REXX Grundlagen für die z/OS Praxis

von
Johann Deuring

Oldenbourg Verlag München Wien

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

© 2005 Oldenbourg Wissenschaftsverlag GmbH
Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg.de

Das Werk einschließlich aller Abbildungen ist urheberrechtlich geschützt. Jede Verwertung außerhalb der Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Bearbeitung in elektronischen Systemen.

Lektorat: Stephanie Schumacher-Gebler
Herstellung: Anna Grosser
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: R. Oldenbourg Graphische Betriebe Druckerei GmbH

ISBN 3-486-20025-9

Inhalt

1 Einleitung 1

1.1 Ziel des Buchs ... 1

1.2 Feedback erwünscht ... 1

1.3 Verwendete Syntax-Notationen ... 2

2 Einführung in REXX 5

2.1 Entstehung und Verbreitung von REXX ... 5

2.2 Verwendung ... 5
2.2.1 Einsatzgebiete ... 6
2.2.2 Wer programmiert in REXX? ... 7

3 Sprachaufbau 9

3.1 Formatfreiheit ... 10

3.2 Bestandteile einer REXX-Exec ... 10
3.2.1 Token-Klassen ... 11
3.2.2 String ... 12
3.2.3 Symbole, Variablen, Zahlen ... 13
3.2.4 Arten von Clauses ... 17
3.2.5 Expression ... 19

4 Speicherung und Aufruf 21

4.1 Speicherung eines REXX-Programms ... 22

4.2 Aufruf eines REXX-Programms ... 23
4.2.1 Expliziter Aufruf ... 23
4.2.2 Impliziter Aufruf ... 25
4.2.3 Die Bibliotheksverkettung beim TSO-Logon ... 27
4.2.4 REXX-Exec-Ausführung im Batch (Background) ... 29

5 Erster Dialog 31

5.1 SAY ... 31

5.2 PARSE EXTERNAL ... 32

5.3	DROP	33
5.4	EXIT	35
6	**PARSE**	**37**
6.1	Zerlegbare Speicherbereiche	38
6.2	Das Template	41
6.2.1	Zerlegen nach Wörtern (Blank-Zerlegung)	42
6.2.2	Zerlegen nach Argumenten (Komma-Zerlegung)	44
6.2.3	Zerlegen nach Trennzeichen	45
6.2.4	Zerlegen nach Komma (als Argument und Trennzeichen)	46
6.2.5	Zerlegen nach Spalten mit absoluten Angaben	47
6.2.6	Zerlegen nach Spalten mit relativen Angaben (Offset)	48
6.2.7	Trennzeichen in Variablen	49
6.2.8	Kombinationen	49
7	**Programmablauf verfolgen**	**51**
7.1	TRACE	51
7.2	TRACE-Aktivierung durch den Benutzer	53
7.3	TRACE Output	54
8	**Arithmetik, Vergleiche, Logik, Verkettungen**	**57**
8.1	Arithmetik	57
8.1.1	Arithmetische Operatoren	57
8.1.2	NUMERIC DIGITS	59
8.1.3	NUMERIC FORM	60
8.2	Vergleiche	62
8.2.1	Vergleichsoperatoren für Standard-Vergleiche	62
8.2.2	Vergleichsoperatoren für Exakt-Vergleiche	64
8.2.3	NUMERIC FUZZ	66
8.3	Logik (Boole'sche Algebra)	68
8.4	Verkettungen	70
8.5	Gemischte Ausdrücke	71
9	**Programmsteuerung**	**73**
9.1	Verzweigungen	73
9.1.1	IF	73
9.1.2	SELECT	75
9.2	SIGNAL	77
9.3	DO-Gruppen und -Schleifen	79
9.3.1	Simple DO Group	81

9.3.2 Simple Repetitive DO Loop 83
9.3.3 Controlled Repetitive Loop (Iteration) 85
9.3.4 LEAVE 89
9.3.5 ITERATE 90
9.3.6 Conditional Loop mit DO WHILE 91
9.3.7 Conditional Loop mit DO UNTIL 92
9.3.8 DO FOREVER 93
9.3.9 Loop-Mischformen 94

10 Unterprogramme 95

10.1 Subroutines 95
10.1.1 CALL 95
10.1.2 RETURN und RESULT 98

10.2 Functions 100

10.3 Interne Routinen 102

10.4 Externe Routinen 102

10.5 Variablentransparenz 102

10.6 Das Merken der Programmeinstellungen beim Upro-Aufruf 106
10.6.1 Status von DO-Schleifen und anderen Strukturen 106
10.6.2 TRACE-Status 106
10.6.3 NUMERIC-Einstellungen 106
10.6.4 ADDRESS-Einstellungen 107
10.6.5 Condition Traps 107
10.6.6 Zeitmessungen mit TIME('R') und TIME('E') 107
10.6.7 OPTIONS-Einstellungen 107
10.6.8 Schachtelungstiefe von Unterroutinen 107

10.7 Builtin Functions 108
10.7.1 ABBREV 108
10.7.2 ABS 109
10.7.3 ADDRESS 110
10.7.4 ARG 111
10.7.5 BITAND 113
10.7.6 BITOR 114
10.7.7 BITXOR 115
10.7.8 B2X 116
10.7.9 CENTER 117
10.7.10 COMPARE 118
10.7.11 CONDITION 119
10.7.12 COPIES 121
10.7.13 C2D 122
10.7.14 C2X 124
10.7.15 DATATYPE 125

10.7.16 DATE 127
10.7.17 DBCS-Builtin Functions 131
10.7.18 DELSTR 133
10.7.19 DELWORD 134
10.7.20 DIGITS 136
10.7.21 D2C 136
10.7.22 D2X 138
10.7.23 ERRORTEXT 139
10.7.24 EXTERNALS 140
10.7.25 FIND 140
10.7.26 FORM 141
10.7.27 FORMAT 142
10.7.28 FUZZ 145
10.7.29 GETMSG 146
10.7.30 INDEX 150
10.7.31 INSERT 152
10.7.32 JUSTIFY 154
10.7.33 LASTPOS 155
10.7.34 LEFT 156
10.7.35 LENGTH 157
10.7.36 LINESIZE 158
10.7.37 LISTDSI 159
10.7.38 MAX 170
10.7.39 MIN 171
10.7.40 MSG 172
10.7.41 MVSVAR 173
10.7.42 OUTTRAP 176
10.7.43 OVERLAY 180
10.7.44 POS 181
10.7.45 PROMPT 183
10.7.46 QUEUED 184
10.7.47 RANDOM 185
10.7.48 REVERSE 187
10.7.49 RIGHT 188
10.7.50 SETLANG 189
10.7.51 SIGN 191
10.7.52 SOURCELINE 192
10.7.53 SPACE 193
10.7.54 STORAGE 194
10.7.55 STRIP 197
10.7.56 SUBSTR 199
10.7.57 SUBWORD 200
10.7.58 SYMBOL 201
10.7.59 SYSCPUS 202
10.7.60 SYSDSN 203

10.7.61 SYSVAR ... 205
10.7.62 TIME ... 209
10.7.63 TRACE ... 211
10.7.64 TRANSLATE ... 212
10.7.65 TRUNC ... 213
10.7.66 USERID ... 214
10.7.67 VALUE ... 215
10.7.68 VERIFY ... 216
10.7.69 WORD ... 217
10.7.70 WORDINDEX ... 219
10.7.71 WORDLENGTH ... 220
10.7.72 WORDPOS ... 221
10.7.73 WORDS ... 222
10.7.74 XRANGE ... 223
10.7.75 X2B ... 224
10.7.76 X2D ... 225

10.8 Suchfolge beim Aufruf von Unterroutinen ... 227

11 TSO/E – Commands & Utilities 229

11.1 Command-Übersicht ... 229

11.2 Häufig verwendete TSO-Commands ... 231
11.2.1 EXECUTIL ... 231

11.3 PROFILE ... 232

11.4 ALLOC ... 232

11.5 ALLOC und FREE ... 233

11.6 ALTLIB ... 234

11.7 DELETE-/DEFINE-Cluster ... 234

11.8 LISTC ... 234

11.9 LISTDS ... 235

11.10 REPRO ... 235

11.11 IEBGENER ... 235

11.12 SMCOPY ... 236

11.13 SUBMIT ... 236

11.14 PERMIT ... 236

11.15 TRANSMIT ... 237

11.16 RECEIVE ... 237

11.17 SEND ... 237

12 ISPF-Panels 239

12.1 Äußerer Aufbau ... 239

12.2 Panel Coding – Ein Beispiel ... 240

12.3 Innerer Aufbau ... 241

12.4 Panel-Aufruf aus einer REXX-Exec ... 243

13 SUBCOM 245

14 Stack-Verarbeitung 247

14.1 QUEUE ... 247

14.2 PUSH ... 248

14.3 LIFO und FIFO ... 248

14.4 PARSE PULL ... 249

14.5 MAKEBUF ... 250

14.6 DROPBUF ... 251

14.7 QBUF ... 252

14.8 QELEM ... 253

14.9 NEWSTACK ... 254

14.10 QSTACK ... 255

14.11 DELSTACK ... 256

15 Dateiverarbeitung mit EXECIO 257

15.1 Allgemeines ... 258

15.2 DISKR ... 258
15.2.1 Massenlesen in den Stack ... 259
15.2.2 Massenlesen in Compound-Variablen ... 260
15.2.3 Einzellesen in eine Variable ... 261
15.2.4 Einzellesen in den Stack ... 262

15.3 DISKW ... 262
15.3.1 Massenschreiben aus dem Data Stack ... 263
15.3.2 Massenschreiben von Compound-Variablen ... 264
15.3.3 Massenschreiben auf neue PS-Datei ... 265
15.3.4 Einzelschreiben auf Member einer neuen PO-Datei ... 267
15.3.5 Rewrite mit DISKRU + DISKW ... 268
15.3.6 Sonderfall: Partielles Lesen ... 268

16 INTERPRET 269

17 Anhang 271

17.1 EBCDIC Code-Tabelle ... 271

17.2 Wurzel-Routine ... 280

17.3 Literaturverzeichnis/Verwandte Publikationen ... 282

17.4 z/Series- und z/OS-Abkürzungsverzeichnis ... 283

18 Stichwortverzeichnis 303

1 Einleitung

1.1 Ziel des Buchs

Mit diesem Buch über REXX habe ich mir die Aufgabe gestellt, ein lehrbegleitendes Buch für all diejenigen zu schreiben, die sich im Studium, Selbststudium oder im Rahmen eines Seminars in die Programmierwelt der IBM-Großrechner vortasten. Sie können das erworbene Wissen anschließend im Rechenzentrum, der Produktionssteuerung, der klassischen Anwendungsentwicklung, aber auch in der Internetprogrammierung und nicht zuletzt in der Systemprogrammierung einsetzen.

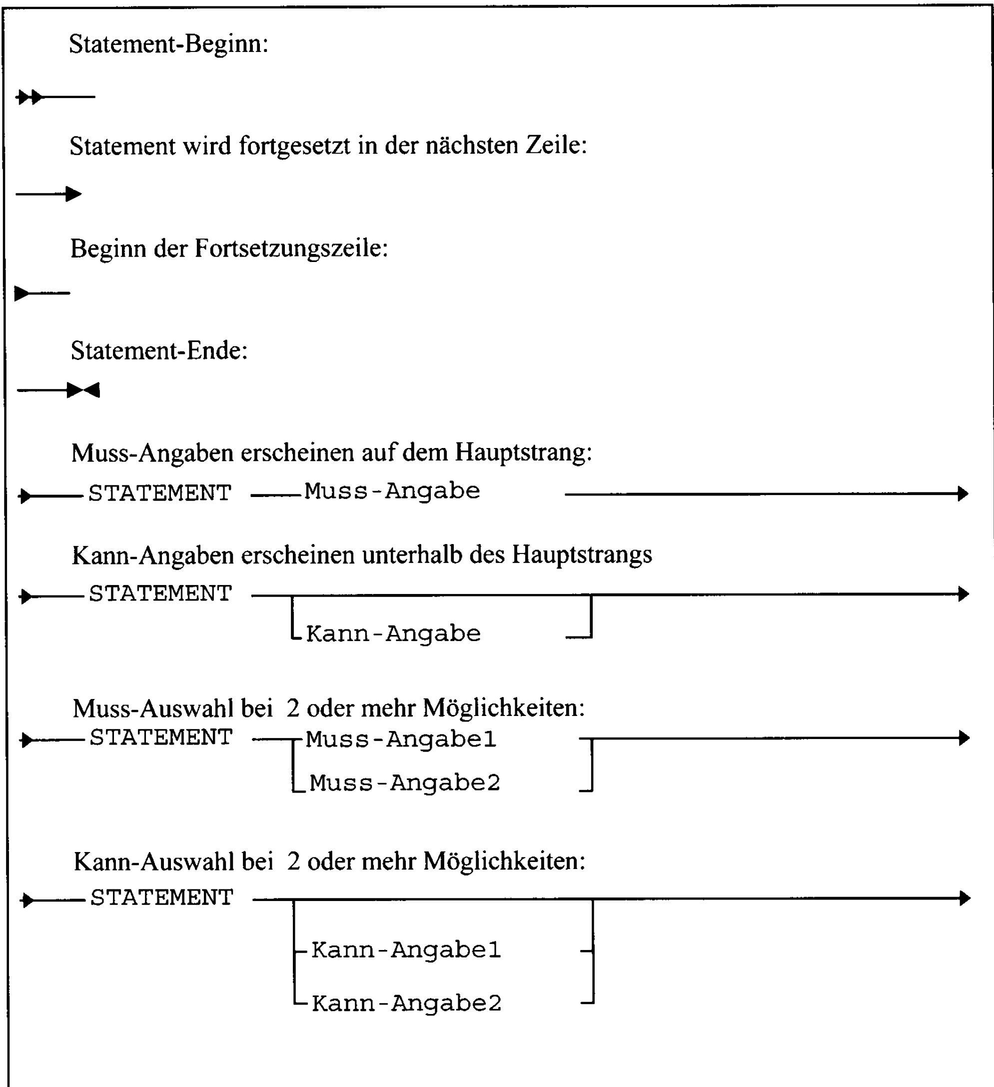
Darüber hinaus ist dieses Buch als arbeitsbegleitendes Werk für Praktiker gedacht. Die vielen Beispiele sollen zum einen die Theorie verständlich machen, zum anderen Anregung für viele neue Anwendungen sein.

1.2 Feedback erwünscht

Wenn Ihnen beim Lesen Fehler oder Ungenauigkeiten auffallen, wenn Sie Verbesserungs- und Erweiterungsvorschläge haben, aber auch wenn Ihnen das Buch einfach gefällt, dann schreiben Sie ihre Bemerkungen einfach an rexx@cross-systems.de.

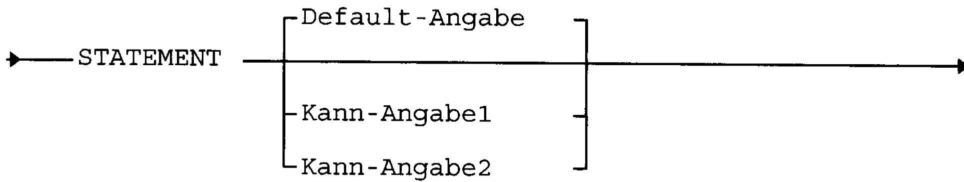
1.3 Verwendete Syntax-Notationen

Erklärung der verwendeten Syntax (1)

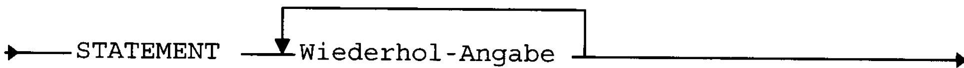


Erklärung der verwendeten Syntax (2)

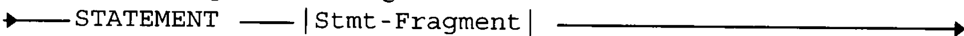
Angabe über dem Hauptstrang ist Default-Wert



Wiederholbare Angabe



Statement-Fragment-Einleitung:



Statement-Fragment-Beschreibung

Schlüsselwörter sind in Großbuchstaben, variable Angaben, die der Programmierer wählen kann, in Kleinbuchstaben geschrieben.
Die Angabe expr erlaubt den Einsatz einer Expression (siehe Definition).
Sonderzeichen (z.B. Klammern und Kommata) müssen als Teil der Syntax angegeben werden:

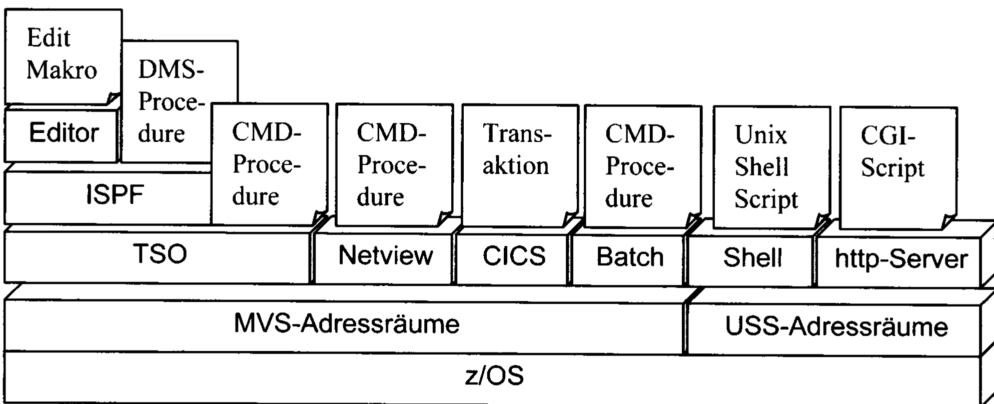
2 Einführung in REXX

2.1 Entstehung und Verbreitung von REXX

Am 20. März 2004 feierte die Computersprache REXX (Restructured Extended Executor Language), die von Mike Cowlishaw in den IBM-Laboratories in England entwickelt wurde, ihr 25-jähriges Jubiläum. Mitte der 80er Jahre wurde REXX von IBM auf dem MVS-Markt als Alternative zur Sprache CLIST (Command List) angeboten. MVS (Multiple Virtual Storage) war damals das mächtigste, kommerziell genutzte Betriebssystem der IBM, gebaut für die Verwaltung der ganz großen Unternehmen und Behörden. Dieses Betriebssystem wurde Ende der 90er Jahre von OS/390 (OS/390 = Open System der Generation 390) abgelöst, welches – janusköpfig – sowohl die bisherige MVS-Architektur fortführte als auch ein UNIX-Derivat mit dem Namen USS (UNIX System Services) besaß. Als die IBM im Jahr 2002 die Architektur wechselte und das 31-Bit-basierte OS/390 mit dem 64-Bit-Betriebssystem z/OS ersetzte, war REXX auf IBMs Big Servern längst etabliert.

2.2 Verwendung

Einsatz von Rexx-Prozeduren unter z/OS



2.2.1 Einsatzgebiete

REXX gehört systematisch zu den Sprachen der 4. Generation, basiert auf der strukturierten Programmierung und wurde zunächst als plattformunabhängige, nicht-kompilierbare Interpreter-Sprache im Rahmen der IBM-SAA (System Application Architecture) entwickelt. Mittlerweile gibt es auch REXX-Compiler. Für die Anhänger objektorientierter Sprachen am PC bietet IBM seit Mitte der 90er Jahre ObjectREXX an.

Der Einsatzschwerpunkt unter dem MVS der 80er Jahre lag von Anfang an auf dem TSO/E (Time Sharing Option/Extended) und dem ISPF (Interactive System Productivity Facility), das als menügesteuerte Full-Screen-Oberfläche mit seinem mächtigen Editor und den vielen Utilities das TSO überhaupt erst vernünftig bedienbar macht. REXX-Routinen dienen hier als Command Procedures der Anpassung des TSO/E und des ISPF an die Firmenbedürfnisse. Gerade die DMS (Dialog Management Services) des ISPF sowie SCLM (Software Configuration and Library Management) basieren auf REXX-Routinen und tauschen mit ihnen ihre Variablen aus. Im ISPF-Editor werden in REXX geschriebene Programme als Edit-Makros eingesetzt. Die alte und wegen ihrer mangelnden String- und Arithmetikelemente leistungsschwächere CLIST, ist heute, obwohl im TSO als zweiter Interpreter aktiv, keine ernstzunehmende Alternative mehr.

REXX dient den IT-Spezialisten nicht ausschließlich unter IKJEFT01 (Programmname des TSO). REXX-Routinen lassen sich im Batch auch unter dem Programm IRXJCL (Name des REXX-Interpreters) ausführen, unterliegen dann allerdings der Einschränkung, keine TSO- und ISPF-Befehle benutzen zu können.

Um den Datenbankzugriff zu DB2 (DB2=Database 2) zu gewährleisten, hat IBM die Schnittstelle DSNREXX eingerichtet. Bis auf ganz wenige Ausnahmen sind damit sämtliche SQL-Statements für DB2 möglich.

Im Bereich der Kommunikationsüberwachung setzen die IT-Spezialisten REXX-Routinen zur Optimierung von NETVIEW ein. Sogar für das CICS (Customer Information Control System) können Transaktionen in REXX geschrieben werden.

REXX wird nicht nur unter z/OS-TSO, sondern auch unter z/OS-USS interpretiert und dient hier als UNIX-Shell-Script-Sprache der Ablaufoptimierung und Erstellung maßgeschneiderter Anwendungen des Rechenzentrums und der IT-Entwicklungsabteilungen. Damit kann jede Firma ihren Großrechner-Script-Standard bei REXX belassen und muss nicht auch noch PERL oder andere UNIX-Script-Sprachen neu hinzunehmen. Setzt man in einer REXX-Routine UNIX-Commands wie CP, einen FTP-Aufruf oder MVS-Commands wie OCOPY ein, dann dienen sie als „Mixed Application" dem Austausch von Informationen über die Betriebssystem-Plattformen hinweg.

Interessant für Programmierer ist es, REXX-Routinen als CGI-Scripts (Common Gateway Interface) im Internet oder Intranet unter IBMs http-Server einzusetzen. Bei einer extrem hohen Entwicklungsgeschwindigkeit und mit direktem Zugriff auf die DB2-Datenbanken des Hosts lassen sich Erfassungs- und Ausgabemasken, Formulare genannt, in HTML unter Einsatz von CSS (Cascading Style Sheet) erstellen. Diese Technik erlaubt es, große Teile des

bisherigen CICS-, TSO- und ISPF- Geschäfts ins Intranet zu verlagern. Jedoch müssen dann von Anfang an die im Großrechnerbereich besonders hohen Sicherheitsansprüche berücksichtigt werden.

2.2.2 Wer programmiert in REXX?

Von der Arbeitsvorbereitung über die Anwendungsentwicklung bis hin zur Systemprogrammierung ist REXX *das* Werkzeug für IT-Personal, um TSO-, UNIX-Shell-Scripts und IT-interne Verwaltungsanwendungen zu schreiben. Aber auch für kleinere und mittlere Großrechneranwendungen ist die leicht erlernbare Sprache ideal – gerade in Verbindung mit den Dialog Management Services des ISPF. Auf Windows-Plattformen kommen alle objektorientierten Programmierer auf ihre Kosten mit ObjectREXX.

Ein riesiges Potential tut sich bei der Internet-/Intranet-Programmierung auf. Wer den ressourcenaufwändigen Informationsfluss über das „3-tiered" Kaskadensystem z/OS-Database-Server – Unix-Application-Server – PC-Client vermeiden möchte, findet mit REXX-CGI-Scripts in Verbindung mit HTML und CSS eine schnelle und kostengünstige Alternative, weil hier die Application-Server-Plattform eingespart wird. Insoweit rentiert es sich für Internet-/Intranet-Entwickler, einen Blick über den Großrechnerzaun zu werfen.

3 Sprachaufbau

Beispiel: Schleife mit Zeilendialog und Upro-Aufruf als Funktion

```
①  /* REXX                                 Beginn Hauptprogramm*/
②  say 'Hallo, lieber' userid() '- ich errechne den',
       '3-prozentigen Provisions-Betrag aus einem von',
       'Dir eingegebenen Gesamtbetrag (incl. Prov.)'
③  do forever                          /*Endlos-Schleife    */
④     say 'Bitte gib Gesamtbetrag oder "ENDE" ein'
⑤     parse upper external z1          /*Eingabe abholen    */
⑥     if z1 = 'ENDE' then leave        /*Verlassen des Loops*/
⑦     if datatype(z1)='NUM'        /*wenn Eingabe numerisch,*/
        then say up1(z1)         /*dann Ausgabe UP1-Ergebnis*/
⑧       else say  'Wert falsch'     /*sonst Fehlermeldung*/
⑨  end                                 /*Schleifenende      */
⑩  exit                                /*Ende d. Haupt-Pgms */

❶  up1: procedure                      /*Unterroutine UP1   */
❷  parse arg ges
❸  prov = ges / 103 * 3               /*Berechnen Provision */
❹  return prov                    /*Rückkehr mit Rückgabewert*/
```

Dieses Programm besteht aus 10 Clauses der Hauptroutine und 4 Clauses der Unterroutine.

3.1 Formatfreiheit

Die Sprache REXX ist formatfrei. Einrückungen, Leerzeichen (**Blanks**) und Leerzeilen sind bis auf seltene Ausnahmen überall erlaubt. Statements und Variablennamen sind case-insensitive, das heißt, sie können großbuchstabig, kleinbuchstabig oder gemischt geschrieben werden.

3.2 Bestandteile einer REXX-Exec

Ein REXX-Programm ist eine Ansammlung von 1–n Clauses (siehe ① bis ⑩ und ❶ bis ❹), die ihrerseits zusammengesetzt sind aus 1–n Blanks, einer Abfolge von Token, 1–n Blanks und einem Semikolon, das in den meisten Fällen weggelassen werden kann, weil es implizit durch das Zeilenende oder den Doppelpunkt eines Labels unterstellt wird.

REXX-Exec
REXX-Exec oder kurz Exec ist die Bezeichnung für ein REXX-Programm. Häufig spricht man auch ganz allgemein von einer REXX-Prozedur. REXX-Unterprogramme unterscheidet man je nach Art des Aufrufs in die Typen Subroutine und Function. Standardisierte Functions des Herstellers nennt man Builtin Functions.

Clause
Jede REXX-Exec besteht aus 1–n Clauses (Clause = Statement). Es gibt 3 Grundtypen von Clauses: Null Clauses (Null Clause = Leerzeile oder ganzzeiliger Comment), Labels (Sprungmarke) und Instructions. Instructions sind wiederum unterteilt in Assignments (Zuweisungen), Keyword Instructions und Commands. Eine Clause kann zusätzlich aus 1–n Comments sowie führenden, eingebetteten und nachfolgenden Blanks bestehen. Erstreckt sich eine Clause über mehr als eine Zeile, ist als Fortsetzungszeichen das Komma zu verwenden (siehe ②). Teilen sich hingegen mehrere Clauses eine Zeile, dient als Trennzeichen der Strichpunkt.

Token
Eine Clause, in seine Einzelteile zerlegt, besteht aus 0 (dann Null-Clause) bis n Token (Zeichen) sowie optionalen Comments und Blanks (Leerzeichen).

Comment
Ein Comment (Kommentar) besteht aus einer beliebigen Zeichenfolge, beginnend mit „/*“ und endend mit „*/“. Ein Kommentar in der 1. Zeile, in dem das Wort REXX vorkommt, wird REXX-Identifier genannt. Dieser „missbrauchte“ Kommentar dient der Sprachidentifikation und verhindert im z/OS die Interpretation des Codings durch den CLIST-Interpreter.

3.2.1 Token-Klassen

Token teilt man ihrer Funktion nach in folgende Klassen ein:

Klasse	**Variante**	**erlaubte Zeichen**	**Beispiel**

String	Literal String	0–n beliebige Charcter-Zeichen, eingekleidet in Hochkommata oder Gänsefüßchen	`''` (Null-String)
			`'Abc 1'`
			`"Abc 1"`

	Hex String	0–9 und A–F und Kombinationen daraus, eingekleidet in ' oder ", gefolgt von einem x/X	`'01'x` `"C1828340F1"X`

	Binary String	Binärziffern (0, 1) in Gruppen von 4 (Nibble) oder 8 (Byte), eingekleidet in ' oder ", gefolgt von einem b/B	`"1"b`
			`'00000001'B`
			`'1100 0001'B`

Symbol	Variable Label	A–Z a–z 0–9 @ # $ ¢ . ! ² ? _	`Pnr = Tab.i.j` `call UPRO#01`

Number	Integer
	Dezimalzahl
	negative Zahl
	Exponentiale
		0–9 und die Vorz. + –
		Dezimalpunkt .
		Blank nach + – erlaubt
		E und die Vorz. + –
			`17`
			`3.4`
			`-135.020`
			`1.2E+3`

Operator	Arithmetik
	Vergleich
	Logik
	Verkettung
		+ – * / % // **
		= < <= > >= <> \= ^=
		& ! &&
		!!
			`a = b + 7`
			`if x<=y ...`
			`if x<y & a=b ...`
			`a = b !! c`

Special Character		, ; :) (	`up: a=(3+4)*2` `return;`

Anmerkung: Einige Sonderzeichen werden im deutschen Zeichensatz anders benutzt. So ist z.B. das Zeichen @ auf Großrechnern mit deutscher EBCDIC-Codepage als § zu schreiben.

Token (Zeichen und Zeichenkombinationen) sind die Low-Level-Elemente einer EXEC, aus denen sich eine Clause bilden lässt. Man kann sagen, dass sich ein REXX-Programm letztlich aus lauter Token zusammensetzt

3.2.2 String

REXX kennt drei Arten von String-Token: Literal-, Hexadecimal- und Binary Strings.

Literal Strings sind konstante Zeichenketten, die in Hochkommata oder Gänsefüßchen gekleidet sind. Sie dienen häufig der Variablenvorbelegung oder als Parameter von Programmaufrufen und Bildschirmausgaben. Ein einzelner Literal String kann bis zu 250 Bytes beinhalten. Beachten Sie aber, dass Verkettungen von Strings untereinander sowie Verkettungen von Strings mit Variablen längere Ergebnisse zulassen, deren Limit nur durch Längenrestriktionen für Variableninhalte und den der EXEC zur Verfügung stehenden Hauptspeicher gegeben ist.

```
knum = '241' /*Füllt Variable KNUM mit '241' (3 Zeichen)*/
knam = "Fred's Kneipe"  /* füllt KNAM        (13 Zeichen)*/
say 'Bitte gib Umsatz fuer' knum knam 'ein'
parse external kums
...
```

Hexadecimal Strings sind Konstanten, die sich nur aus Zeichenkombinationen von 0–9 und A–F zusammensetzen. Sie sind in Hochkommata oder Gänsefüßchen zu kleiden, unmittelbar gefolgt von einem X oder x. Blanks als Füllzeichen zwischen Hex-Pärchen werden vom Interpreter ignoriert. Unpaarigen Kombinationen wird eine führende Null unterstellt. Hex Strings werden heute seltener und meist nur zur Ermittlung von Hauptspeicheradressen eingesetzt. Ein einzelner Hex String kann bis zu 250 Bytes beinhalten.

```
knum = 'F2 F4 F1'X                           /* knum = 241*/
addr = "00 08 F2 E1 0E 23"x /*Zuweisung Hauptsp.-Adresse*/
```

Binary Strings sind Konstanten, die sich nur aus Zeichenkombinationen von 0 und 1 zusammensetzen. Sie sind in Hochkommata oder Gänsefüßchen zu kleiden, unmittelbar gefolgt von einem B oder b. Einen Viererblock solcher Binärziffern nennt man Nibble, einen Achterblock nennt man Byte. Blanks als Füllzeichen zwischen Bytes oder Nibbles werden vom Interpreter ignoriert. Einem Binary String, der nicht aus einer durch 8 teilbaren Anzahl von Binärziffern besteht, werden führende Nullen unterstellt. Binary Strings werden in REXX nur selten und dann eher zu IT-technischen Aufgaben eingesetzt:

```
knum = '11110010 11110100 11110001'B        /* knum = 241*/
antw = "1"b                        /*Zuweisung als Schalter*/
```

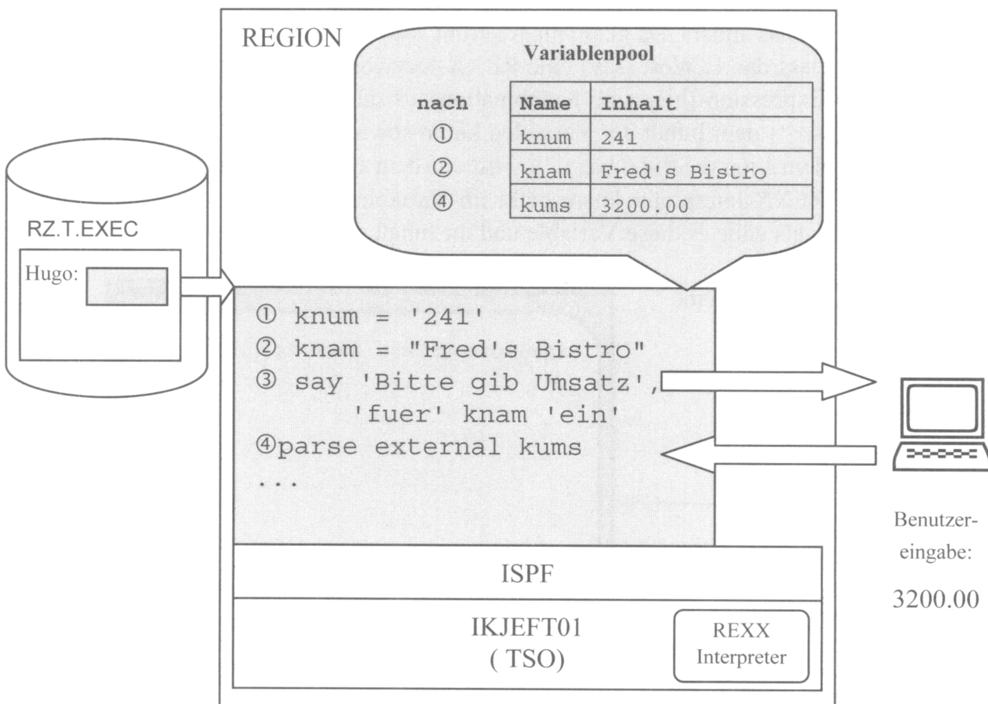
3.2.3 Symbole, Variablen, Zahlen

Die Symbol-Schreibweise regelt die Bildung von Variablen, Simple Symbols, Compound Symbols, Constant Symbols und Labels.

Erlaubte Zeichen sind:

- Die Groß- /Kleinbuchstaben des englischen Alphabets A–Z a–z (keine Umlaute, kein ß).
- Die Ziffern von 0 bis 9.
- Die Sonderzeichen @ # $ ¢ . ! ? _.
  Beim deutschen Zeichensatz gilt @ =X'7C' = §, ¢ = X'4A' = Ä, ! =X'5A' = Ü.
- DBCS-Characters in Verbindung mit (X'41'-X'FE')-ETMODE.

Eine **Variable** ist ein benannter Hauptspeicherbereich des Variablenpools einer EXEC. Ein Symbol wird durch Zuweisung zur Variable. Aber auch durch die PARSE-Instruction oder bestimmte Builtin Functions werden Variablen erzeugt. Der Name einer Variable ist auf ein Maximum von 250 Zeichen begrenzt. Er darf nicht mit einer Ziffer oder einem Punkt beginnen. Wohl aber dienen Punkte innerhalb des Namens zur Bestimmung von Array-Dimensionen. Variablen, deren Namensteile durch „." getrennt sind, nennt man „compound variables". Der Inhalt einer einzelnen Variable, ob normal oder compound, darf die 16 MB-Grenze nicht überschreiten.



TSO-User-Adressraum

Eine Variable, die verwendet wird, ohne zuvor einen Wert zugewiesen bekommen zu haben, nennt man **Simple Symbol**.

```
①knum = '241'

②say 'Bitte gib Umsatz fuer' knum knam 'ein'
...
```

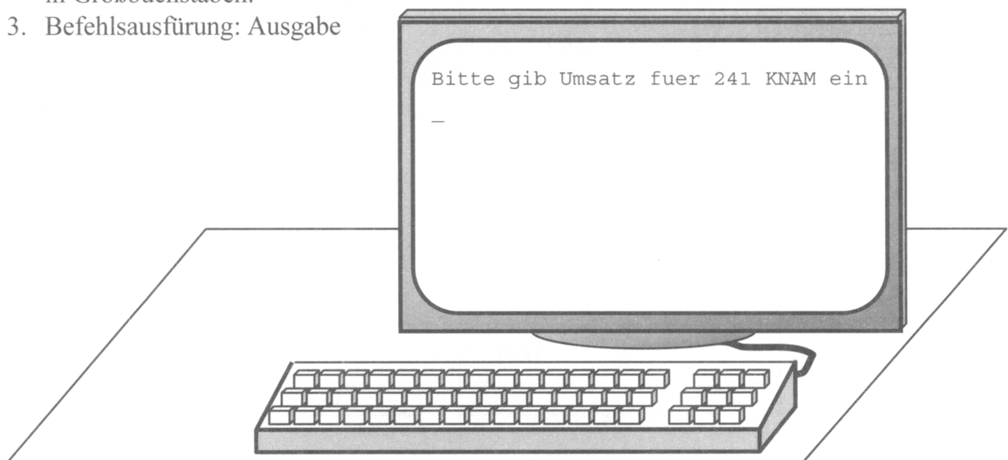
Arbeitsweise des Interpreters:

Zeile ①

1. Auflösung von Variablen (Hier kommet keine Variable vor). knum ist ein Simple Symbol und 241 ein Literal String.)
2. Feststellung, dass das 1. Wort (knum) zwar keine Instruction ist, aber dem Simple Symbol ein Gleichheitszeichen folgt.
3. Befehlsausführung: Zuweisung des Literalwerts 241. Jetzt erfolgt der Eintrag im Variablenpool. Ab jetzt spricht man bei knum von einer Variable.

Zeile ②

1. Auflösung von Variablen: Da knum zuvor gefüllt wurde, handelt es sich um eine Variable. Bei knam ist das anders. Da knam nicht gefüllt wurde, liegt ein Simple Symbol vor.
2. Feststellung, dass das 1. Wort (say) eine REXX-Keyword Instruction ist. Die rechts von say codierte Expression (hier eine Kombination aus dem Literal String `'Bitte gib Umsatz fuer'`, dem Inhalt der Variablen knum sowie dem Simple Symbol knam und zuletzt noch dem Literal String `'ein'`) wird auf dem Bildschirm als Zeile ausgegeben. Obwohl der REXX-Interpreter knam nicht im Variablenpool findet, bricht er nicht ab, sondern tut so, als gäbe es diese Variable und ihr Inhalt sei gleich ihrem Namen, nur eben in Großbuchstaben.
3. Befehlsausfürung: Ausgabe



Eine Compound Variable, die verwendet wird, ohne zuvor einen Wert zugewiesen bekommen zu haben, nennt man **Compound Symbol**.

```
①summe = 0
②do i = 1 to 3
③  say 'Bitte gib Umsatz der Filiale' i 'ein'
④  parse external ums.i
⑤  summe = summe + ums.i
⑥end
⑦do k = 1 to 3
⑧  say 'Filiale' k ums.k
⑨end
⑩say 'Gesamt:   'summe
```

User-Eingabe:

234.00

333.00

541.00

Zeile	Interpretertätigkeit
①	Die Variable „summe“ wird erstmals im Variablenpool erzeugt. Ihr Inhalt ist 0.
②	Nur beim 1. Schleifendurchlauf: Die Variable „i“ wird erstmals im Variablenpool erzeugt. Ihr Inhalt ist 1.

Bei jedem Schleifendurchlauf gilt: Ist der to-Wert überschritten, wird die Schleife verlassen und es geht weiter mit der Clause, die nach dem zugehörigen END ⑤ codiert ist. Ist der to-Wert nicht überschritten, werden die Befehle, die dem DO folgen, durchgeführt.

Zeile	Interpretertätigkeit
③	Prompting: `Bitte gib Umsatz der Filiale i ein,` wobei i im ersten Schleifendurchlauf auf 1 steht, im zweiten auf 2, im dritten auf 3 und im vierten auf 4.
④	Mit dem PARSE des 1. Durchlaufs wird die Compound Variable UMS.1 erzeugt. Der Inhalt ist 234.00. In den folgenden Durchläufen werden UMS.2 mit 333.00 und UMS.3 mit 541.00 gefüllt.
⑤	Die Variable „summe“ erhält pro Durchlauf einen neuen Wert durch Addition ihres bisherigen Inhalts mit demjenigen von UMS.1, UMS.2 usw.
⑥	Beim END wird Variable i um +1 (Defaultwert) erhöht.
⑦ ⑧ ⑨	Die Variableninhalte von UMS.1 bis UMS.3 werden zeilenweise ausgegeben.
⑩	Der Inhalt der Variable „summe“ wird ausgegeben.

Anmerkung: Den Namensteil bis zum ersten Punkt eines Compound Symbols bzw. einer Compound Variable nennt man Stem. Der Stem der Compound Variablen ums.1, ums.2 und ums.3 ist „ums.“.

Unter einem **Constant Symbol** versteht man einen Token, der mit einer Ziffer oder einem Dezimalpunkt beginnt. Constant Symbols werden ohne Einkleidungszeichen codiert. Ihr Wert ist im Gegensatz zu Variablen nicht veränderbar.

Falls sich das Constant Symbol nur aus Ziffern und einem Dezimalpunkt zusammensetzt, ist es gleichbedeutend mit einer als Literal String verwendeten Zahl:

```
knum1 = 3.01
say knum1 + 2.41                        /* Ausgabe: 5.42         */
```

Falls das Constant Symbol nichtnumerische Zeichen beinhaltet, werden diese bei der Interpretation lediglich in Großbuchstaben umgesetzt:

```
say 320csi                              /* Ausgabe: 320CSI       */
```

Anmerkung: Hier wäre aus Gründen der Nachvollziehbarkeit die Verwendung eines Literal Strings (`SAY '320CSI'`) angemessener.

Number

Als Number bezeichnet man Literal Strings oder Constant Symbols, die aus rechenbaren Zeichen bestehen:

```
a = 2000
b = '-1.5E+3'                           /* -1.5 mal 10 hoch 3   */
c = ' + 18.00'
d = +1e-2                               /*    1 mal 10 hoch -2 */

erg = a + b + c + d                     /* 2000-1500+18.00+0.01*/
say erg                                 /* Ausgabe:     518.01  */
```

Anmerkung: In der Exponentialschreibweise sind hinter dem E nur Ganzzahlen erlaubt.

Operator

Operatoren sind Sonderzeichen-Token, die zur Durchführung von Arithmetik, einfachen und exakten Vergleichen, Boole'scher Logik und Verkettungen benötigt werden.

Special Character

Special Characters werden eingesetzt, um das Coding übersichtlicher zu gestalten. Runde Klammern fassen Ausdrücke zusammen, das Blank lockert Clauses auf, das Komma leitet die Fortsetzung einer Clause in der nächsten Zeile ein, das Semikolon trennt Clauses, die in derselben Zeile codiert sind, und der Doppelpunkt kennzeichnet Labels.

3.2.4 Arten von Clauses

Null Clause	Label	Assignment
`/* Kommentar */`	`call upl`	`TEL = 1436850`
`/*Dies`	`...`	`A123 = 37.18`
`ist`	`upl:`	`N.1 = 'Peter U.'`
`ein Kommentar`	`...`	`i = A123 + 0.8`
`*/`	`return`	

Keyword Instruction

ADDRESS	OPTIONS
ARG	PARSE
CALL	PROCEDURE
DO - END	PULL
DROP	PUSH
EXIT	QUEUE
IF - THEN - ELSE	RETURN
INTERPRET	SAY
ITERATE	SELECT - END
LEAVE	SIGNAL
NOP	TRACE
NUMERIC	UPPER

Command

REXX Commands:

```
"DELSTACK"
"DROPBUF"
"EXECIO …"
"EXECUTIL …"
"HE"
"HI"
"HT"
"MAKEBUF"
"QBUF"
"QELEM"
"QSTACK"
"RT"
"SUBCOM …"
"TE"
"TS"
```

TSO Commands
ISPF Commands
Edit Commands
Console Commands
DB2 Commands
UNIX Commands

Null Clause
Darunter ist eine Leerzeile, eine einzelne Kommentarzeile oder ein sich über mehrere Zeilen erstreckender Kommentar zu verstehen. Null Clauses werden vom Interpreter ignoriert. Sie dienen dem Programmierer lediglich zur optischen Gestaltung des Programms und zu Dokumentationszwecken.

Label
Label, auch Marke genannt, ist ein Ort im Programm, zu dem mit den Instruktionen SIGNAL (Bedeutung: „go to") und CALL sowie mit einem Funktionsaufruf gesprungen werden kann.

Assignment
Ein Assignment (Zuweisung) dient dem Erzeugen bzw. Überschreiben einer Variable. Dabei wird dem Symbol links vom Gleichheitszeichen ein Wert zugewiesen, der sich aus einem Ausdruck, der rechts vom Gleichheitszeichen steht, ergibt. Man sagt: Die Zuweisung erfolgt von rechts nach links.

Command
Neben Keyword Instructions und Unterroutinen kann man in einer EXEC auch REXX-Commands und Commands der aktuellen Host-Adressumgebung (Default: TSO) verwenden.

3.2.5 Expression

Eine Expression kann sein ...	Beispiel	Wert
... ein Literal String,	`d1 = "RZ.P.DATA"`	`RZ.P.DATA`
... ein Symbol,	`m1 = pers01`	`PERS01`
... eine Variable,	`v1 = m1`	`PERS01`
... der Rückgabewert einer Funktion,	`UHR = time('N')`	`10:33:59`
... das Ergebnis einer arithmetischen Operation,	`erg = 1 + 4 * 3`	`13`
... das Ergebnis einer Vergleichsoperation,	`x = 10 > 8`	`1`
... das Ergebnis einer logischen (Booleschen) Operation,	`x = 10 > 8 & 1 > 3`	`0`
... oder eine Verkettung verschiedener Expressions.	`d2 = "'"d1"("m1")'"`	`'RZ.P.DATA(PERS01)'`

Ein häufig auftretender Begriff in der REXX-Syntax ist die „Expression". Eine Expression (Ausdruck) umfasst 1–n Token. Expressions findet man häufig als Bestandteile von Keyword Instructions, als Teil eines Assignments rechts vom Gleichheitszeichen, manchmal aber auch als ganze Commands.

Beispiel: Keyword Instruction

```
say 'Heute ist der' date('E')
```

Beispiel: Assignment

```
a = 17
b =  4
c = (a + b)/3                        /* Ergebnis:   7        */
```

Beispiel: TSO-Command

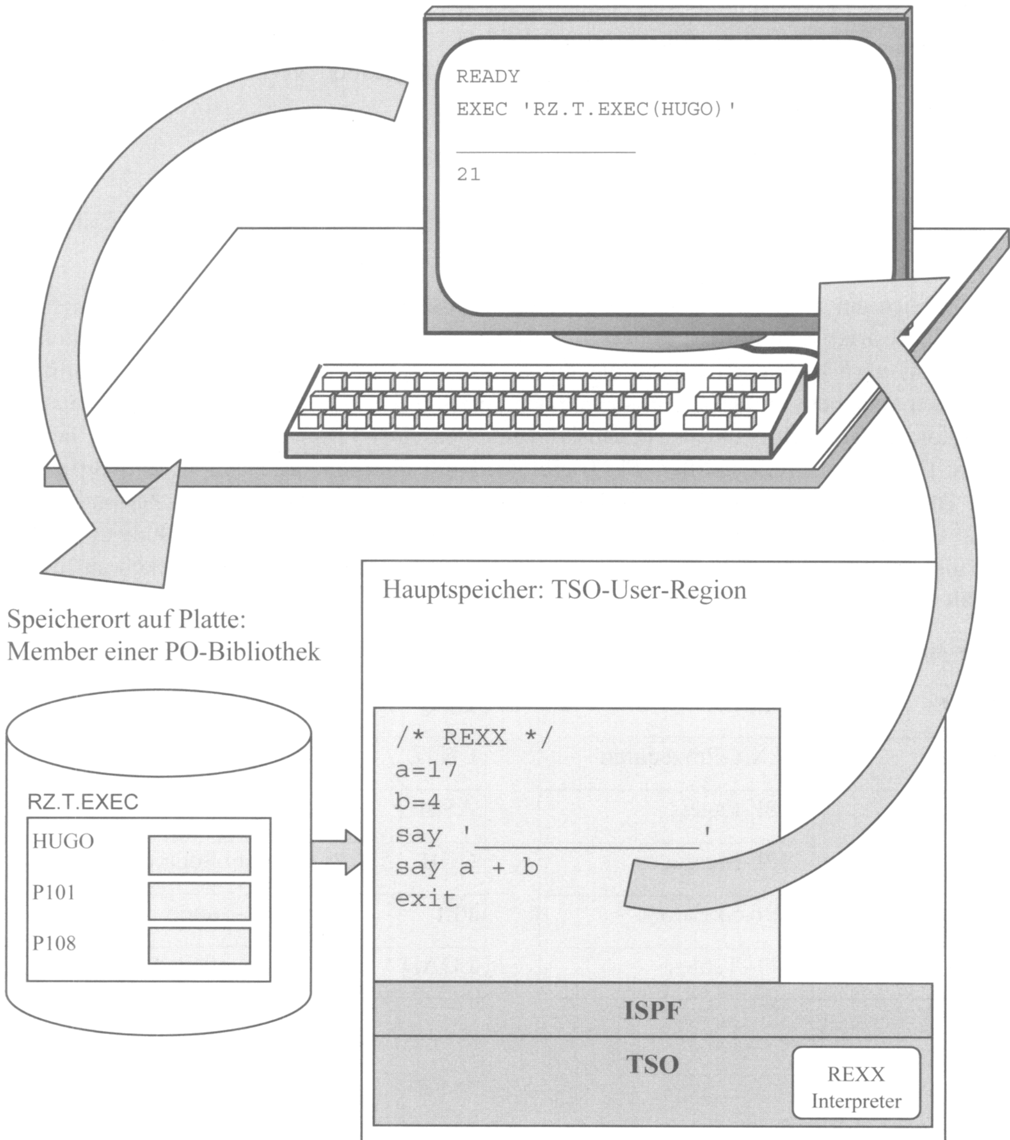
```
cmd = "DELETE"
datei = "'RZ.TEST.DATA'"
cmd datei                        /* DELETE 'RZ.TEST.DATA'*/
```

Expressions können Sie schachteln:

```
UHRZEIT = substr(time('N'),7,2)    /*10:28:59 → 59          */
```

In diesem Beispiel ist die Builtin Function time(...) die Expression, deren Resultat von der Function substr(...) ab Stelle 7 in Länge 2 ermittelt und nach UHRZEIT übertragen wird: In UHRZEIT steht anschließend die Sekundenangabe.

4 Speicherung und Aufruf



4.1 Speicherung eines REXX-Programms

REXX-Programme werden im z/OS als Members in Bibliotheken (PO-Dateien) gespeichert. Die logische Satzlänge solcher Bibliotheken kann bis zu 255 Bytes sein, das Satzformat fix oder variabel geblockt (FB oder VB). In der Praxis werden meist Bibliotheken mit 80-stelligen, geblockten Sätzen verwendet. Die neuere PDSE-Technik ist grundsätzlich der bisherigen PDS-Technik vorzuziehen. PO-PDSE-Bibliotheken sind nicht so störungsanfällig, ihre Directory-Größe ist unlimitiert und sie können dynamischer wachsen, als PO-Bibliotheken. PDSE-Bibliotheken entstehen bei der Allocation durch DSN-Type-Parameter LIBRARY. Bei der Online Allocation mit ISPF 3.2 ist das Feld „Data set name type" mit dem Wert LIBRARY auszufüllen, bei der Batch Allocation verwenden Sie in der Job Control Language den DD-Statement-Parameter

```
//              DSNTYPE=LIBRARY.
```

Der Dateiname (DSN) einer REXX-Bibliothek sollte immer mit dem dem Wort EXEC enden, z.B.: `RZ.T.EXEC` oder `U123456.MUC.PROD.EXEC`.

Für Dateien am z/OS gilt generell: Sie haben nicht einfach irgendeinen Namen. Zum einen sind die Restriktionen des Herstellers zu beachten: Die max. DSN-Länge beträgt 44 Stellen, spätestens nach 8 Zeichen kommt ein Punkt, die so entstehenden Namensteile (Qualifier), beginnen alphabetisch und setzen sich alphanumerisch fort. Zum anderen legen aufbau- und ablauforganisatorisch bedingte Namenskonventionen einer Firma fest, wie ein DSN heißen muss. Der First Level Qualifier (FLQ) gibt Auskunft über den/die Datei-Verantwortlichen. Das ist entweder ein RACF-User oder der Name einer RACF-Gruppe. Der Second Level Qualifier (SLQ) und weitere Qualifier stehen der Namensvergabe einer Abteilung zur freien Verfügung, während der Last Level Qualifier (LLQ) bei Programm-Bibliotheken auf deren Inhalt rückschließen lässt.

Hier ein Auszug von LLQs, die weltweit üblich sind:

LLQ	Inhalt
EXEC	REXX-Prozeduren
PANELS	ISPF-Panels
MSGS	ISPF-Messages
SKELS	ISPF-Skeletons
TABLES	ISPF-Tables

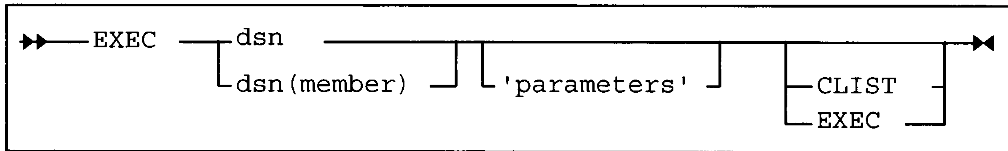
LLQ	Inhalt
CNTL	Job Control
ASM	Assembler Source
COB	Cobol Source
PLI	PL1-Source
LOAD	Gebundene Lademodule

4.2 Aufruf eines REXX-Programms

Es gibt zwei verschiedene Arten des Aufrufs von REXX-Routinen unter TSO/ISPF: Explizit nennt man den Aufruf mit dem TSO-Command „EXEC“, dagegen implizit, wenn einfach der Member-Name als Command dient.

4.2.1 Expliziter Aufruf

Syntax



Beispiel

Variante 1: Unconditional

```
EXEC 'RZ.T.EXEC(HUGO)' EXEC
```

Mit Suffix: EXEC

Region

HUGO

```
a=17
b=4
say '_______________'
say a + b
exit
```

ISPF

TSO

REXX Interpreter | TSO-Cmd Interpreter | CLIST Interpreter

Variante 2: Conditional

```
EXEC 'RZ.T.EXEC(HUGO)'
```

Ohne Suffix (Default: CLIST)

HUGO

```
/*        REXX        */
a=17
b=4
say '_______________'
say a + b
exit
```

ISPF

TSO

REXX Interpreter | TSO-Cmd Interpreter | CLIST Interpreter

Mit dem TSO-Command EXEC lassen sich CLIST- und REXX-Programme sowohl im TSO-Ready-Mode als auch in ISPF 6 (ISPF Command Shell) ausführen. Unter Voranstellung des Wortes „TSO“ ist der Aufruf in jeder Panel Command Line möglich. Explizit nennt man diesen Aufruf deshalb, weil der User das zu interpretierende Objekt, sei es ein einzelner sequentieller Dataset oder ein PO-Member, ausdrücklich benennen muss.

Es empfiehlt sich, das Objekt immer in Hochkommata zu kleiden, z.B.:

EXEC 'RZ.T.EXEC(HUGO)' **EXEC**

Wenn Sie die Hochkommata weglassen, unterstellt TSO den TSO-Prefix als FLQ. Der TSO-Prefix ist bei den meisten Installationen die TSO-UserId des Aufrufers.

Bei einer angenommenen UserId „RZ01“ bewirkt der Aufruf EXEC T(HUGO) EX folgendes Ergebnis: EXEC 'RZ01.T.EXEC(HUGO)' EXEC.

Die Erklärung liegt im Defaultverhalten: Der FLQ RZ01 wird bei allen DSN-Angaben ohne Hochkommata als DSN-Prefix generiert. Der Suffix „EX“ ist die Kurzschreibweise von EXEC. Er ruft nicht nur den REXX-Interpreter, der das Coding des Objekts ausführen soll, sondern substituiert auch den LLQ, wenn das Objekt nicht in Hochkommata gekleidet ist.

Der Aufruf
EXEC T(HUGO) CLIST
wird folgendermaßen interpretiert:

EXEC 'RZ01.T.CLIST(HUGO)' CLIST.

Somit ist der EXEC-Suffix nicht nur ausschlaggebend für die Wahl des Interpreters, sondern auch für die Bildung des Dataset-Namens, in dem das Member HUGO zu suchen ist. Der Defaultwert ist CLIST.
EXEC T(HUGO) erzeugt deshalb: EXEC 'RZ01.T.CLIST(HUGO)' CLIST.

Fehlt also die Interpreterangabe, wird immer zuerst der CLIST-Interpreter bemüht. Unterstellen wir, dass es die Bibliothek „RZ01.TEST.CLIST“ gibt und HUGO in der CLIST-Sprache codiert ist, so wird die Routine ausgeführt. Ist jedoch HUGO ein REXX-Programm, bringt der scheiternde CLIST-Interpreter folgende Fehlermeldung:

```
IKJ56479I COMMAND .....NOT FOUND OR REXX IDENTIFIER MISSING
IKJ56479I SUPPLY '/* REXX */ AS THE FIRST RECORD
```

Um diesen Fehler zu vermeiden, codieren Sie in Zeile 1 ihres Members: /* **REXX** */
Daraufhin tritt der CLIST-Interpreter zurück und übergibt die Kontrolle an den REXX-Interpreter.

Das Gleiche geschieht, wenn Sie unter ISPF im Directory einer REXX-Bibliothek ein „EX“ vor einem Member-Namen eingeben. Dieser „quick-and-dirty“-Aufruf erfordert im gerufenen Member zwingend einen REXX-Identifier.