일 ID를 찾을 수 없음

## **GETVERS**

**0**

일반 리턴

**1910**

요청이 실패함

## **COPYR2S**

**0**

일반 리턴

**2002**

올바르지 않은 피연산자

**2021**

올바르지 않은 구조 정의

**2022**

올바르지 않은 변수 구조 정의

**2023**

필드 이름을 찾을 수 없음

**2025**

GETVAR 요청 처리 실패

**2026**

올바르지 않은 숫자 입력

**2027**

RFS 읽기 오류

**2028**

올바르지 않은 오프셋

**2029**

올바르지 않은 길이 값

## **COPYS2R**

**0**

일반 리턴

**2102**

올바르지 않은 피연산자

**2121**

올바르지 않은 구조 정의

**2122**

올바르지 않은 변수 구조 정의

**2123**

필드 이름을 찾을 수 없음

**2125**

GETVAR 요청 처리 실패

**2126**

올바르지 않은 숫자 입력

**2127**

RFS 읽기 오류

**2128**

올바르지 않은 오프셋

**2129**

올바르지 않은 길이 값

**LISTPOOL****0**

일반 리턴

**2225**

파일 풀 정보를 검색하는 중에 오류 발생

**2226**

올바르지 않은 스템 변수 이름

**LISTTRNID****0**

일반 리턴

**2325**

트랜잭션 테이블(trantable) 정보를 검색하는 중에 오류 발생

**C2S****0**

일반 리턴

**2440**

변수 이름이 지정되지 않음

**2441**

변수를 검색하는 중에 오류 발생

**2442**

변수를 저장하는 중에 오류 발생

**2448**

사용 가능한 클라이언트가 없음

**PSEUDO****0**

일반 리턴

**2502**

올바르지 않은 피연산자

**2521**

피연산자가 지정되지 않음

**AUTHUSER****0**

일반 리턴

**2602**

올바르지 않은 피연산자 또는 피연산자 누락

**2621**

지정된 사용자 ID의 길이가 올바르지 않음

**2642**

사용자 ID를 저장하는 중에 오류 발생

## **SETSYS**

**0**

일반 리턴

**2721**

올바르지 않은 SETSYS 하위 명령

**2722**

변수를 저장하는 중에 오류 발생

**2723**

올바르지 않은 언어

**2726**

올바르지 않은 RETRIEVE PFkey 피연산자

**2727**

올바르지 않은 TERMOUT 피연산자

**2732**

올바르지 않은 PSEUDO 피연산자

## **S2C**

**0**

일반 리턴

**2840**

변수 이름이 지정되지 않음

**2841**

변수를 검색하는 중에 오류 발생

**2842**

변수를 저장하는 중에 오류 발생

**2848**

사용 가능한 클라이언트가 없음

## **TERMID**

**0**

일반 리턴

**2921**

터미널 ID를 얻는 중에 오류 발생

**2928**

TERMID 값을 설정하는 중에 오류 발생

## **WAITREAD**

**0**

일반 리턴

**3021**

터미널이 접속되지 않음

**3099**

내부 오류

## WAITREQ

**0**

일반 리턴

**3121**

WAITREQ가 사용 가능하지 않음

**3122**

Exec이 서버가 아님

**3123**

요청 변수를 저장하는 중에 오류 발생

**3199**

내부 오류

## 특정 명령과 연관되지 않은 리턴 코드

**-3**

Exec을 찾을 수 없거나 인식되지 않는 명령입니다.

**-4**

사용자가 권한이 부여된 사용자가 아니거나 EXEC이 권한 부여된 명령을 사용하도록 허용되지 않습니다.

**-5**

TWA 크기가 지정되지 않았거나 지정된 TWA 크기가 이 CICS 트랜잭션 정의에 대해 4자 미만 길이입니다.

**-6**

REXX/CICS가 CICS LINK로 시작되었고 통신 영역에 16자 미만이 제공되었습니다. 또는 REXX/CICS가 CICS XCTL로 시작되었고 통신 영역에 MVS SIB 유형 1 제어 블록이 포함되어 있지 않거나 통신 영역 길이가 16자 미만입니다.

**-99**

내부 오류

## EXEC

***n***

호출된 exec의 종료로 설정되는 리턴 코드입니다. [160 페이지의 『EXIT』](#)의 내용을 참조하십시오.

**0**

일반 리턴

**-3**

Exec을 찾을 수 없음

**-10**

Exec 이름이 지정되지 않음

**-11**

올바르지 않은 exec 이름

**-12**

GETMAIN 오류

**-99**

내부 오류

## CEDA 및 CEMT

*n*

오류가 발견되는 경우 CICS에 의해 다시 전달되는 리턴 코드입니다. REXX에서 CEMT 호출 시 DFHEMTA(CEMT 프로그래밍 가능한 인터페이스)를 사용하므로 리턴되는 코드는 CEMT 명령에 대한 코드가 됩니다. CEMT에 대해 가능한 리턴 코드는 다음과 같습니다.

- 1** 찾을 수 없음
- 2** 클래스를 찾을 수 없음
- 3** ERROR
- 4** DFH로 시작됨
- 5** 변경이 올바르지 않음
- 6** 새로 사본을 작성할 수 없음
- 7** 권한이 부여되지 않음
- 8** 옵션 충돌
- 9** 우선순위 > 255
- 10** 콘솔에 적용되지 않음
- 11** 프로그램을 찾을 수 없음
- 12** 올바르지 않은 AUTHID
- 13** 올바르지 않은 ATI / TTI
- 14** INTRA가 아님
- 15** 열기/전환 실패
- 16** SDUMP 사용 중
- 17** 성공하지 못함
- 18** 0>=트리거>32767
- 19** 간접에 적용되지 않음
- 20** 0>최대>999
- 21** 추가가 아님
- 22** 열기/닫기가 실패함

- 23** SDUMP가 억제됨
- 24** C로 시작됨
- 25** 활성이 아님
- 26** 닫히지 않음
- 27** SYS LOG에 적용되지 않음
- 28** 이미 존재함
- 29** 카탈로그 입출력(I/O) 오류
- 30** 1>최대 태스크>2000
- 31** 원격에 적용되지 않음
- 32** NOSTG DSALIMIT
- 33** 0>유효 기간>65535
- 34** AKP가 시스템에 없음
- 35** MAXT. < AMAXT.
- 36** 시스템에 없음
- 37** 올바르지 않은 COMAUTHID
- 38** 50>AKP>65535
- 39** 카탈로그가 가득 참
- 40** 2M>DSALIMIT>16M
- 41** 1>MAXACTIVE>999
- 42** 올바르지 않은 DUMPCODE
- 43** 올바르지 않은 DB2ID
- 44** 500>실행. >2700000
- 45** 100>시간>3600000
- 46** 잘못된 트랜잭션 클래스
- 47** 시간 < 스캔 지연

- 48** 상한치에 도달함
- 49** 1>MROBATCH>255
- 50** 48M>EDSALIM>2047M
- 51** 닫기에 실패함
- 52** BDAM에 적용되지 않음
- 53** 시계가 작동하지 않음
- 54** 올바르지 않은 VTAM
- 55** 경로에 적용되지 않음
- 56** 열린 파일
- 57** 닫는 중
- 58** 즉시 닫는 중(BEING IMMCLOSED)
- 59** 0>스캔 지연>5000
- 60** SNASVCMG에 적용되지 않음
- 61** 이 태스크에 적용되지 않음
- 62** 사용자의 조건에 적용되지 않음
- 63** 올바르지 않은 MSGQUEUE
- 64** 사용자의 행에 적용되지 않음
- 65** 큐가 사용 안함 설정됨
- 66** NOSTG EDSALIMIT
- 67** 부분 덤프
- 68** DDNAME을 찾을 수 없음
- 69** 획득 중
- 70** 강제 닫기 중
- 71** HOLD PROG에 적용되지 않음
- 72** 로드 실패함

- 73** SDUMP가 실패함
- 74** 비어 있거나 닫히지 않음
- 75** 사용 안함으로 설정되지 않음
- 76** 닫기가 요청됨
- 77** 열려 있음
- 78** 사용 불가능함
- 79** 사용 안함으로 설정됨
- 80** 일시 정지 중
- 81** MSG DFHIR3793 참조
- 82** 시작/전환 실패
- 83** 올바르지 않은 계획
- 84** 올바르지 않은 간격
- 85** OUT &-REL이 올바르지 않음
- 86** MSG DFHIR3768 참조
- 87** MSG DFHIR3786 참조
- 88** MAPSET에 적용되지 않음
- 89** MSG DFHIR3771 참조
- 90** 파티션에 적용되지 않음
- 91** MSG DFHIR3773 참조
- 92** 올바르지 않은 DSRTPROGRAM
- 93** MSG DFHIR3775 참조
- 94** MSG DFHIR3776 참조
- 95** MSG DFHIR3777 참조
- 96** MSG DFHIR3778 참조
- 97** MSG DFHIR3779 참조



- 98**  
MSG DFHIR3780 참조
- 99**  
MSG DFHIR3781 참조
- 100**  
MSG DFHIR3791 참조
- 101**  
VTAM에만 적용됨
- 102**  
밖으로 이동
- 103**  
숫자 지정
- 104**  
숫자 오류
- 105**  
NEGPOLL이 올바르지 않음
- 106**  
>20000
- 107**  
올바르지 않은 ENDOFDAY
- 108**  
최대 | 시스템 종료
- 109**  
행 DCB가 열리지 않음
- 110**  
올바르지 않은 PLANEXITNAME
- 111**  
설정에 실패함
- 112**  
제거에 실패함
- 113**  
올바르지 않은 SIGNID
- 114**  
파일 개수 > 0
- 115**  
올바르지 않은 STATSQUEUE
- 116**  
백아웃에 실패함
- 117**  
올바르지 않은 COMTHREADLIM
- 118**  
>최대
- 119**  
올바르지 않은 PURGECYCLE
- 120**  
인다우트임
- 121**  
4>TCBLIMIT>2000
- 122**  
연결 -ACQD

- 123**  
데이터 세트 일시 정지
- 124**  
진행 중
- 125**  
데이터 세트 사용 불가능
- 126**  
데이터 세트가 일시 정지됨
- 127**  
백업 발생
- 128**  
자원이 누락됨
- 129**  
STATS 누락됨
- 130**  
IS SIT 매개변수
- 131**  
PLT를 로드할 수 없음
- 132**  
올바르지 않은 THREADLIMIT
- 133**  
데이터 세트가 없음
- 134**  
복구가 필요함
- 135**  
영구 제거에 실패함
- 136**  
사용 중인 파일임
- 137**  
BDAM에만 적용됨
- 138**  
중지됨
- 139**  
MSG DFHIR3798 참조
- 140**  
프로그램이 URM임
- 141**  
사용 중
- 142**  
ICE가 사용 중임
- 143**  
PCT가 사용 중임
- 144**  
해제되지 않음
- 145**  
삭제에 실패함
- 146**  
올바르지 않은 RECOVSTAT
- 147**  
고장이 아님

- 148**  
올바르지 않은 PROTECTNUM
- 149**  
올바르지 않은 DB2ENTRY
- 150**  
사용 설정된 종료
- 151**  
올바르지 않은 DTRPROGRAM
- 152**  
지원에서 사용 중임
- 153**  
세션에 적용되지 않음
- 154**  
파이프라인에 적용되지 않음
- 155**  
버릴 수 없음
- 156**  
로컬 시스템이 아님
- 157**  
시스템에 적용되지 않음
- 158**  
모델에 적용되지 않음
- 159**  
조치가 없음
- 160**  
XLT를 로드할 수 없음
- 161**  
ISC가 정의되지 않음
- 162**  
이미 활성화됨
- 163**  
올바르지 않은 TRANSID
- 164**  
중복 TRANSID
- 165**  
BWO 지원이 없음
- 166**  
DSNB가 올바르지 않음
- 167**  
DSNB BDAM 또는 경로
- 168**  
업데이트 개수 > 0
- 169**  
DSN - SMS가 관리됨
- 170**  
BWO 오류
- 171**  
DFHTMP 오류
- 172**  
사전 설정된 사인온 오류

- 173**  
CPSVCMG에 적용되지 않음
- 174**  
PRM이 사용 불가능함
- 175**  
XLN이 수행된 경우에는 아님
- 176**  
올바르지 않은 PROGAUTOINST
- 177**  
올바르지 않은 PROGAUTOCTLG
- 178**  
올바르지 않은 PROGAUTOEXIT
- 179**  
해제가 완료되지 않음
- 180**  
해제 중임
- 181**  
재사용이 정의됨
- 182**  
블록 해제가 정의됨
- 183**  
0< 최대값 <999999999
- 184**  
형식이 VARBLE이 아님
- 185**  
LSRPOOL이 없음
- 186**  
연결 중
- 187**  
올바르지 않은 빈도
- 188**  
0>PURGET.>1000000
- 189**  
올바르지 않은 PSDINT
- 190**  
XRF를 사용하지 않음
- 191**  
SETLOGON 실패
- 192**  
이전 레벨 VTAM
- 193**  
ACB가 닫힘
- 194**  
복구 오류
- 195**  
연기됨
- 196**  
로컬 시스템에 적용되지 않음
- 197**  
MSG DFHIR3799 참조

- 198**  
APPC에만 적용됨
- 199**  
ESM이 비활성임
- 200**  
등록 오류
- 201**  
등록 취소 오류
- 202**  
지원이 취소됨
- 203**  
DB2 대기 중
- 204**  
DB2가 비활성임
- 205**  
올바르지 않은 INITPARM
- 206**  
필요하지 않음
- 207**  
여전히 닫는 중
- 208**  
취소된 지원이 없음
- 209**  
DFSMS 카탈로그 오류
- 210**  
DSNB가 제거됨
- 211**  
RLS 지원이 없음
- 212**  
시스템이 잠김
- 213**  
SDTRAN을 찾을 수 없음
- 214**  
SDTRAN이 사용 안함으로 설정됨
- 215**  
SDTRAN SHUT DIS
- 216**  
DSN - DFSMS VSAM
- 217**  
데이터 세트가 마이그레이션됨
- 218**  
올바르지 않은 유틸
- 219**  
LU61 또는 LU62가 아님
- 220**  
NETID 0이 PRFRM 사용
- 221**  
MSG DFHZC0178 참조
- 222**  
GR이 등록되지 않음

- 223**  
NETID 입력
- 224**  
선택도를 찾을 수 없음
- 225**  
세션이 사용 중임
- 226**  
MSG DFHZC0176 참조
- 227**  
인플라이트 삭제
- 228**  
간접에서 사용됨
- 229**  
IRC가 열림
- 230**  
오류 콘솔임
- 231**  
SDTRAN이 원격임
- 232**  
XRF가 활성이 아님
- 233**  
SYSID가 오류 상태임
- 234**  
오류: 처리 지연된 UOW
- 235**  
MSG DFHZC0173 참조
- 236**  
RLS 및 CMT
- 237**  
연결을 끊는 중
- 238**  
키 길이 오류
- 239**  
레코드 크기 오류
- 240**  
누락된 풀 이름
- 241**  
올바르지 않은 이름
- 242**  
풀을 찾을 수 없음
- 243**  
CONTEN 및 RECOV
- 244**  
올바르지 않은 조치
- 245**  
마지막 사용<간격
- 246**  
대기 중
- 247**  
AUDITLOG가 없음

- 248**  
매개변수 불일치
- 249**  
CFDT 서버가 없음
- 250**  
TCPIP가 닫힘
- 251**  
사용 중인 포트임
- 252**  
권한이 부여된 포트가 아님
- 253**  
올바르지 않은 상태
- 254**  
프로파일을 찾을 수 없음
- 255**  
주소를 알 수 없음
- 256**  
올바르지 않은 Q-TYPE
- 257**  
1>MAXOPENTCBS>2000
- 258**  
JVMCLASS 세트가 없음
- 259**  
JVM 프로그램이 아님
- 260**  
올바르지 않은 JVMCLASS
- 261**  
올바르지 않은 DSNAME
- 262**  
1>MAXSOCKETS>65535
- 263**  
엄격한 한계를 초과함
- 264**  
MAXSOCKET에 있음
- 265**  
TCIPSERVICE가 열리지 않음
- 266**  
SESSBEANTIME > 143999
- 267**  
자원이 서비스 상태가 아님
- 268**  
누락된 CORBASERVER 이름
- 269**  
DJAR이 해결을 보류 중임
- 270**  
올바르지 않은 DB2GROUPID
- 271**  
DB2ID & GROUPID가 입력됨
- 272**  
1>MAXJVMTCBS>999

- 273**  
1>MAXXPTCBS>999
- 274**  
IIOPLISTENER가 아님
- 275**  
DSNAPRH를 찾을 수 없음
- 276**  
MAXOPENTCBS < DB2CONN TCBLIMIT
- 277**  
TCBLIMIT > MAXOPENTCBS
- 278**  
DB2 GROUPLID를 찾을 수 없음
- 279**  
DB2 ID를 찾을 수 없음
- 280**  
DJAR 충돌(CORBASERVER 스캔)
- 281**  
JVMPROFILE이 올바르지 않음..프로그램 설정
- 282**  
시작됨
- 283**  
다시 로드되는 중
- 284**  
사용으로 설정되는 중
- 285**  
버려지는 중
- 286**  
시작되지 않음
- 287**  
중지되지 않음
- 288**  
DB2 다시 시작 - 라이트
- 289**  
캐시 크기가 올바르지 않음
- 290**  
TCP/IP가 비활성임
- 291**  
강제 영구 제거 시도
- 292**  
1>MAXSSLTCBS>1024
- 293**  
파이프라인이 사용으로 설정되지 않음
- 294**  
누락된 JVMPROFILE
- 295**  
DFHRPL에 대해 올바르지 않음
- 296**  
DFHRPL에 대해 예약된 랭크 10개
- 297**  
범위를 벗어난 랭크



- 298**  
전송 = 0에 대해 획득하지 않음
- 299**  
MAXJVMTCBS가 초과됨
- 300**  
PSTYPE=NOPS 및 PSDI > 0
- 301**  
올바르지 않은 FILELIMIT
- 302**  
올바르지 않은 PROGRAMLIMIT
- 303**  
올바르지 않은 TSQUEUELIMIT
- 304**  
사용 중인 MQNAME
- 305**  
올바르지 않은 MQNAME
- 306**  
MQNAME을 찾을 수 없음
- 307**  
QMGR 대기 중
- 308**  
1>스레드 한계>256
- 309**  
추가 스레드가 없음
- 310**  
스레드가 제한됨
- 311**  
유출 중
- 312**  
MSG DFHPI2024 참조
- 313**  
과도한 요소 수
- 314**  
1M>TSMMAINLIM>32G
- 315**  
영구 제거 시도 1차
- 316**  
JVMSERVER 사용 중
- 317**  
올바르지 않은 REUSELIMIT
- 318**  
>MEMLIMIT의 25.00%
- 319**  
번들에 의해 정의됨
- 320**  
MSG DFHSO0123 참조
- 321**  
JVM PROG에 적용되지 않음
- 322**  
MGMTPART에 의해 사용됨

**참고:** 모든 리턴 코드가 모든 CICS 릴리스에서 발생 가능한 것은 아닙니다.

**0**

일반 리턴

**-101**

올바르지 않은 명령

## EXECIO

**n**

오류가 발견되면 CICS에 의해 다시 전달되는 리턴 코드입니다.

**0**

일반 리턴

**-202**

올바르지 않은 피연산자

**-221**

너무 많은 피연산자가 지정됨

**-222**

Recno 피연산자가 범위를 벗어남

**-224**

행 피연산자가 올바르지 않음

## CONVTMAP

**n**

MVS 데이터 세트 처리 시도에서 발생하는 리턴 코드입니다.

**0**

일반 리턴

**-302**

올바르지 않은 피연산자

**-321**

올바르지 않은 입력 레코드

**-322**

출력 파일을 쓰는 동안 RFS 오류 발생

## SCRNINFO

**n**

오류가 발견되면 CICS에 의해 다시 전달되는 리턴 코드입니다.

**0**

일반 리턴

**-499**

내부 오류

## CICS

**-521**

명령이 지원되지 않음

**-522**

올바르지 않은 명령 또는 키워드

**-523**

옵션을 지정해야 함

- 524  
지원되지 않는 옵션이 지정됨
- 525  
충돌하는 옵션이 지정됨
- 526  
내재적 옵션이 지정되지 않음
- 527  
옵션에 대한 중복 스펙
- 528  
옵션에 대한 값이 지정되지 않음
- 529  
값이 없어야 하는 옵션에 대해 값을 지정함
- 530  
옵션에 대해 지정된 값이 숫자가 아님
- 531  
올바르지 않은 값
- 532  
지정된 값이 너무 김
- 533  
지정된 값이 너무 짧음
- 534  
값이 지정되지 않음
- 535  
변수 테이블 오버플로우
- 536  
변수의 숫자가 변수 테이블 한계를 초과함
- 537  
인수가 변수여야 함
- 538  
변수가 없음
- 539  
올바르지 않은 변수 이름
- 540  
추적을 위해 마스터 시스템 추적 플래그가 켜져야 함
- 541  
구문 분석 오류
- 542  
일반 이름이 올바르지 않음
- 543  
누락된 오른쪽 소괄호
- 544  
모호한 값/키워드
- 545  
RIDFLD는 전자 변수여야 함
- 546  
RIDFLD는 변수여야 함
- 547  
올바르지 않은 GETVAR 리턴 코드
- 548  
내부 GETVAR 오류

**-549**

잘못된 PUTVAR 리턴 코드

**-550**

PUTVAR이 실패함

**-551**

스토리지를 얻을 수 없음

**-552**

Exec CICS 명령 테이블을 찾을 수 없음

## 제 33 장 2바이트 문자 세트(DBCS) 지원

2바이트 문자 세트는 8비트로 표시할 수 있는 추가 문자가 있는 언어를 지원합니다(한국어 한글과 일본어 간지와 같이). REXX에는 모든 범위의 DBCS 함수 및 처리 기술이 있습니다.

이 DBCS 함수 및 처리 기술은 다음을 포함합니다.

- DBCS 문자가 포함된 기호 및 문자열 처리 기능
- 기호, 주석 및 리터럴 문자열에서 DBCS 문자를 허용하는 옵션.
- 데이터 문자열이 DBCS 문자를 포함할 수 있게 허용하는 옵션.
- 특히 DBCS 문자열의 처리를 지원하는 여러 기능
- 현재 명령과 함수에 대해 정의된 DBCS 개선사항.

**참고:** DBCS 사용은 179 페이지의 『제 19 장 함수』에 설명된 대로 기본 제공 함수의 의미에 영향을 주지 않습니다. 결과에서의 문자를 선택, 연결 및 채우기와 같은 조치로 인수의 특성으로부터 얻는 방법을 설명합니다. 이 정보는 결과 문자가 바이트로 표시되는 방법을 설명합니다. 결과가 인쇄되면 이 내부 표현은 일반적으로 보이지 않습니다. 결과가 특정 터미널에 표시되면 보일지도 모릅니다.

### DBCS: 일반 설명

DBCS가 확장 문자를 나타내기 위해 사용하는 규칙을 정의하는 데 도움이 되는 문자가 나열됩니다.

- 각 DBCS 문자는 2 바이트로 구성됩니다.
- DBCS 제어 문자가 없습니다.
- 코드는 다음 테이블에서 정의된 범위에 있으며, DBCS 공백을 위한 유효한 DBCS 코드를 표시합니다. 간단한 기호, 복합 변수의 스템 또는 레이블에 DBCS 공백이 없을 수 있습니다.

표 6. DBCS 범위	
바이트	EBCDIC
1번째	X'41'에서 X'FE'까지
2번째	X'41'에서 X'FE'까지
DBCS 공백	X'4040'

- DBCS 영숫자 및 특수 기호:

DBCS는 1바이트 문자 세트(SBCS)의 영숫자 및 특수 기호에 해당되는 2진수 표시를 포함합니다. EBCDIC에서, 2바이트 영숫자나 특수 기호의 첫 번째 바이트는 X'42'이며 두 번째는 해당 EBCDIC 코드와 동일한 16진 코드입니다.

다음은 일부 예제입니다.

X'42C1'은 EBCDIC 2바이트 A이며,  
X'4281'은 EBCDIC 2바이트 a이며,  
2X'427D'는 EBCDIC 바이트 인용임

- 대소문자 변환 없음:

일반적으로 DBCS에서 소문자와 대문자의 개념이 없습니다.

- 다음 표기법은 이 정보에서 사용됩니다.

- DBCS 문자: .A .B .C .D
- SBCS 문자: a b c d e
- DBCS 공백: ' . '
- EBCDIC shift-out(X'0E'): <

- EBCDIC shift-in(X'0F'): >

참고: EBCDIC에서, SO(Shift-Out) 및 SI(Shift-In) 문자는 DBCS 문자를 SBCS 문자에서 구별합니다.

## DBCS 데이터 조작 및 기호 사용 설정

OPTIONS 명령어는 REXX가 DBCS 데이터를 인식하는 방법을 제어합니다.

- DBCS 조작을 사용으로 설정하려면 EXMODE 옵션을 사용하십시오.
- DBCS 기호를 사용으로 설정하려면 OPTIONS 명령어에서 ETMODE 옵션을 사용하십시오. 프로그램에서 첫 번째 명령어이어야 합니다. 165 페이지의 『OPTIONS』의 내용을 참조하십시오.

OPTIONS ETMODE이 적용되면, 언어 프로세서는 SO와 SI이 주석에서 패어링시킨다는 것을 확인하기 위해 유효성 검증을 수행합니다. 그렇지 않으면, 주석의 콘텐츠는 확인되지 않습니다. 주석 구분 기호(/\*과 \*/)는 SBCS 문자여야 합니다.

## 기호와 문자열

DBCS에는 DBCS 전용 기호 및 문자열과 혼합 기호 및 문자열이 있습니다.

### DBCS 전용 기호 및 혼합 SBCS/DBCS 기호

DBCS 전용 기호는 다음 테이블에 표시된 대로 공백이 아닌 DBCS 코드로만 구성됩니다.

표 7. DBCS 범위	
바이트	EBCDIC
1번째	X'41'에서 X'FE'까지
2번째	X'41'에서 X'FE'까지
DBCS 공백	X'4040'

혼합 DBCS 기호는 SBCS 기호, DBCS 전용 기호 및 기타 혼합 DBCS 기호의 연결로 구성됩니다. EBCDIC에서는 SO 및 SI가 DBCS 기호를 대괄호로 묶어 SBCS 기호와 구분합니다.

DBCS 기호의 기본값은 SBCS 문자가 대문자 변환된 기호 그 자체입니다.

상수 기호는 SBCS 숫자(0-9) 또는 SBCS 기간으로 시작되어야 합니다. 복합 기호에서 구분 기호(마침표)는 SBCS 문자여야 합니다.

### DBCS 전용 문자열 및 혼합 SBCS/DBCS 문자열

DBCS 전용 문자열은 DBCS 문자로만 구성됩니다.

혼합 SBCS/DBCS 문자열은 SBCS와 DBCS 문자의 조합으로 구성됩니다. EBCDIC에서는 SO와 SI가 DBCS 데이터를 대괄호로 묶어 SBCS 데이터와 구분합니다. SO 및 SI는 혼합 문자열에만 필요하므로 DBCS 전용 문자열과 연관되지 않습니다.

EBCDIC에서:

- DBCS 전용 문자열: .A.B.C
- 혼합 문자열: ab<.A.B>
- 혼합 문자열: <.A.B>
- 혼합 문자열: ab<.C.D>ef

### DBCS 기호 유효성 검증

DBCS 기호를 사용하기 위해 적용되는 규칙과 조건이 나열됩니다.

- 기호의 DBCS 일부는 길에서 짝수인 바이트 수여야 합니다.
- DBCS 영숫자 및 특수 기호는 해당 SBCS 문자와 다르게 간주됩니다. SBCS 문자만이 숫자, 명령어 키워드 또는 연산자에서 REXX로 인식됩니다.

- DBCS 문자는 REXX에서 특수 문자로 사용될 수 없습니다.
- SO와 SI는 연속될 수 없습니다.
- SO 또는 SI의 중첩은 허용되지 않습니다.
- SO와 SI는 패어링시켜야 합니다.
- DBCS 문자로 구성되는 기호의 일부가 DBCS 공백을 포함할 수 없습니다.
- DBCS 문자로 구성되는 기호의 일부 각각은 SO 및 SI로 대괄호 처리되어야 합니다.

이러한 예는 일부 가능한 오용을 표시합니다.

- <.A.BC> Incorrect because of odd byte length.
- <.A.B><.C> Incorrect contiguous SO/SI.
- <> Incorrect contiguous SO/SI (null DBCS symbol).
- <.A<.B>.C> Incorrectly nested SO/SI
- <.A.B.C Incorrect because SO/SI not paired
- <.A. .B> Incorrect because contains blank
- ' . A<.B><.C> Incorrect symbol

### 혼합 문자열 유효성 검증

혼합된 문자열의 유효성 검증은 명령어, 연산자 또는 함수에 따라 다릅니다.

혼합 문자열을 허용하지 않는 명령어, 연산자 또는 함수에 혼합 문자열을 사용하는 경우 구문 오류의 원인이 됩니다.

혼합 문자열 유효성 검증에 다음 규칙이 적용됩니다.

- SO 및 SI가 없는 경우 DBCS 문자열의 길이는 짝수 바이트 수여야 합니다.

EBCDIC에만 해당됨:

- SO 및 SI는 문자열에서 쌍으로 되어 있어야 합니다.
- SO 또는 SI의 중첩은 허용되지 않습니다.

다음 예는 발생할 수 있는 오용을 몇 가지 보여줍니다.

- 'ab<cd' 올바르지 않음 - 쌍을 이루지 않음.
- '<.A<.B>.C>' 올바르지 않음 - 중첩됨.
- '<.A.BC>' 올바르지 않음 - 홀수 바이트 길이.

DBCS 문자 시퀀스에서 주석 구분 기호의 끝을 찾을 수 없습니다. 예를 들어, 프로그램에 /\* < \*/가 포함된 경우 스캔에서는 \*/를 찾는 것이 아니라 <(SO)와 쌍을 이루는 >(SI)를 찾으므로 \*/는 주석의 종료로 인식되지 않습니다.

변수가 REXX 프로그램에서 OPTIONS EXMODE로 작성되거나 수정되거나 참조되는 경우 이 변수는 올바른 혼합 문자열을 포함하는지 여부에 상관없이 유효성 검증됩니다. 참조된 변수가 올바르지 않은 혼합 문자열을 포함하는 경우 이 변수는 구문 오류를 발생시키는데 상관없이 명령어, 함수 또는 연산자에 따라 달라집니다.

ARG, PARSE, PULL, PUSH, QUEUE, SAY, TRACE 및 UPPER 명령어 모두에 OPTIONS EXMODE가 적용된 올바른 혼합 문자열이 필요합니다.

## 명령어 및 DBCS

DBCS와 함께 동작하는 명령어의 설명과 예를 제공합니다.

### PARSE

DBCS를 포함하는 PARSE 명령어의 예.

EBCDIC에서:

```

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>'

```

선행과 후행 SO 및 SI는 단어 구문 분석에 불필요하므로 제거됩니다. 그러나 한 쌍은 리턴될 유효한 혼용 DBCS 문자열을 위해 여전히 필요합니다.

```

PARSE VAR x1 . w2
w2 -> '<. ><.E><.F><>'

```

여기에서 첫 번째 공백은 단어를 구분하며 SO이 DBCS 공백과 유효한 혼용 문자열을 보증하기 위해 문자열에 추가됩니다.

```

PARSE VAR x1 w1 w2
w1 -> '<.A.B>'
w2 -> '<. ><.E><.F><>'

PARSE VAR x1 w1 w2 .
w1 -> '<.A.B>'
w2 -> '<.E><.F>'

```

단어 구분으로 불필요한 SO와 SI가 삭제될 수 있습니다.

```

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 -> ' '

```

마지막 세 개의 예 ", <> 및 <><>는 각각 null 문자열(길이 0의 문자열)임에 주의하십시오. 구문 분석 시, null 문자열이 문자열의 끝과 일치합니다. 이러한 이유로, w1은 전체 문자열의 값으로 지정되고 w2가 null 문자열로 지정됩니다.

## PUSH 및 QUEUE

DBCS를 포함하는 PUSH 및 QUEUE 명령어.

PUSH 및 QUEUE 명령어는 항목을 프로그램 스택에 추가합니다. 스택 항목이 255바이트로 제한되기 때문에, *expression*은 256바이트 미만으로 절단되어야 합니다. 잘라내기가 DBCS 문자열을 분할하면, REXX를 사용하면 SO-SI 쌍의 무결성이 OPTIONS EXMODE의 상태로 유지됩니다.

## SAY 및 TRACE

DBCS를 포함하는 SAY 및 TRACE 명령어.

SAY 및 TRACE 명령어는 출력 스트림에 데이터를 기록합니다. PUSH 및 QUEUE 명령어와 같은 방식으로, SO-SI 쌍이 출력 스트림의 요구사항을 충족하기 위해 분리되는 데이터를 위해 유지됨을 REXX가 보장합니다. SAY 및 TRACE 명령어는 사용자의 터미널에 데이터를 표시합니다. PUSH 및 QUEUE 명령어와 같은 방식으로, SO-SI 쌍이 터미널 행 크기의 요구사항을 충족하기 위해 분리되는 데이터를 위해 유지됨을 REXX가 보장합니다. DIAG-24 값이 더 작은 값을 리턴하면 일반적으로 130 바이트 이하입니다.

데이터가 더 짧은 길이로 분할되면 DBCS 데이터 무결성은 OPTIONS EXMODE의 상태로 듭니다. EBCDIC에서 터미널 행 크기가 4 미만인 경우, 4가 혼용 문자열 데이터에 대해 최소값이기 때문에 문자열은 SBCS 데이터로 처리됩니다.



## UPPER

DBCS를 포함하는 UPPER 명령어.

OPTIONS EXMODE에서, UPPER 명령어는 하나 이상의 변수의 콘텐츠로 SBCS 문자만을 대문자로 변환합니다. DBCS 문자를 변환하지 않습니다. 변수의 콘텐츠가 유효한 혼용 문자열 데이터가 아닌 경우 대문자 변환이 발생하지 않습니다.

## DBCS 함수 처리

일부 기본 제공 함수는 DBCS를 처리할 수 있습니다.

단어 구분 및 길이 구분을 처리하는 함수는 OPTIONS EXMODE 아래 다음 규칙을 준수합니다.

- 문자 계산.

문자열의 길이를 계산할 때 논리 문자 길이가 사용됩니다(즉, 하나의 SBCS 논리 문자의 경우 1바이트, 하나의 DBCS 논리 문자의 경우 2바이트). EBCDIC에서, SO와 SI는 투명한 것으로 간주되며, 모든 문자열 조작의 경우 계산되지는 않습니다.

- 문자열로부터의 문자 추출.

문자는 논리 문자를 기준으로 문자열에서 추출됩니다. EBCDIC에서 선행 SO 및 후행 SI는 하나의 DBCS 문자의 하트로 간주되지 않습니다. 예를 들면, .A 및 .B은 <.A.B>에서 추출되며 마침내 완료된 DBCS 문자로서 유지될 때 SO와 SI가 각 DBCS 문자에 추가됩니다. 다중 문자열이 연속적으로 문자열에서 추출되면 문자 사이에 있는 SO와 SI도 추출됩니다. 예를 들면, .A><.B는 <.A><.B>에서 추출되며, 문자열이 완료된 문자열로 마지막에 사용되면 앞에 SO가 붙으며 뒤에 SI가 붙여 <.A><.B>를 제공합니다.

일부 EBCDIC 예는 다음과 같습니다.

```
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>'
DELSTR(S3,3,1)    -> 'ab<><.A.B>'
DELSTR(S3,4,1)    -> 'abc<><.B>'
DELSTR(S3,3,2)    -> 'ab<.B>'
```

- 문자 연결.

문자열 연결은 유효한 혼용 문자열로만 끝낼 수 있습니다. EBCDIC에서, 결과로 문자열이 연결되는 인접한 SI와 SO(또는 SO 및 SI는) 제거됩니다. DELSTR 함수로서의 내재적 연결 중에도 불필요한 SO 및 SI가 제거됩니다.

- 문자 비교.

문자 기준에 대한 문자열 비교 시 유효한 혼용 문자열이 사용됩니다. 비교되는 경우 DBCS 문자가 항상 SBCS 문자보다 큰 것으로 간주됩니다. 엄격한 비교를 제외하고, SBCS 공백, DBCS 공백 및 선행과 후행 근접 SO 및 SI(또는 SI 및 SO)가 EBCDIC에서 제거됩니다. 길이가 동일하지 않으면 SBCS 공백이 추가될 수 있습니다.

EBCDIC에서, 공백이 없는 문자 사이의 근접한 SO와 SI(또는 SI와 SO)도 비교를 위해 제거됩니다.

**참고:** 유효하지 않는 혼용 문자열을 지정해도 엄격한 비교 연산자 오인한 문자열 오류는 발생하지 않습니다.

EBCDIC에서:

```
'<.A>' = '<.A. >'    -> 1    /* true */
'<><<.A>' = '<.A><><>' -> 1    /* true */
'<> <.A>' = '<.A>'    -> 1    /* true */
```

```
'<.A><><.B>' = '<.A.B>'      -> 1      /* true */
'abc' < 'ab<. >'      -> 0      /* false */
```

#### • 문자열로부터의 단어 추출

단어는 문자열에서 문자가 SBCS 또는 DBCS 공백으로 구분됨을 의미합니다.

EBCDIC에서, 단어가 문자열에서 분리된 경우 선행과 후행 근접한 SO와 SI(또는 SI와 SO)도 비교를 위해 제거되지만, 단어에서 SO와 SI(또는 SI와 SO)가 제거되지 않거나 단어 조작을 위해 분리됩니다. 동시에 추출된 단어 중에 있는 경우 단어의 선행과 후행 근접한 SO 및 SI(또는 SI 및 SO)는 제거되지 않습니다.

EBCDIC에서:

```
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>'
```

## 기본 제공 함수 예

DBCS을 지원하고, 정의된 규칙을 따르는 기본 제공 함수의 예를 제공합니다.

전체 기능 설명 및 구문 다이어그램은 [179 페이지의 『제 19 장 함수』](#)의 내용을 참조하십시오.

### ABBREV

문자열 규칙으로부터 문자 비교와 문자 추출 적용.

EBCDIC에서:

```
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

채우기, 문자열로부터 문자 추출 및 문자 비교 규칙에 대해 문자 연결 적용.

EBCDIC에서:

```
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

문자 연결 규칙 적용.

EBCDIC에서:

```
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 함수의 예.

```
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 */
```

string이 유효한 혼용 문자열이 아니고 C 또는 D가 type으로 지정된 경우, 0이 리턴됩니다.

## FIND

문자열과 문자 비교 규칙으로부터 단어 추출 적용.

```
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 및 LASTPOS

문자열과 문자 비교 규칙으로부터 문자 추출 적용.

```
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 및 OVERLAY

문자열과 문자 비교 규칙으로부터 문자 추출 적용.

EBCDIC에서:

```
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

문자열 규칙으로부터 채우기와 문자 추출에 대해 문자 연결 적용.

```
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. . .B><.C. .D>',11,'<.P>')
-> '<.X.P.P.A.P.P.B><.C.P.P.D>'
```

## LEFT, RIGHT 및 CENTER

문자열 규칙으로부터 채우기와 문자 추출에 대해 문자 연결 적용.

EBCDIC에서:

```
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

개수 문자 규칙 적용.

EBCDIC에서:

```
LENGTH('<.A.B><.C.D><<')              -> 4
```

## REVERSE

문자열과 문자 연결 규칙으로부터 문자 추출 적용.

EBCDIC에서:

```
REVERSE('<.A.B><.C.D><>') -> '<><.D.C><.B.A>'
```

## SPACE

문자열과 문자 연결 규칙으로부터 단어 추출 적용.

EBCDIC에서:

```
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

문자열과 문자 연결 규칙으로부터 문자 추출 적용.

EBCDIC에서:

```
STRIP('<><.A><.B><.A><>',', '<.A>') -> '<.B>'
```

## SUBSTR 및 DELSTR

문자열과 문자 연결 규칙으로부터 문자 추출 적용.

EBCDIC에서:

```
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 및 DELWORD

문자열과 문자 연결 규칙으로부터 단어 추출 적용.

EBCDIC에서:

```
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 함수의 예.

EBCDIC에서:

```
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

문자열, 문자 비교 및 문자 연결 규칙으로부터 문자 추출 적용

EBCDIC에서:

```
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 함수의 예.

EBCDIC에서:

```
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

문자열과 문자 비교 규칙으로부터 문자 추출 적용.

EBCDIC에서:

```
VERIFY('<><><.A.B><><.X>','<.B.A.C.D.E>') -> 3
```

## WORD, WORDINDEX 및 WORDLENGTH

WORDINDEX 및 WORDLENGTH에 대해 문자열로부터 단어 추출 적용, 개수 문자 규칙 적용.

EBCDIC에서:

```
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

문자열 규칙에서 단어 추출 적용.

EBCDIC에서:

```
W = '<><.A. .B><.C. .D>'

WORDS(W)      -> 3
```

## WORDPOS

문자열과 문자 비교 규칙으로부터 단어 추출 적용.

EBCDIC에서:

```
WORDPOS('<.B.C> abc','<.A. .B.C> abc') -> 2
WORDPOS('<.A.B>','<.A.B. .A.B><.B.C. .A.B>',3) -> 4
```

## DBCS 처리 함수

DBCS 혼용 문자열을 지원하는 함수가 설명됩니다. 이러한 함수는 OPTIONS 모드에 상관없이, 혼용 문자열을 처리합니다.

**참고:** DBCS 함수와 함께 사용하는 경우, *length*는 항상 바이트로 측정됩니다(LENGTH(*string*)과 대조적으로, 문자로 측정됨).

## 개수 옵션

EBCDIC에서, 함수에 지정되는 경우 길이를 판별할 때 SO와 SI가 존재한 것으로 간주될지 여부를 제어할 수 있습니다. Y는 개수 SO 및 SI를 혼용 문자열에 지정합니다. N이 기본값으로 SO 및 SI를 계산하지 않도록 지정합니다.

## DBADJUST

EBCDIC에서, DBADJUST는 지정된 *operation*을 기반으로 *string*에서 모든 근접한 SI와 SO(또는 SO와 SI) 문자를 조정합니다.

➤ DBADJUST — ( — *string* — , — *operation* — ) ➤

다음은 유효한 조작입니다. 대문자로 쓰여지고 강조표시된 문자만이 필요합니다. 뒤에 오는 모든 문자가 무시됩니다.

### 공백

근접한 문자를 공백으로 변경합니다(X'4040').

### 제거

근접한 문자를 제거하며 기본값입니다.

## EBCDIC 예

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

EBCDIC에서, DBBRACKET는 DBCS-전용 문자열에 SO와 SI 대괄호를 추가합니다.

➤ DBBRACKET — ( — *string* — ) ➤

*string*이 DBCS-전용 문자열이 아닌 경우, SYNTAX 오류가 발생합니다. 즉, 입력 문자열은 길이에서 짝수인 바이트 수여야 하고 각 바이트가 유효한 DBCS 값이어야 합니다.

## EBCDIC 예제

DBBRACKET(' .A.B')	->	' <.A.B>'
DBBRACKET('abc')	->	SYNTAX error
DBBRACKET(' <.A.B>')	->	SYNTAX error

## DBCENTER

DBCENTER는 *string*을 중앙에 두고 길이를 구성하는 데 필요한 만큼 *pad* 문자를 추가하여 길이의 문자열 *length*를 리턴합니다.

➤ DBCENTER — ( — *string* — , — *length* — , — *pad* — , — *option* — ) ➤

기본 *pad* 문자는 공백입니다. *string*이 *length*보다 길면, 양쪽 끝이 맞도록 잘립니다. 홀수의 문자가 절단되거나 추가되면, 오른쪽 끝은 왼쪽 끝보다 하나가 많은 문자를 잃거나 얻습니다.

*option*은 개수 규칙을 제어합니다. Y는 혼용 문자열 내 SO와 SI를 각각 하나로 계산합니다. N은 SO와 SI를 계산하지 않으며, 이것이 기본값입니다.

## EBCDIC 예

DBCENTER(' <.A.B.C>' , 4)	->	' <.B> '
DBCENTER(' <.A.B.C>' , 3)	->	' <.B> '

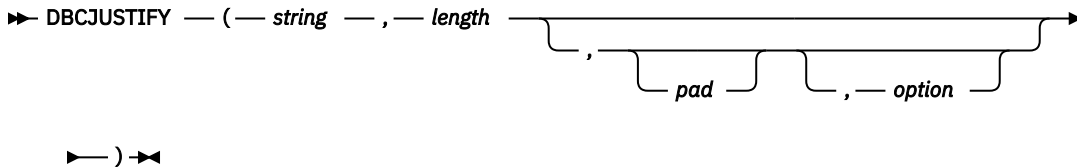
```

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

DBCJUSTIFY는 *pad* 문자를 공백이 없는 문자 사이에 추가하고 양 여백과 바이트 길이 *length* (*length*는 음수가 아니어야 함)를 맞춰 *string*의 형식으로 지정합니다.



*pad* 문자를 공백이 없는 문자 사이에 추가하고 양 여백과 바이트 길이 *length* (*length*는 음수가 아니어야 함)를 맞춰 *string*의 형식으로 지정합니다. 조정 규칙은 JUSTIFY 함수와 동일합니다. 기본 *pad* 문자는 공백입니다.

*option*은 개수 규칙을 제어합니다. Y는 혼용 문자열 내 SO와 SI를 각각 하나로 계산합니다. N은 SO와 SI를 계산하지 않으며, 이것이 기본값입니다.

### 예제

```

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>'

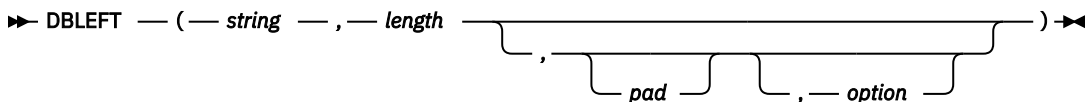
DBCJUSTIFY('<>< AA BB>< CC>',11,'<XX>', 'Y')
-> '<AAXXXBB>'

DBCJUSTIFY('<>< AA BB>< CC>',11,'<XX>', 'N')
-> '<AAXBBXXCC>'

```

## DBLEFT

DBLEFT는 *string*의 가장 왼쪽 *length* 문자를 포함하는 일련의 길이 *length*을 리턴합니다.



리턴된 문자열은 필요에 따라 오른쪽에 *pad* 문자로 채웁니다(또는 자르기). 기본 *pad* 문자는 공백입니다.

*option*은 개수 규칙을 제어합니다. Y는 혼용 문자열 내 SO와 SI를 각각 하나로 계산합니다. N은 SO와 SI를 계산하지 않으며, 이것이 기본값입니다.

### EBCDIC 예

```

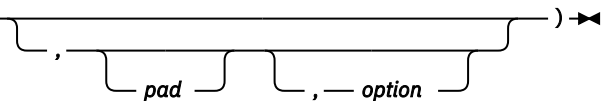
DBLEFT('ab<.A.B>',4)      -> 'ab<.A>'
DBLEFT('ab<.A.B>',3)      -> 'ab'
DBLEFT('ab<.A.B>',4,'x','Y') -> 'abxx'
DBLEFT('ab<.A.B>',3,'x','Y') -> 'abx'
DBLEFT('ab<.A.B>',8,'<.P>') -> 'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>') -> 'ab<.A.B.P>'
DBLEFT('ab<.A.B>',8,'<.P>', 'Y') -> 'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>', 'Y') -> 'ab<.A.B>'

```

## DBRIGHT

DBRIGHT는 *string*의 가장 왼쪽 *length* 문자를 포함하는 일련의 길이 *length*를 리턴합니다.

➡ DBRIGHT — ( — *string* — , — *length* — ) ➡



리턴된 문자열은 필요에 따라 왼쪽에 *pad* 문자로 채웁니다(또는 자르기). 기본 *pad* 문자는 공백입니다.

*option*은 개수 규칙을 제어합니다. Y는 혼용 문자열 내 SO와 SI를 각각 하나로 계산합니다. N은 SO와 SI를 계산하지 않으며, 이것이 기본값입니다.

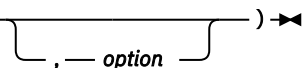
### EBCDIC 예

```
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>'
```

## DBRLEFT

DBRLEFT는 *string*의 DBLEFT 함수에서의 나머지를 리턴합니다. *length*가 *string*의 길이보다 크면, null 문자열을 리턴합니다.

➡ DBRLEFT — ( — *string* — , — *length* — ) ➡



*option*은 개수 규칙을 제어합니다. Y는 혼용 문자열 내 SO와 SI를 각각 하나로 계산합니다. N은 SO와 SI를 계산하지 않으며, 이것이 기본값입니다.

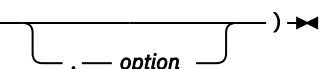
### EBCDIC 예

```
DBRLEFT('ab<.A.B>',4)      -> '<.B>'  
DBRLEFT('ab<.A.B>',3)      -> '<.A.B>'  
DBRLEFT('ab<.A.B>',4,'Y')  -> '<.A.B>'  
DBRLEFT('ab<.A.B>',3,'Y')  -> '<.A.B>'  
DBRLEFT('ab<.A.B>',8)      -> ''  
DBRLEFT('ab<.A.B>',9,'Y')  -> ''
```

## DBRRIGHT

DBRRIGHT는 *string*의 DBRIGHT 함수에서 나머지를 리턴합니다. *length*가 *string*의 길이보다 크면, null 문자열을 리턴합니다.

➡ DBRRIGHT — ( — *string* — , — *length* — ) ➡



*option*은 개수 규칙을 제어합니다. Y는 혼용 문자열 내 SO와 SI를 각각 하나로 계산합니다. N은 SO와 SI를 계산하지 않으며, 이것이 기본값입니다.

### EBCDIC 예

```
DBRRIGHT('ab<.A.B>',4)      -> 'ab'  
DBRRIGHT('ab<.A.B>',3)      -> 'ab<.A>'  
DBRRIGHT('ab<.A.B>',5)      -> 'a'  
DBRRIGHT('ab<.A.B>',4,'Y')  -> 'ab<.A>'  
DBRRIGHT('ab<.A.B>',5,'Y')  -> 'ab<.A>'
```



```
DBRRIGHT('ab<.A.B>',8)    -> ''
DBRRIGHT('ab<.A.B>',8,'Y') -> ''
```

## DBTODBCS

DBTODBCS는 *string* 내 전달된 올바른 모든 SBCS 문자(SBCS 공백 포함)를 해당하는 DBCS 등가로 변환합니다.

➡ DBTODBCS — ( — *string* — ) ➡

기타 단일 바이트 코드 및 모든 DBCS 문자는 변환되지 않습니다. EBCDIC에서 SO 및 SI 대괄호는 적절하게 추가 및 제거됩니다.

### EBCDIC 예제

```
DBTODBCS('Rexx 1988')    -> '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')    -> '<.A. .B>'
```

참고: 이러한 예제에서 .x는 SBCS x에 해당하는 DBCS 문자입니다.

## DBTOSBCS

DBTOSBCS는 *string* 내 전달된 올바른 모든 DBCS 문자(DBCS 공백 포함)를 해당하는 SBCS 등가로 변환합니다.

➡ DBTOSBCS — ( — *string* — ) ➡

기타 DBCS 문자 및 모든 SBCS 문자는 변경되지 않습니다. EBCDIC에서 SO와 SI 대괄호는 적절하게 제거됩니다.

### EBCDIC 예제

```
DBTOSBCS('<.S.d>/<.2.-.1>') -> 'Sd/2-1'
DBTOSBCS('<.X. .Y>')        -> '<.X> <.Y>'
```

참고: 이러한 예제에서 .d는 SBCS d에 해당하는 DBCS 문자입니다. 그러나 .X 및 .Y는 해당하는 SBCS 문자가 없으므로 변환되지 않습니다.

## DBUNBRACKET

EBCDIC에서 DBUNBRACKET은 SO 및 SI 대괄호로 묶인 DBCS 전용 *string*에서 SO 및 SI 대괄호를 제거합니다. *string*이 대괄호로 묶여 있지 않은 경우 SYNTAX 오류가 결과로 발생합니다.

➡ DBUNBRACKET — ( — *string* — ) ➡

### EBCDIC 예제

```
DBUNBRACKET('<.A.B>')    -> '.A.B'
DBUNBRACKET('ab<.A>')    -> SYNTAX error
```

## DBVALIDATE

*string*이 유효한 혼용 문자열 또는 SBCS 문자열이면 DBVALIDATE는 1을 리턴합니다. 그렇지 않으면 0을 리턴합니다.

➡ DBVALIDATE — ( — *string* — ) ➡

혼용 문자열 유효성 검증 규칙은 다음과 같습니다.

- 유효한 DBCS 문자 코드만
- DBCS 문자열은 길이에서 짝수인 바이트 수입니다.

- EBCDIC 전용: 적절한 SO와 SI 쌍.

EBCDIC에서, C가 생략되면, 각 DBCS 문자의 가장 왼쪽 바이트만이 실행 중인 구현에 유효한 범위에 있는지 보기 위해 확인됩니다(즉, EBCDIC에서 가장 왼쪽 바이트 범위는 X'41'에서 X'FE'까지임).

## EBCDIC 예

```
z='abc<de'
DBVALIDATE('ab<.A.B>')      -> 1
DBVALIDATE(z)                -> 0

y='C1C20E111213140F'X
DBVALIDATE(y)                -> 1
DBVALIDATE(y,'C')            -> 0
```

**DBWIDTH**

DBWIDTH는 *string*의 길이를 바이트 단위로 리턴합니다.

➤ DBWIDTH — ( — *string* — ) ➤

option은 계수 규칙을 제어합니다. Y는 혼합 문자열 내에 있는 SO 및 SI를 각각 계수합니다. N은 SO 및 SI를 계수하지 않고 기본값입니다.

## EBCDIC 예제

```
DBWIDTH('ab<.A.B>', 'Y')    ->    8
DBWIDTH('ab<.A.B>', 'N')    ->    6
```

## 제 34 장 예약된 키워드 및 특수 변수

모호성이 없는 다양한 상황에서 키워드를 일반적인 기호로 사용할 수 있습니다.

정확한 규칙이 다음에 제공됩니다.

세 개의 특수 변수인 RC, RESULT 및 SIGL이 있습니다.

### 예약 키워드

REXX의 사용 가능 구문은 일부 기호가 특정 컨텍스트에서 언어 프로세서의 사용에 대해 예약된다는 것을 의미합니다. 특정 명령어에서, 일부 기호는 명령어의 일부를 분리하기 위해 예약될 수 있습니다. 이러한 기호는 키워드로 참조됩니다.

REXX 키워드의 예는 IF 또는 WHEN 절이 다음에 오는 DO 명령어와 THEN(이 경우 절 터미네이터의 역할을 함)에서 WHILE입니다.

이러한 경우를 제외하고, 절에 첫 번째 토큰이고 다음에 = 또는 :이 오지 않는 단순 기호만이 명령어 키워드인지를 확인하기 위해 검사됩니다. 키워드로 사용하지 않고 절의 다른 곳에서 자유롭게 기호를 사용할 수 있습니다.

그러나 사용자가 REXX 키워드(예: QUEUE)와 동일한 이름으로 호스트 명령을 실행하는 것을 권장하지 않습니다. REXX 프로그램이 얼마 동안 제어 밖의 상황에 있을 수 있는 프로그래머 및 프로그램을 절대적으로 *watertight*로 작성하려는 프로그래머에게 문제가 발생할 수 있습니다.

이런 경우 REXX 프로그램은 명령행에서(적어도) 첫 번째 단어가 따옴표로 묶인 채로 작성될 수 있습니다. 예:

```
'SCRNINFO'
```

더 효율적인 장점이 또한 있으며 이 스타일로 SIGNAL ON NOVALUE 조건을 사용하여 exec의 무결성을 검사할 수 있습니다.

대체 전략은 인접한 두 개의 따옴표로 이러한 명령 문자열 앞에 오는 것이며, null 문자열을 앞에 연결합니다. 예:

```
' 'SCRNINFO
```

세 번째 옵션은 전체 표현식(또는 첫 번째 기호)를 괄호로 묶는 것입니다. 예:

```
(SCRNINFO)
```

전략 중에서 선택사항은 프로그래머에 의한 개인 선택사항입니다. REXX 언어는 이를 적용하지 않습니다.

### 특수 변수

RC, RESULT 및 SIGL은 특수 변수입니다.

다음과 같이 언어 프로세서가 자동으로 설정할 수 있는 세 가지 특수 변수가 있습니다.

#### RC

호스트 명령(또는 하위 명령) 실행의 리턴 코드로 설정됩니다. SIGNAL 이벤트인 SYNTAX, ERROR 및 FAILURE에 따라 RC는 이벤트에 적합한 코드 즉, 구문 오류 번호(391 페이지의 『제 31 장 오류 번호 및 메시지』 참조) 또는 명령 리턴 코드로 설정됩니다. RC는 NOVALUE 또는 HALT 이벤트에 따라 변경되지 않습니다.

**참고:** 디버그 모드에서 수동으로 실행되는 호스트 명령은 RC의 값을 변경하지 않습니다.

#### RESULT

RETURN 명령어가 표현식을 지정하는 경우 호출된 서브루틴에서 RETURN 명령어에 의해 설정됩니다. RETURN 명령어에 그에 대한 표현식이 없는 경우 RESULT가 삭제됩니다(초기화되지 않게 됨).

#### SIGL

레이블에 대한 제어의 마지막 전송이 발생했을 당시 실행 중이었던 절의 행 번호를 포함합니다. (SIGNAL, CALL, 내부 함수 호출 또는 트랩된 오류 조건으로 인해 이러한 상황이 발생할 수 있음).

이러한 변수 중 어느 것도 초기값을 가지지 않습니다. 다른 변수에서와 같이 변경할 수 있으며 PROCEDURE 및 DROP 명령어는 일반적인 방식으로 이러한 변수에 영향을 미칩니다.

특정 기타 정보는 항상 REXX 프로그램에 사용 가능합니다. 여기에는 프로그램이 호출된 이름과 프로그램의 소스 (PARSE SOURCE 명령어 사용 시 사용 가능함. 166 페이지의 『PARSE』 참조)가 포함됩니다. PARSE SOURCE 출력은 문자열 CICS와 그 뒤에 오는 호출 유형, 대문자로 된 exec의 이름, 파일의 이름 및 실행 중인 PDS 또는 DDNAME/멤버로 구성됩니다. 이들 뒤에는 프로그램 호출에 사용된 이름 및 초기(기본값) 명령 환경이 옵니다.

또한 PARSE VERSION(166 페이지의 『PARSE』 참조)은 실행 중인 언어 프로세서 코드의 버전과 날짜를 사용 가능하게 합니다. 기본 제공 함수인 TRACE와 ADDRESS는 현재 추적 설정과 환경 이름을 각각 리턴합니다.

DIGITS, FORM 및 FUZZ 기본 제공 함수를 사용하여 NUMERIC 함수의 현재 설정을 얻을 수 있습니다.

## 제 35 장 디버그 지원

이 섹션에서는 문제점의 대화식 디버깅, 실행 인터럽트 및 추적 제어를 설명합니다.

### 프로그램 대화식 디버깅

디버그 기능은 대화식으로 제어된 프로그램 실행을 허용합니다.

TRACE 조치를 접두부 ?가 있는 조치로 변경(예를 들어, TRACE ?A 또는 TRACE 기본 제공 함수)는 대화식 디버그를 켜고 대화식 디버그가 활성화된 사용자에게 표시합니다. 프로그램에서 추가 TRACE 명령어가 무시되며 언어 프로세서는 콘솔에서 추적되는 거의 모든 명령어 다음에 일시정지됩니다(예외는 다음 참조). 언어 프로세서가 일시정지되면, 화면의 낮은 모서리에서 READ로 표시되며 3개의 디버그 조치가 사용 가능합니다.

1. null 행을 입력하면(문자, 심지어 공백이 없는) 디버그 입력을 위한 다음 일시정지까지 언어 프로세서를 계속 실행합니다. 반복적으로 null 행을 입력하므로 일시정지 지점에서 일시정지 지점으로 단계적으로 이동합니다. TRACE ?A의 경우, 예를 들면, 이는 프로그램을 통한 단일 스텝과 동일합니다.
2. 공백 없이 등호(=)를 입력하면 언어 프로세서가 마지막으로 추적된 절을 재실행하게 합니다. 예를 들면 IF 절이 잘못된 분기를 잡으려는 경우 의존하는 변수의 값을 변경한 다음 재실행할 수 있습니다.

일단 절이 재실행되면, 언어 프로세서는 다시 일시정지됩니다.

3. 입력된 다른 항목은 하나 이상의 절 행으로 처리되며 즉시 처리됩니다(즉 DO; line; END;가 프로그램에 삽입된 것처럼). 동일한 규칙이 INTERPRET 명령어로 적용됩니다(예를 들어 DO-END 구성이 완전해야 함). 명령어에 구문 오류가 있는 경우 표준 메시지가 표시되며 다시 입력하도록 프롬프트가 표시됩니다. 마찬가지로 문자열이 의도적이지 않은 제어 전송을 방지하기 위해 처리되는 동안 다른 모든 SIGNAL 조건이 사용 불가능합니다.

문자열의 실행 동안 호스트 명령으로부터 0이 아닌 리턴 코드가 표시된다는 점을 제외하고 추적이 발생하지 않습니다. 호스트 명령이 항상 실행되지만(즉, 접두부 !가 TRACE 명령어에서 영향을 받지 않음), 변수 RC는 설정되지 않습니다.

일단 문자열이 처리되면, TRACE 명령어가 입력되지 않은 경우 언어 프로세서는 추가 디버그 입력을 위해 다시 일시정지합니다. 이 후자의 경우, 언어 프로세서는 바로 추적 조치를 변경한 다음(필요한 경우) 다음 일시정지 지점까지(있는 경우) 계속 실행합니다. 추적 조치를 변경한 다음(예를 들어 모두에서 결과까지) 명령어를 재실행하려면, 기본 제공 함수 TRACE를 사용해야 합니다(203 페이지의 『TRACE』 참조). 예를 들면, CALL TRACE I는 해당 조치를 I로 변경하고 일시정지된 후 명령문의 재실행을 허용합니다. TRACE 명령어가 접두부를 사용하는 경우 적용되면, 또는 TRACE 0이나 옵션이 없는 TRACE가 입력되면 항상 대화식 디버그가 꺼집니다.

TRACE 명령어의 숫자 양식을 사용하여 프로그램의 섹션이 디버그 입력에 대한 일시정지 없이 실행할 수 있습니다. TRACE n(즉, 정상적인 결과)을 사용하면 계속 실행할 수 있으며 다음 n 일시정지를 건너뛸 수 있습니다(대화식 디버그가 활성화되거나 활성화가 되면). TRACE -n(즉, 비정상적인 결과)을 사용하면 그렇지 않으면 추적될 수 있는 n 절에 대해 금지된 추적을 포함하거나 포함하지 않고 계속 실행할 수 있습니다.

TRACE 명령어에서 선택된 추적 조치가 저장되고 서브루틴 호출 전체에 걸쳐 복원됩니다. 프로그램을 단계별로 실행한 다음(TRACE ?R을 사용하여 결과 추적 후 말함) 관련이 없는 서브루틴을 입력하는 경우, TRACE 0을 입력하여 추적을 끝낼 수 있음을 의미합니다. 서브루틴에서 추가 명령어가 추적되지 않지만 호출자에 대한 리턴 시 추적이 복원됩니다.

마찬가지로, 서브루틴에만 관심이 있는 경우 시작에 TRACE ?R 명령어를 입력할 수 있습니다. 루틴을 추적하는 추적의 원래 상태가 복원되며(서브루틴에 대한 입력 시 추적을 끈 경우) 서브루틴에 대한 다음 입력 때까지 추적(및 대화식 디버그)가 꺼집니다.

명령어가 대화식 디버그에서 실행될 수 있기 때문에 실행 시 상당한 제어가 있습니다.

**참고:** 대화식 디버그 중, 대화식 디버그 때문뿐 아니라 PULL 명령문 때문에 일시정지될 수 있습니다. PULL 명령문을 포함하는 프로그램의 경우, 각 일시정지를 위한 이유를 아는 것이 중요합니다. 프로그램에서, PULL 명령문은 자주 SAY 명령문과 쌍을 이룹니다. 사용자는 PULL을 위한 추적 행 다음 일시정지 시 PULL을 위한 데이터를 입

력해야 합니다(특히 PULL을 위한 데이터 입력을 위해 일시 정지). 사용자는 해당 SAY 명령문(이는 대화식 디버그 일시정지임) 다음 일시 정지 시 데이터를 입력하지 않아야 합니다.

**참고:** 일부 절이 안전하게 재실행될 수 있으므로 언어 프로세서는 추적된 경우에도 일부 절 다음 일시정지되지 않습니다. 다음과 같습니다.

- 루프 주위의 두 번째 또는 추후 시간 시 반복적 DO 절.
- 모든 END 절(모든 경우에 일시정지하기에 유용한 위치는 아님).
- All THEN, ELSE, OTHERWISE 또는 null 절.
- All RETURN 및 EXIT 절.
- All SIGNAL 및 CALL 절(대상 레이블이 추적된 후 언어 프로세서가 일시정지됨).
- CALL ON 또는 SIGNAL ON이 트래핑하는 조건을 발생시키는 모든 절(CALL 또는 SIGNAL에 대한 대상 레이블이 추적된 후 일시정지가 발생함).
- 구문 오류를 발생시키는 모든 절.(SIGNAL ON SYNTAX에서 트래핑될 수 있지만 다시 실행될 수 없습니다.)

## 예제

```
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.             */
```

## 실행 인터럽트 및 추적 제어

표준 CICS 함수를 사용하여 REXX exec(REXX 트랜잭션)을 인터럽트합니다.

적절한 권한이 부여된 경우, CEMT SET TASK PURGE 명령을 실행하여 exec를 정지할 수 있습니다. 자세한 정보는 [CEMT SET TASK](#)의 내용을 참조하십시오.

## 제 36 장 BMS(Basic Mapping Support) 예

이 예는 REXX/CICS 환경에서 CICS BMS(Basic Mapping Support)를 사용하기 위해 프로시저를 표시합니다.

1. BMS 맵은 어셈블되고 CICS 라이브러리로 링크되어야 합니다. 이 라이브러리는 CICS 리전 시작 JCL의 LIBDEF에 있어야 합니다.
2. 파일 구조를 생성하기 위해 REXX/CICS CONVTPMAP 명령을 사용하려는 경우, BMS 맵은 맵 DSECT를 생성하기 위해 어셈블되어야 합니다.
3. BMS 맵은 자원 Definition Online(RDO)를 사용하여 CICS에 정의되어야 합니다.
4. 화면에 데이터를 읽거나 보내고 싶으면, 맵 입출력(I/O) 영역의 길이를 위한 GETMAIN CICS 스토리지에 필요합니다. 스토리지는 널로 초기화되어야 합니다.
5. 파일 구조 내 필드가 두 레이블 이상 나타내면, 필드를 참조할 때 구조에서 해당 필드에 참조하기 위한 마지막 레이블이 사용되어야 합니다.

이 예는 BMS 맵 PANELG를 사용합니다. 맵핑 정의는 다음과 같습니다.

```

      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
      DFHMDF POS=(1,1),LENGTH=1,ATTRB=(PROT,BRT)                   00000080
      DFHMDF POS=(5,27),LENGTH=22,INITIAL='REXXCICS HEADER PANEL1', *00000090
      ATTRB=(PROT,NORM)                                             00000100
      DFHMDF POS=(5,73),LENGTH=6,INITIAL='PANEL1',ATTRB=(PROT,NORM) 00000110
      DFHMDF POS=(9,6),LENGTH=25,                                   *00000120
      INITIAL='PLEASE ENTER YOUR USERID:',ATTRB=(PROT,NORM)        00000130
* DUSERID                                                            00000140
DUSERID DFHMDF POS=(9,32),LENGTH=8,ATTRB=(UNPROT,BRT,IC,FSET)      00000150
      DFHMDF POS=(9,41),LENGTH=1,ATTRB=(PROT,NORM)                 00000160
      DFHMDF POS=(14,6),LENGTH=4,INITIAL='MSG:',ATTRB=(PROT,NORM)   00000170
* DMSG                                                                00000180
DMSG    DFHMDF POS=(14,11),LENGTH=29,ATTRB=(UNPROT,NORM,FSET)      00000190
      DFHMDF POS=(14,41),LENGTH=1,ATTRB=(PROT,NORM)                 00000200
      DFHMSD TYPE=FINAL                                             00000210
      END                                                            00000220

```

맵 DSECT는 다음과 같습니다. DSECT는 USER.REXXCICS.MAPS(PANELG)라는 MVS PDS입니다.

```

      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))
DPANEL1O DS 0CL(DPANEL1L+1-(DPANEL1L/DPANEL1L))
      ORG
* * * END OF MAP DEFINITION * * *

```

SPACE 3  
ORG

CONVTMAP 명령은 DSECT을 수행하고 RFS에 저장된 파일 구조를 작성하는 데 사용됩니다. 명령은 다음과 같이 입력됩니다.

```
'CONVTMAP USER.TEST(PANELG) POOL1:\USERS\USER1\PANELG.DATA'
```

다음은 CONVTMAP에서 작성된 파일 구조입니다.

```
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*****
```

다음 예는 exec BSMAP1입니다. 사용자 ID를 요청하는 간단한 패널을 작성합니다.

```
/* 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' DUSERIDO, 'Welcome to REXX/CICS !!'
exit
```



BMSMAP1 exec는 다음 패널을 작성했습니다.

```
                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
```



---

## 제 37 장 참고 목록

### 자세한 정보 참조 위치

CICS TS 및 REXX에 관한 자세한 정보는 다음 정보에서 찾을 수 있습니다.

- [애플리케이션 개발 참조서](#)는 애플리케이션 프로그래밍 명령에 관한 정보를 포함합니다.
- [참조: 시스템 프로그래밍](#)은 시스템 프로그래밍 명령에 관한 정보를 포함합니다.
- *REXX 언어*, *Practical Approach to Programming*(저자 M. F. Cowlshaw)는 REXX 언어에 관한 일반 정보를 제공합니다.
- [CICS 제공 트랜잭션 설명](#)은 CEMT 및 CEDA에 관한 정보를 포함합니다.
- [z/OS MVS Programming: Assembler Services Guide](#).
- [z/OS MVS Programming: 어셈블러 서비스 참조서 ABE-HSP](#).
- [z/OS MVS Programming: 어셈블러 서비스 참조서 IAR-XCT](#).



## 주의사항

이 정보는 미국에서 제공되는 제품 및 서비스용으로 작성된 것입니다. 본 자료는 다른 언어로도 제공될 수 있습니다. 그러나 자료에 접근하기 위해서는 해당 언어로 된 제품 또는 제품 버전의 사본이 필요할 수 있습니다.

IBM은 다른 국가에서 이 책에 기술된 제품, 서비스 또는 기능을 제공하지 않을 수도 있습니다. 현재 사용할 수 있는 제품 및 서비스에 대한 정보는 한국 IBM 담당자에게 문의하십시오. 이 책에서 IBM 제품, 프로그램 또는 서비스를 언급했다고 해서 해당 IBM 제품, 프로그램 또는 서비스만을 사용할 수 있다는 것을 의미하지는 않습니다. IBM의 지적 재산을 침해하지 않는 한, 기능상 동등한 제품, 프로그램 또는 서비스를 대신 사용할 수 있습니다. 그러나 비IBM 제품, 프로그램 또는 서비스의 운영에 대한 평가 및 검증은 사용자의 책임입니다.

IBM은 이 책에서 다루고 있는 특정 내용에 대해 특허를 보유하고 있거나 현재 특허 출원 중일 수 있습니다. 이 책을 제공한다고 해서 특허에 대한 라이선스까지 부여하는 것은 아닙니다. 라이선스에 대한 의문사항은 다음으로 문의하십시오.

07326

서울특별시 영등포구

국제금융로 10, 31FC

한국 아이.비.엠 주식회사

대표전화서비스: 02-3781-7114

2바이트(DBCS) 정보에 관한 라이선스 문의는 한국 IBM에 문의하거나 다음 주소로 서면 문의하시기 바랍니다.

*Intellectual Property Licensing*

*Legal and Intellectual Property Law*

*IBM Japan Ltd.19-21, Nihonbashi-Hakozakicho, Chuo-ku*

*Tokyo 103-8510, Japan*

IBM은 타인의 권리 비침해, 상품성 및 특정 목적에의 적합성에 대한 묵시적 보증을 포함하여(단, 이에 한하지 않음) 묵시적이든 명시적이든 어떠한 종류의 보증 없이 이 책을 "현상태대로" 제공합니다. 일부 국가에서는 특정 거래에서 명시적 또는 묵시적 보증의 면책사항을 허용하지 않으므로, 이 사항이 적용되지 않을 수도 있습니다.

이 정보에는 기술적으로 부정확한 내용이나 인쇄상의 오류가 있을 수 있습니다. 이 정보는 주기적으로 변경되며, 변경된 사항은 최신판에 통합됩니다. IBM은 이 책에서 설명한 제품 및/또는 프로그램을 사전 통지 없이 언제든지 개선 및/또는 변경할 수 있습니다.

이 정보에서 언급되는 비IBM 웹 사이트는 단지 편의상 제공된 것으로, 어떤 방식으로든 이들 웹 사이트를 옹호하고자 하는 것은 아닙니다. 해당 웹 사이트의 자료는 본 IBM 제품 자료의 일부가 아니므로 해당 웹 사이트 사용으로 인한 위험은 사용자 본인이 감수해야 합니다.

IBM은 귀하의 권리를 침해하지 않는 범위 내에서 적절하다고 생각하는 방식으로 귀하가 제공한 정보를 사용하거나 배포할 수 있습니다.

(i) 독립적으로 작성된 프로그램과 기타 프로그램(본 프로그램 포함) 간의 정보 교환 및 (ii) 교환된 정보의 상호 이용을 목적으로 본 프로그램에 관한 정보를 얻고자 하는 라이선스 사용자는 다음 주소로 문의하십시오.

07326

서울특별시 영등포구

국제금융로 10, 31FC

한국 아이.비.엠 주식회사

대표전화서비스: 02-3781-7114

이러한 정보는 해당 조건(예를 들면, 사용료 지불 등)하에서 사용될 수 있습니다.

이 정보에 기술된 라이선스가 부여된 프로그램 및 프로그램에 대해 사용 가능한 모든 라이선스가 부여된 자료는 IBM이 IBM 기본 계약, IBM 프로그램 라이선스 계약(IPLA) 또는 이와 동등한 계약에 따라 제공한 것입니다.

비IBM 제품에 관한 정보는 해당 제품의 공급업체, 공개 자료 또는 기타 범용 소스로부터 얻은 것입니다. IBM에서는 이러한 제품들을 테스트하지 않았으므로, 비IBM 제품과 관련된 성능의 정확성, 호환성 또는 기타 청구에 대해서는 확신할 수 없습니다. 비IBM 제품의 성능에 대한 의문사항은 해당 제품의 공급업체에 문의하십시오.

이 정보에는 일상의 비즈니스 운영에서 사용되는 자료 및 보고서에 대한 예제가 들어 있습니다. 이들 예제에는 개념을 가능한 완벽하게 설명하기 위하여 개인, 회사, 상표 및 제품의 이름이 사용될 수 있습니다. 이들 이름은 모두 가공의 것이며 실제 인물 또는 기업의 이름과 유사하더라도 이는 전적으로 우연입니다.

#### 저작권 라이선스:

이 정보에는 여러 운영 플랫폼에서의 프로그래밍 기법을 보여주는 원어로 된 샘플 응용프로그램이 들어 있습니다. 귀하는 이러한 샘플 프로그램의 작성 기준이 된 운영 플랫폼의 애플리케이션 프로그래밍 인터페이스(API)에 부합하는 애플리케이션을 개발, 사용, 판매 또는 배포할 목적으로 IBM에 추가 비용을 지불하지 않고 이들 샘플 프로그램을 어떠한 형태로든 복사, 수정 및 배포할 수 있습니다. 이러한 샘플 프로그램은 모든 조건하에서 완전히 테스트된 것은 아닙니다. 따라서 IBM은 이들 샘플 프로그램의 신뢰성, 서비스 가능성 또는 기능을 보증하거나 진술하지 않습니다. 샘플 프로그램은 어떠한 종류의 보증없이 "현상태대로" 제공됩니다. IBM은 귀하의 샘플 프로그램 사용과 관련되는 손해에 대해 책임을 지지 않습니다.

#### 프로그래밍 인터페이스 정보

이 서적은 REXX/CICS 해석기를 위한 프로그램 작성에 도움을 주기 위한 것입니다. 이 서적은 CICS(Customer Information Control System)용 REXX(CICS용 REXX)에서 제공된 일반용 프로그래밍 인터페이스와 연관 안내 정보를 주로 문서화합니다. 일반용 프로그래밍 인터페이스를 사용하여 고객이 CICS용 REXX의 서비스를 제공하는 프로그램을 작성할 수 있습니다. 그러나 이 서적은 CICS용 REXX에서 제공된 제품별 프로그래밍 인터페이스와 관련 지침 정보도 문서화합니다. 제품별 프로그래밍 인터페이스를 사용하여 고객 설치로 CICS용 REXX의 진단, 수정, 모니터링, 수리, 사용자 조정 또는 튜닝과 같은 태스크를 수행할 수 있습니다. 이러한 인터페이스를 사용하여 IBM 소프트웨어 제품의 상세 설계나 구현에 관한 종속성을 작성합니다.

제품 관련 프로그래밍 인터페이스는 이러한 특수 용도로만 사용해야 합니다. 세부 설계 및 구현에 대한 종속성으로 인해, 이러한 인터페이스를 위해 작성된 프로그램을 새 제품 릴리스나 버전 또는 서비스 결과로 실행하려면 프로그램을 변경해야 할 수 있습니다.

제품별 프로그래밍 인터페이스 및 연관 지침 정보는 절이나 섹션에 소개 부분에서 확인할 수 있습니다.

#### 상표

IBM, IBM 로고 및 ibm.com®은 전세계 여러 국가에 등록된 International Business Machines Corp.의 상표 또는 등록상표입니다. 기타 제품 및 서비스 이름은 IBM 또는 타사의 상표입니다. 현재 IBM 상표 목록은 웹 [저작권 및 상표 정보\(www.ibm.com/legal/copytrade.shtml\)](http://www.ibm.com/legal/copytrade.shtml)에 있습니다.

Adobe, Adobe 로고, PostScript 및 PostScript 로고는 미국 또는 기타 국가에서 사용되는 Adobe Systems Incorporated의 등록상표 또는 상표입니다.

Intel, Intel 로고, Intel Inside Inside, Intel Inside 로고, Intel Centrino, Intel Centrino 로고, Celeron, Intel Xeon, Intel SpeedStep SpeedStep, Itanium 및 Pentium은 미국 또는 기타 국가에서 사용되는 Intel Corporation 또는 그 계열사의 상표 또는 등록상표입니다.

Java™ 및 모든 Java 기반 상표와 로고는 Oracle 및/또는 그 계열사의 상표 또는 등록상표입니다.

Linux®는 미국 또는 기타 국가에서 사용되는 Linus Torvalds의 등록상표입니다.

Microsoft, Windows, Windows NT 및 Windows 로고는 미국 또는 기타 국가에서 사용되는 Microsoft Corporation의 상표입니다.

UNIX는 미국 또는 기타 국가에서 사용되는 The Open Group의 등록상표입니다.

#### 제품 문서의 이용 약관

다음 이용 약관에 따라 이 책을 사용할 수 있습니다.

##### 적용성

본 이용 약관은 IBM 웹 사이트의 모든 이용 약관에 추가됩니다.

##### 개인적 사용

모든 소유권 사항을 표시하는 경우에 한하여 귀하는 이 책을 개인적, 비상업적 용도로 복제할 수 있습니다. 귀하는 IBM의 명시적 동의 없이 본 발행물 또는 그 일부를 배포 또는 전시하거나 2차적 저작물을 만들 수 없습니다.

## 상업적 사용

모든 소유권 사항을 표시하는 경우에 한하여 귀하는 이 책을 귀하 기업집단 내에서만 복제, 배포 및 전시할 수 있습니다. 귀하는 귀하의 기업집단 외에서는 IBM의 명시적 동의 없이 이 책의 2차적 저작물을 만들거나 이 책 또는 그 일부를 복제, 배포 또는 전시할 수 없습니다.

## 권한

본 허가에서 명시적으로 부여된 경우를 제외하고, 이 책이나 이 책에 포함된 정보, 데이터, 소프트웨어 또는 기타 지적 재산권에 대한 어떠한 허가나 라이선스 또는 권한도 명시적 또는 묵시적으로 부여되지 않습니다.

IBM은 본 발행물의 사용이 IBM의 이익을 해친다고 판단되거나 위에서 언급된 지시사항이 준수되지 않는다고 판단하는 경우 언제든지 이 사이트에서 부여한 허가를 철회할 수 있습니다.

귀하는 미국 수출법 및 관련 규정을 포함하여 모든 적용 가능한 법률 및 규정을 철저히 준수하는 경우에만 본 정보를 다운로드, 송신 또는 재송신할 수 있습니다.

IBM은 이 책의 내용과 관련하여 아무런 보장을 하지 않습니다. 타인의 권리 침해, 상품성 및 특정 목적에의 적합성에 대한 묵시적 보증을 포함하여 (단 이에 한하지 않음) 묵시적이든 명시적이든 어떠한 종류의 보증 없이 현 상태대로 제공합니다.

## IBM 온라인 개인정보처리방침

서비스 솔루션 소프트웨어를 비롯한 IBM 소프트웨어 제품("소프트웨어 오퍼링")은 제품 사용 정보 수집, 일반 사용자 편의성 향상, 일반 사용자와의 상호작용 조정 및 기타 목적을 위해 쿠키 또는 기타 기술을 사용할 수 있습니다. 많은 경우에 있어서, 소프트웨어 오퍼링은 개인 식별 정보를 수집하지 않습니다. IBM의 일부 소프트웨어 오퍼링은 귀하가 개인 식별 정보를 수집하도록 도울 수 있습니다. 본 소프트웨어 오퍼링이 쿠키를 사용하여 개인 식별 정보를 수집할 경우, 본 오퍼링의 쿠키 사용에 대한 특정 정보가 다음에 규정되어 있습니다.

이 소프트웨어 오퍼링에 대해 배치된 구성이 고객님의 귀하에게 쿠키 및 기타 기술을 통해 일반 사용자로부터 개인적으로 식별 가능한 정보를 수집하는 기능을 제공하는 경우에는 공지사항 및 동의에 대한 요구사항을 포함하여 해당 데이터 컬렉션에 적용할 수 있는 법률에 대한 자체 법률 자문을 구해야 합니다.

해당 용도로 쿠키를 비롯한 다양한 기술을 사용하는 데 관한 자세한 정보는 [IBM 개인정보처리방침](#) 및 [IBM 온라인 개인정보처리방침](#), "쿠키, 웹 비콘 및 기타 기술"이라는 절과 [IBM 소프트웨어 제품 및 Software-as-a-Service 개인정보처리방침](#)을 참조하십시오.





# 색인

## 특수 문자

! TRACE 옵션 위의 접두부 [174](#)  
? TRACE 옵션 위의 접두부 [174](#)  
.DEFINE verb [306](#), [307](#)  
.PANEL verb [309](#), [310](#)  
\*. \* 추적 플래그 [174](#)  
\*(곱하기 연산자) [20](#), [138](#), [226](#)  
\*\*(거듭제곱 연산자) [20](#), [138](#)  
/(나누기 연산자) [20](#), [138](#), [226](#)  
// (나머지 연산자) [20](#), [138](#)  
/(= (같지 않음 연산자) [138](#)  
/==(엄격하게 같지 않음 연산자) [138](#)  
\ (NOT 연산자) [23](#), [24](#), [138](#)  
\< (보다 작지 않음 연산자) [23](#), [138](#)  
\<< (엄격하게 보다 작지 않음 연산자) [138](#)  
\= (같지 않음 연산자) [22](#), [138](#)  
\==(엄격하게 같지 않음 연산자) [22](#), [138](#)  
\> (보다 크지 않음 연산자) [23](#), [138](#)  
\>> (엄격하게 보다 크지 않음 연산자) [138](#)  
& (AND 논리 연산자) [24](#), [139](#)  
&& (배타적 OR 연산자) [24](#), [139](#)  
%(정수 나누기 연산자) [20](#), [138](#)  
+(더하기 연산자) [20](#), [138](#), [226](#)  
+++ 추적 플래그 [174](#)  
< (미만 연산자) [22](#), [138](#)  
<< (엄격하게 미만 연산자) [138](#)  
<< (엄격하게 보다 적음 연산자) [138](#)  
<<= (엄격하게 이하 연산자) [138](#)  
<= (이하 연산자) [23](#), [138](#)  
<> (미만 또는 초과 연산자) [138](#)  
= (등호) [22](#)  
==(엄격하게 같음 연산자) [22](#), [138](#), [226](#)  
> (초과 연산자) [22](#), [138](#)  
>. > 추적 플래그 [174](#)  
>< (초과 또는 미만 연산자) [22](#), [138](#)  
>= (이상 연산자) [22](#)  
>= (초과 또는 같음 연산자) [138](#)  
>> (엄격하게 초과 연산자) [138](#)  
>>= (엄격하게 이상 연산자) [138](#)  
>>> 추적 플래그 [174](#)  
>C> 추적 플래그 [174](#)  
>F> 추적 플래그 [174](#)  
>L> 추적 플래그 [174](#)  
>O> 추적 플래그 [174](#)  
>P> 추적 플래그 [174](#)  
>V> 추적 플래그 [174](#)  
~ (NOT 연산자) [23](#), [139](#)  
~< (보다 작지 않음 연산자) [138](#)  
~<< (엄격하게 보다 작지 않음 연산자) [138](#)  
~= (같지 않음 연산자) [138](#)  
~==(엄격하게 같지 않음 연산자) [138](#)  
~> (보다 크지 않음 연산자) [138](#)  
~>> (엄격하게 보다 크지 않음 연산자) [138](#)  
| (포함적 OR 연산자) [24](#), [139](#)  
|| (연결 연산자) [26](#), [137](#)

## 숫자

10, 거듭제곱 [230](#)

## A

ABBREV 함수로 기본값 선택 [182](#)  
abuttal [26](#), [137](#)  
ADDRESS 명령어  
예제 [8](#)  
ALLOC 명령 [331](#)  
AND, 논리 연산자 [139](#)  
ARBCHAR 명령 [242](#)  
ARG 명령어 [14](#), [69](#)  
ARG 함수로 인수 검사 [183](#)  
ARGS 명령 [242](#)  
AUTH 명령 [269](#)  
AUTHUSER 명령 [115](#), [121](#), [332](#)

## B

BACKWARD 명령 [243](#)  
BASSM 어셈블러 명령어 [292](#)  
BMS 예 [449](#)  
BMS(Basic Mapping Support) [149](#), [350](#)  
BMSMAP1 exec [449](#)  
BOTTOM 명령 [243](#)  
BSM 어셈블러 명령어 [292](#)

## C

C2S 명령 [292](#), [303](#), [354](#)  
CALL 명령어 [47](#), [63](#)  
CANCEL 명령 [243](#), [259](#)  
CASE 명령 [244](#)  
CATMOUSE EXEC [98](#)  
CD 명령 [146](#), [239](#), [268](#), [333](#), [360](#)  
CECI(Command Level Interpreter Transaction) [121](#)  
CEDA 명령 [334](#)  
CEMT 명령 [335](#)  
CHANGE 명령 [245](#)  
CICEPROF 매크로 [248](#)  
CICGETV 루틴 [293](#)  
CICPARMS 제어 블록 [292](#)  
CICREX 프로그램 [116](#)  
CICREX1106E [400](#)  
CICREX218E [392](#)  
CICREX219E [392](#)  
CICREX255T [392](#)  
CICREX449E [392](#)  
CICREX450E [392](#)  
CICREX451E [393](#)  
CICREX452E [393](#)  
CICREX453E [393](#)  
CICREX454E [393](#)  
CICREX455E [393](#)

CICREX456E [393](#)  
CICREX457E [394](#)  
CICREX458E [394](#)  
CICREX459E [394](#)  
CICREX460E [394](#)  
CICREX461E [395](#)  
CICREX462E [395](#)  
CICREX463E [395](#)  
CICREX464E [395](#)  
CICREX465E [395](#)  
CICREX466E [396](#)  
CICREX467E [396](#)  
CICREX468E [396](#)  
CICREX469E [396](#)  
CICREX470E [396](#)  
CICREX471E [396](#)  
CICREX472E [397](#)  
CICREX473E [397](#)  
CICREX474E [397](#)  
CICREX475E [397](#)  
CICREX476E [397](#)  
CICREX477E [398](#)  
CICREX478E [398](#)  
CICREX479E [398](#)  
CICREX480E [398](#)  
CICREX481E [398](#)  
CICREX482E [398](#)  
CICREX483E [399](#)  
CICREX484E [399](#)  
CICREX485E [399](#)  
CICREX486E [399](#)  
CICREX487E [399](#)  
CICREX488E [399](#)  
CICREX489E [400](#)  
CICREX490E [400](#)  
CICREX491E [400](#)  
CICREX492E [400](#)  
CICS

리턴 코드 [428](#)

CICS 개요 [125](#)

CICS 메시지

REXX 오류 코드 [391](#)

CICS 명령 [85](#)

CICS 초기화 JCL 업데이트 [111](#)

CICSECX1 보안 엑시트 [116](#)

CICSECX2 보안 엑시트 [117](#)

CICS PROF exec [121](#)

CICSTART exec [115](#), [116](#)

CICSTART.PROC 업데이트 [110](#)

CICXPROF 매크로 [248](#)

CKDIR 명령 [270](#), [284](#)

CKFILE 명령 [270](#)

CLD 명령 [284](#), [349](#)

CMDLINE 명령 [245](#)

command

따옴표 [85](#)

COMPARE 함수 [185](#)

CON EXEC (exercise) [97](#)

CONVTMAP 명령 [350](#)

COPIES로 문자열 반복 [186](#)

COPIES를 사용하여 문자열 복사 [186](#)

COPY 명령 [270](#)

COPYR2S 명령 [351](#)

COPYS2R 명령 [352](#)

CTLCHAR 명령 [246](#)

CURLINE 명령 [246](#)

## D

DATATYPE으로 검사하는 데이터 유형 [188](#)

DATATYPE으로 검사하는 영숫자 [188](#)

DATATYPE을 사용하여 검사된 비트 [188](#)

DATE 함수의 Base 옵션 [189](#)

DATE 함수의 Century 옵션 [189](#)

DATE 함수의 European 옵션 [189](#)

DATE 함수의 Julian 옵션 [189](#)

DATE 함수의 Month 옵션 [189](#)

DATE 함수의 Normal 옵션 [189](#)

DATE 함수의 Ordered 옵션 [189](#)

DATE 함수의 Standard 옵션 [189](#)

DATE 함수의 Usa 옵션 [189](#)

DATE 함수의 Weekday 옵션 [189](#)

Db2 인터페이스 [295](#)

Db2 임베드 [299](#)

DBADJUST 함수 [440](#)

DBBRACKET 함수 [440](#)

DBCENTER 함수 [440](#)

DBCJUSTIFY 함수 [441](#)

DBCS

문자 [431](#)

문자열 [431](#)

설명 [431](#)

이름, 사용 [10](#)

지원 [431](#)

처리 [431](#)

처리 함수 [439](#)

함수 처리 [435](#)

DBCS 기호 유효성 검증 [432](#)

DBCS의 기호와 문자열 [432](#)

DBLEFT 함수 [441](#)

DBRIGHT 함수 [442](#)

DBRLEFT 함수 [442](#)

DBRRIGHT 함수 [442](#)

DBTODBCS 함수 [443](#)

DBTOSBCS 함수 [443](#)

DBUNBRACKET 함수 [443](#)

DBVALIDATE 함수 [443](#)

DBWIDTH 함수 [444](#)

DEFCMD 명령 [121](#), [291](#), [303](#), [355](#)

DEFCMD 명령의 CICSLINK 옵션 [292](#)

DEFCMD 명령의 CICSLOAD 옵션 [292](#)

DEFSCMD 명령 [291](#), [357](#)

DEFTRNID 명령 [359](#)

DELETE 명령 [271](#), [285](#)

DIR 명령 [10](#), [360](#)

DISKR 명령 [271](#)

DISKW 명령 [267](#), [272](#)

DISPLAY 명령 [247](#)

DO 명령어에서 FOREVER 리피티터 [155](#)

DO 명령어의 BY 구문 [155](#)

DO 명령어의 FOR 구문 [155](#)

DO 명령어의 TO 구문 [155](#)

DO 명령어의 UNTIL 구문 [155](#)

DO 명령어의 WHILE 구문 [155](#)

DO...END 명령어 [37](#)

DOWN 명령 [247](#)

DPATH 명령 [268](#)

## E

EDIT 명령 [239](#), [248](#), [361](#)  
EDITSVR [86](#)  
ETMODE [165](#), [432](#)  
EXEC 명령 [249](#), [361](#)  
exec ID [8](#)  
EXECDB2 명령 환경 [295](#)  
EXECDROP 명령 [362](#)  
EXECIO 명령 [363](#)  
EXECLOAD 명령 [364](#)  
EXECMAP 명령 [366](#)  
EXECSQL 명령 환경 [295](#)  
EXIT 명령어 [46](#), [63](#)  
EXMODE [432](#)  
EXPORT 명령 [366](#)

## F

FILE 명령 [250](#)  
FILEPOOL 명령 [116](#), [368](#)  
FIND 명령 [250](#)  
FIND 함수 [192](#)  
FLST  
    트랜잭션 [281](#)  
    exec [281](#)  
FLST 명령  
    CANCEL [275](#)  
    COPY [275](#)  
    DELETE [276](#)  
    DOWN [276](#)  
    END [276](#), [281](#)  
    EXEC [276](#)  
    FLST [277](#)  
    MACRO [278](#)  
    PFKEY [278](#)  
    REFRESH [279](#)  
    RENAME [279](#)  
    SORT [280](#)  
    SYNONYM [280](#)  
    UP [281](#)  
FLSTSVR [86](#)  
FORM 함수 [193](#)  
FORWARD 명령 [251](#)  
FREE 명령 [370](#)  
FUZZ 함수 [194](#)

## G

GET 명령 [252](#)  
GETDIR 명령 [272](#)  
GETPDS 명령 [252](#)  
GETVERS 명령 [371](#)

## H

HELLO EXEC [11](#)  
HELP 명령 [371](#)  
HI(Halt Interpretation) 즉각적 명령 [447](#)

## I

IMPORT 명령 [371](#)

INDEX [194](#)  
INPUT 명령 [253](#)

## J

JOIN 명령 [253](#)  
JUSTIFY 함수로 텍스트 자리 맞추기 [195](#)

## L

LEAVE 명령어 [45](#)  
LEFT 명령 [254](#)  
LINEADD 명령 [254](#)  
LISTCMD 명령 [373](#)  
LISTPOOL 명령 [374](#)  
LISTTRNID 명령 [374](#)  
LPREFIX 명령 [255](#)  
LPULL 명령 [285](#)  
LPUSH 명령 [285](#)  
LQUEUE 명령 [286](#)  
LSRPOOL 정의 업데이트 [110](#)

## M

MACRO 명령 [255](#)  
MKDIR 명령 [267](#), [273](#), [283](#), [286](#)  
MSGLINE 명령 [256](#)

## N

NOETMODE [165](#)  
NOEXMODE [165](#)  
NOT 연산자 [132](#), [139](#)  
NULLS 명령 [256](#)  
NUMBERS 명령 [257](#)  
NUMERIC 명령어의 DIGITS 옵션 [164](#), [226](#)  
NUMERIC 명령어의 FORM 옵션 [164](#), [230](#)

## O

OPTIONS 명령어 [432](#)

## P

PANEL 명령 [311](#), [312](#)  
PARSE 명령어의 ARG 옵션 [166](#)  
PARSE 명령어의 EXTERNAL 옵션 [166](#)  
PARSE 명령어의 LINEIN 옵션 [166](#)  
PARSE 명령어의 PULL 옵션 [166](#)  
PARSE 명령어의 SOURCE 옵션 [166](#)  
PARSE 명령어의 VALUE 옵션 [166](#)  
PARSE 명령어의 VAR 옵션 [166](#)  
PARSE 명령어의 VERSION 옵션 [166](#)  
PARSE ARG 명령어 [78](#)  
PARSE PULL 명령어 [77](#)  
PARSE UPPER ARG 명령어 [78](#)  
PARSE UPPER PULL 명령어 [78](#)  
PARSE UPPER VALUE [78](#)  
PARSE UPPER VAR [79](#)  
PATH 명령 [146](#), [268](#), [375](#)  
PFKEY 명령 [257](#)  
PFKLINE 명령 [258](#)

PLT(Program List Table) [116](#)  
PROCEDURE 명령어 [67](#), [68](#)  
PROCEDURE 명령의 EXPOSE 옵션 [168](#)  
PSEUDO 명령 [376](#)  
PULL 명령어 [11](#), [14](#), [77](#)

## Q

QQUIT 명령 [258](#), [260](#)  
QUERY 명령 [259](#)  
QUIT 명령 [260](#)

## R

RANDOM의 난수 함수 [198](#)  
RANDOM의 의사 난수 함수 [198](#)  
RC [445](#)  
RDIR 명령 [273](#)  
READ 명령 [287](#)  
RENAME 명령 [273](#)  
RESERVED 명령 [260](#), [274](#)  
RESET 명령 [261](#)  
RESULT [445](#)  
RETURN 명령어 [47](#), [63](#)  
REXX  
    절 [8](#)  
REXX 명령어  
    구문 [5](#), [6](#)  
    형식화 [5](#)  
    PROCEDURE [67](#)  
    PULL [14](#)  
REXX 언어  
    기능의 [3](#)  
REXX 프로그램  
    변수 [17](#)  
    연산자 [17](#)  
    표현식 [17](#)  
REXX 프로그램 ID [5](#), [8](#), [131](#)  
REXX 프로그램에 전달된 명령 인수 [292](#)  
REXX Db2 인터페이스 구성 [114](#)  
REXX/CICS [401](#)  
REXX/CICS 명령 [331](#)  
REXX/CICS 텍스트 편집기 [239](#)  
REXX/CICS 파일 시스템(RFS) [267](#)  
REXX/CICS 패널 기능 [305](#)  
REXX/CICS Command Definition Facility [291](#)  
REXXCICS [86](#)  
RFS [86](#)  
RFS 명령 [267](#), [377](#)  
RFS 파일 폴 작성 [109](#)  
RHS 파일 폴 형식화 [112](#)  
RIGHT 명령 [261](#)  
RLS [86](#)  
RLS 명령  
    CKDIR [284](#)  
    DELETE [285](#)  
    LPULL [285](#)  
    LPUSH [285](#)  
    LQUEUE [286](#)  
    MKDIR [286](#)  
    READ [287](#)  
    VARDROP [287](#)  
    VARGET [288](#)

RLS 명령 (계속)  
    VARPUT [288](#)  
    WRITE [289](#)  
RLS(REXX/CICS List System) [283](#)

## S

S2C 명령 [292](#), [303](#), [386](#)  
SAVE 명령 [262](#)  
SAY 명령 [11](#)  
SAY 명령어 [5](#), [11](#), [148](#)  
SBA(Set Buffer Address) [209](#)  
SBCS 문자열 [431](#)  
SCRNINFO 명령 [382](#)  
SELECT WHEN...OTHERWISE...END [34](#)  
SET 명령 [382](#)  
SETSYS 명령 [116](#), [385](#)  
shift-in (SI) 문자 [431](#)  
shift-out (SO) 문자 [431](#)  
SI(Shift-in) 문자 [435](#)  
SIGL [445](#)  
SIGNAL 명령어 [50](#)  
SIGNAL 명령어의 SYNTAX 조건 [233](#), [235](#)  
SIGNAL 및 CALL 명령어의 ERROR 조건 [235](#)  
SIGNAL 및 CALL 명령어의 FAILURE 조건 [233](#), [235](#)  
SIGNAL 및 CALL 명령어의 HALT 조건 [233](#), [235](#)  
SO(Shift-out) 문자 [435](#)  
SORT 명령 [263](#)  
SPLIT 명령 [263](#)  
SQL 임베드 [295](#)  
SQL 통신 영역(SQLCA) [298](#)  
SQL문 [85](#)  
STRIP 명령 [264](#)  
SUBSTR [200](#)  
SYNONYM 명령 [264](#)  
SYSSBA 함수 [209](#)

## T

TE(Trace End) 즉각적 명령 [447](#)  
TERMINAL 명령 [387](#)  
TOP 명령 [264](#)  
TRACE 명령어 [148](#)  
TRACE 설정 조회 [203](#)  
TRANSLATE 함수를 사용하여 데이터 다시 정렬 [203](#)  
TRUNC 명령 [265](#)  
TS(Trace Start) 즉각적 명령 [447](#)

## U

UP 명령 [265](#)  
USERS 디렉토리 [267](#), [283](#)

## V

VARDROP 명령 [287](#)  
VARGET 명령 [288](#)  
VARPUT 명령 [288](#)  
verb  
    DEFINE [306](#)  
    PANEL [309](#)

## W

WAITREAD 명령 [388](#)  
WAITREQ 명령 [292](#), [303](#), [388](#)  
WRITE 명령 [283](#), [289](#)

## X

X2C로 문자열 패킹 [207](#)  
XOR, 논리 [139](#)  
XRANGE을 사용하여 순서 조합 [207](#)

## 가

가정, XEDIT [10](#)  
간격, 형식화, SPACE 함수 [199](#)  
같음, 테스트 [138](#)  
같지 않음 연산자 [138](#)  
같지 않음, 테스트 [138](#)  
개발  
    REXX 프로그램 [1](#)  
개요  
    CICS [125](#)  
경고, STORAGE 함수 [208](#)  
공백, 처리 [79](#)  
구문  
    REXX의 규칙 [5](#), [6](#)  
구문 다이어그램  
    REXX [129](#)  
구문 분석  
    단어로 [79](#)  
    PARSE ARG [78](#)  
    PARSE UPPER ARG [78](#)  
구문 분석 명령어 [218](#)  
구문 분석 시 문자열 패턴 [79](#)  
구문 분석 시 변수 문자열 패턴 [79](#)  
구문 분석 시 상대적인 숫자 패턴 [79](#)  
구문 분석 시 패턴 [79](#)  
구문 분석에서 플레이스홀더 [14](#), [79](#)  
구문 분석의 개념 개요 [221](#)  
구문 분석의 고급 주제 [219](#)  
구문 분석하는 경우 [220](#)  
구문을 위한 문자열 검색 [192](#)  
구성  
    REXX [109](#)  
구조 및 구문 [131](#)  
규칙  
    구문 [5](#), [6](#)  
    자유 형식 [5](#), [6](#)  
기능  
    REXX의 [3](#)

## 나

내재적 세미콜론 [136](#)  
노출된 변수 [168](#)  
니블 [132](#)

## 다

다중 문자열 [81](#), [219](#)  
다중 방향 호출 [153](#), [173](#)  
대수적 우선순위 [140](#)

대화식 디버그 [174](#), [447](#)  
데이터의 간접 평가 [161](#)  
데이터의 해석적 실행 [161](#)  
도움말 파일 작성 [95](#), [113](#)  
디렉토리 ID [267](#), [283](#)  
디버그 지원 [447](#)

## 라

루트 디렉토리 [267](#), [283](#)  
리터럴 문자열 [6](#), [132](#)  
리턴 코드 [401](#)

## 마

마스터 터미널 트랜잭션(CEMT) [3](#)  
막연한 루프 [155](#)  
메시지  
    해석 [12](#)  
명령  
    EDIT [85](#)  
    RFS [85](#)  
    RLS [85](#)  
명령 사이에 맵핑 [331](#)  
명령 실행 보안 [269](#)  
명령 환경 재설정 [331](#)  
명령문  
    SQL [85](#)  
명령어  
    PROCEDURE [67](#)  
    SAY [5](#)  
무조건적으로 프로그램 중단 [160](#)  
무한 루프 [155](#)  
문서 편집기 [239](#)  
문자열  
    DBCS [431](#)  
문자열 구문 분석 [219](#)  
문자열과 위치 패턴 [220](#)  
문자열과 위치 패턴 결합 [220](#)  
문자열로부터의 단어 [205](#)  
문자열을 다른 문자열에 오버레이 [197](#)  
문자열을 다른 항목에 삽입 [194](#)  
미만 또는 초과 연산자(<>) [138](#)  
미만 연산자 (<) [138](#)

## 바

반복 루프 [37](#)  
반복적으로 실행하기 위해 명령어 그룹으로 지정 [155](#)  
배타적 OR 연산자 [24](#), [139](#)  
백슬래시, 사용 [132](#), [138](#)  
변수  
    설명 [17](#)  
변수 문자열 패턴 [216](#)  
변수 보호 [168](#)  
변수 복원 [159](#)  
변수 지정 해제 [159](#)  
변수에서 파생된 이름 [143](#)  
변수의 값, VALUE로 가져오기 [204](#)  
보다 작지 않음 연산자 [138](#)  
보다 크지 않음 연산자 [138](#)  
보안  
    파일 액세스 [269](#)

보안 엑시트 [116](#)  
보조 목록 [159](#), [168](#)  
보호 숫자 [226](#)  
부정확한 숫자 비교 [229](#)  
빈 행 [9](#)

## 사

사용자 식별 [204](#)  
사용자, 식별 [204](#)  
사전 정의 변수 [295](#), [300](#)  
사전 정의된 변수 [297](#)  
산술 [225](#)  
산술 연산자  
  유형 [19](#)  
산술 정밀도 [226](#)  
산술의 유효 숫자 [226](#)  
상대적인 위치 패턴 [213](#)  
상수 [18](#)  
상수 기호 [143](#)  
서버 exec 예제 [304](#)  
서브 키워드 [142](#)  
서브디렉토리 [267](#), [283](#)  
서브루틴  
  변수 보호 [67](#)  
  작성 [59](#)  
  함수에 대한 비교 [59](#)  
서브루틴 호출 중 저장된 예외 조건 [153](#)  
서브루틴에서 다중 문자열 구문 분석 [219](#)  
선입선출(FIFO) 스택화 [170](#)  
설명  
  변수 [17](#)  
설치 확인 [112](#)  
순서, XRANGE를 사용하여 조합 [207](#)  
숫자 자르기 [203](#)  
숫자에서 10의 거듭제곱 [132](#)  
섬표 [79](#)  
스타일, 코딩 [103](#)  
시스템 관리자 [121](#)  
실패, 정의 [145](#)  
실행  
  REXX 프로그램 [5](#)  
실행 중 도달된 프로그램 맨 아래 [160](#)

## 아

어셈블러 프로그램  
  전달된 명령 인수 [292](#)  
어셈블러 프로그램에 전달되는 명령 인수 [292](#)  
언더플로우, 산술 [232](#)  
언어 프로세서의 버전 및 날짜 [166](#)  
엄격하게 같음 연산자 [138](#)  
엄격하게 미만 연산자 [138](#)  
엄격하게 보다 적음 연산자 [138](#)  
엄격하게 보다 크지 않음 연산자 [138](#)  
엄격하게 적지 않음 연산자 [138](#)  
엄격하게 초과 연산자 [138](#)  
엄격하지 같지 않음 연산자 [138](#)  
엄격한 비교 [138](#)  
엄밀하게 이상 연산자 [138](#)  
엄밀하게 이하 연산자 [138](#)  
엔지니어링 표기법 [230](#)  
역 추적, 구문 오류 [174](#)

연관 스토리지 [143](#)  
연산자  
  산술 [19](#)  
  연결 [26](#)  
연산자의 우선순위 [140](#)  
연습  
  산술 연산식 계산 [21](#)  
  함수 작성 [74](#)  
영구 명령 대상 변경 [151](#)  
예제  
  간단한 REXX 프로그램 [5](#)  
  디버그 지원 [447](#)  
  샘플 패널 [323](#)  
  완전한 파일 ID [267](#)  
  현재 디렉토리 [268](#)  
  ADDRESS 명령어 [8](#)  
  DBCS 이름 사용 [10](#)  
오류  
  디버깅 [89](#), [90](#)  
오류 메시지 [12](#)  
오류 메시지 전에 지워진 NOTYPING 플래그 [391](#)  
오류 코드 [391](#)  
오버플로우, 산술 [232](#)  
온라인 자원 정의(RDO) [449](#)  
완전한 파일 ID [239](#)  
위치 패턴을 포함하는 템플릿 [213](#)  
이하 연산자 (<=) [138](#)  
인스트루멘테이션 기능 인터페이스(IFI) [295](#)  
인터럽트 명령어 [46](#)  
일반 개념 [131](#)  
임시 명령 대상 변경 [151](#)  
임시 저장영역 큐(TSQ) [363](#)  
입력  
  대문자로의 변환 방지 [15](#)

## 자

자동 서버 초기화(ASI) [303](#)  
자리맞춤, 텍스트 오른쪽, RIGHT 함수 [198](#)  
자원 정의 설치 [109](#)  
자원 정의 온라인 트랜잭션(CEDA) [3](#)  
자유 형식  
  REXX 명령어 [6](#)  
자정부터 계산된 분 [201](#)  
자정부터 계산된 시간 [201](#)  
자정부터 계산된 초 [201](#)  
작성  
  REXX 프로그램 [5](#)  
재개된 명령 대상 [151](#)  
재귀 호출 [153](#)  
전달  
  인수 [16](#)  
  정보 [66](#)  
절  
  REXX 유형 [8](#)  
정의  
  디렉토리 ID [267](#)  
  루트 디렉토리 [267](#)  
  서브디렉토리 [267](#)  
  파일 풀 [267](#)  
  패널 [306](#)  
정지, 트래핑 [233](#)  
제어 구조의 중첩 [153](#)  
제어 변수 [37](#)

제어 플로우의 비정상적 변경 [233](#)  
제품 개요 [125](#)  
조건 및 데이터 [136](#)  
조건이 트래핑되지 않은 경우 취해지는 조치 [234](#)  
조건이 트랩될 때 수행 조치 [234](#)  
주소 설정 [151, 153](#)  
지수 표기법 [230](#)

## 차

채움 문자, 정의 [181](#)  
초과 또는 같음 연산자(>=) [138](#)  
초과 또는 미만 연산자(><) [138](#)  
초과 연산자 [138](#)  
초기화되지 않은 변수 [142](#)  
추적  
    추적 오퍼레이션 [89, 90](#)  
추적 중 들어쓰기 [174](#)

## 카

컨텐츠 주소 지정 가능한 스토리지 [143](#)  
코드 페이지 [132](#)  
코딩 스타일 [103](#)  
클라이언트 exec 예제 [304](#)  
클라이언트/서버 [119](#)  
클라이언트/서버 지원  
    REXX/CICS [303](#)  
키워드 예약 [445](#)

## 타

터미널의 행 길이 및 너비 [196](#)  
텍스트 간격 [199](#)  
템플릿  
    구문 분석 [79](#)  
트랜잭션 ID [121, 239](#)  
특수 변수 [445](#)  
특수한 경우 [220](#)

## 파

파일 목록 유틸리티(FLST) [274](#)  
파일 시스템  
    명령 [269](#)  
파일 액세스 보안 [269](#)  
파일 유형 확장 [146](#)  
파일 이름, 유형, 프로그램의 모드 [166](#)  
파일 풀  
    루트 디렉토리 [267](#)  
패널  
    생성 [311](#)  
    오브젝트 생성기 [306](#)  
    입출력(I/O) [311](#)  
    정의 [306](#)  
패널 기능 리턴 코드 [317](#)  
패널 기능 상태 코드 [319](#)  
패턴에서 등호 [81](#)  
페이지, 코드 [132](#)  
포함적 OR 연산자 [24, 139](#)  
표현식  
    비교 [22](#)  
    산술 [19](#)

표현식 (계속)  
    연결 [26](#)  
    정의 [19](#)  
    추적 [89, 90](#)  
표현식의 평가 [136](#)  
프로그램  
    구성 [107](#)  
    오류 메시지 [12](#)  
    입력 수신 [14](#)  
    정보 전달 [13](#)  
프로그램 수정 [102](#)  
프로그램 중단 [160](#)  
프로그램 ID [5, 131](#)  
프로그램에서 입력 [10](#)  
플레이스홀더 [79](#)  
플레이스홀더로서 마침표 사용 [79](#)  
플로우차트 [34](#)

## 하

하위 문자열 [200](#)  
하위 표현식 [136](#)  
함께 문자열을 AND 처리 [184](#)  
함께 문자열을 exclusive-OR(배타적 논리합)로 처리 [184](#)  
함께 문자열을 OR로 처리 [184](#)  
함께 문자열을 XOR로 처리 [184](#)  
함수  
    변수 보호 [67](#)  
    서브루틴에 대한 비교 [59](#)  
현재 디렉토리 [268, 284](#)  
현재 터미널 행 너비 [196](#)  
호스트 명령 환경 [86](#)  
혼용 DBCS 문자열 [188](#)  
활성 루프 [162](#)  
후미 [143](#)  
후입선출(LIFO) 스택화 [170](#)







