Karl-Heinz Wittemann

Inside z/OS

Das große TSO-REXXikon

Bestellung
http://www.REXXikon.de

Geschrieben für alle, denen EDV Freude bereitet und die nicht alles tierisch ernst und verbissen sehen. Keine wissenschaftliche Abhandlung, sondern ein Buch aus der Praxis für die Praxis.

Alle in diesem Buch enthaltenen Programme und Verfahren wurden nach bestem Wissen und Gewissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund ist das im vorliegenden Buch enthaltene Programm-Material mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Der Autor übernimmt infolgedessen keine Verantwortung und wird keine daraus folgende Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials oder Teilen davon entsteht.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Musterschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Satz: Karl-Heinz Wittemann, Bad Herrenalb
Umschlaggestaltung: Claudia Wittemann, Bad Herrenalb
Druck und Bindung: WOGE Druck, 76307 Karlsbad, URL: www.wogedruck.de

Vorwort

Ich bin's mal wieder. Nach 13 Jahren Pause möchte ich die Welt erneut mit einem Werk beglücken. Ab und an erreichen mich Anfragen über das Internet wegen meines REXX-Buches für TSO von 1994. Der HOST lebt! Das vorliegende Exemplar basiert auf z/OS 1.10 und enthält die bis dato vorhandenen Neuerungen.

Hier soll keine wissenschaftliche Abhandlung geliefert werden. Ein Buch vom Praktiker für Praktiker. REXX, Edit Makros, LM-Services, Dialog Management Services, Job Control Language – Alles, außer Tiernahrung (und nicht nur bis Samstag).

Totgesagte leben länger. Eine Erkenntnis, welche noch nie so richtig war, wie in dieser Zeit. Der Mainframe ist teuer, unflexibel und passt nicht mehr in unsere Zeit. Ähnliches ist immer wieder zu hören. Egal woher, egal warum – es ist so falsch wie eh und je.

Es ist immer wieder interessant, wie man sich alles schönrechnen kann, wenn man es nur intensiv genug will. Interessant sind nicht die Kosten eines EDV-Systems, welche durch die Hardware und die Lizenzgebühren anfallen, sondern die Gesamtaufwände, um es zu betreiben. Hierzu zählen auch die Stromkosten, die Stellflächen, die administrativen Kosten für Personal und nicht zu vergessen: die Sicherheit, soweit sich diese in Euro und Cent ausdrücken lässt. Vor einigen Jahren hatte ich einen mehrmonatigen Einsatz im Rechenzentrum eines großen EDV-Dienstleisters, in dem auch diverse Unix-Systeme (Hardware eines namhaften Herstellers) installiert waren. Auch in dieser Umgebung spricht man von variabler Laststeuerung und dynamischen Workload. Wer intensive Bekanntschaft mit dem Mainframe gemacht hat, erkennt bald, dass es sich dabei auf der Middleware wohl eher um Worthülsen handelt.

Noch ein Ärgernis: Oft wird geklagt, dass es keinen Nachwuchs für Mainframe-Systeme gibt. Welch Wunder. Ich erinnere mich an Zeiten, in denen große Betriebe ihre Mitarbeiter permanent geschult haben und Azubis in den Unternehmen während ihrer Ausbildungszeit nicht nur bunte Männchen erzeugen durften. Wenn man im Zuge von Sparmaßnahmen die Ausbildung vernachlässigt, ist das Fehlen von Fachleuten keine wirkliche Überraschung. Oder doch?

Nach nunmehr 34 Jahren Erfahrung mit dem Großrechner und einigen glücklich gescheiterten Versuchen, an Unix oder Windows Freude zu finden, erfreue ich mich noch immer an meinem geliebten Mainframe und bin mittlerweile der festen Überzeugung, dass ich als erster von uns beiden in Rente gehen werde. Egal, wie lange es bei mir noch dauern wird.

Danksagung an…

…alle Seminarteilnehmer, welche in den vergangenen 30 Jahren unter mir leiden mussten. Ich habe von euch allen viel gelernt. Das Eine oder Andere findet sich in Musterbeispielen wieder.

…die Google-NewsGroups TSOREXX, ISPF-L und IBM-MAIN. Eine Art OpenSource-Gemeinde. Gut finde ich auch, dass ihr nicht jedem die Arbeit abnehmt, nur weil er zu faul ist, im Manual nachzulesen.

…www.cbttape.org, der MVS-Freeware-Seite. Oft habe ich mich gefragt, wie ich meine Zeit besser nutze: Kodieren oder die Lösung dort suchen.

…die EMA – European Mainframe Academy. Es hat lange gedauert, bis sich jemand gefunden hat, der Mainframe-Ausbildung auf die Bedürfnisse der jungen Leute abstimmt. Lernen via Internet, moderne Techniken, virtuelle Klassenzimmer, animierte und vertonte Präsentationen und vieles mehr. Geboten werden Abschlüsse als Systemspezialist oder Anwendungsentwickler unter z/OS. Also: es geht doch! Danke an Volker Falch, Wolfram Greis und Prof. Dr. Ing. Wilhelm Spruth (in alphabetic order). Herr Professor Spruth auch dafür, dass er den Mainframe an der Uni nicht sterben lässt.

…Christoph Ranalder, der das Gesamtwerk Probe gelesen hat. Christoph, Du warst die Idealvorstellung meiner Zielgruppe: Jung, vielseitig interessiert, kein 100%iger Mäuschenschubser aber auch kein alleswissender Dinosaurier. Vielen Dank für Deine Unterstützung und die konstruktive Kritik.

…an die moderne Technik. Das vorliegende Exemplar wurde in Word2007 erstellt. Dafür habe ich den PC schätzen gelernt. Zudem haben viele Druckereibetriebe Probleme, DCF-Dateien im EBCDI-Code zu verarbeiten.

Die Zielgruppe

Ich möchte alle erreichen. Dennoch: Eine gewisse Grundlage sollte vorhanden sein. Das Buch richtet sich an z/OS-Nutzer, welche mit REXX einfache oder komplexe Anwendungen unter TSO entwickeln wollen, und bereits Erfahrung in einer Programmiersprache gesammelt haben. Es handelt sich um kein Lehrbuch, sondern vielmehr um die praktische Hilfe am Arbeitsplatz – ein Nachschlagewerk mit einer Vielzahl von Beispielen für die tägliche Arbeit. Es lebt von der Praxis, nicht von der Theorie.

Die Beispiele sollen Hilfe bei der Entwicklung geben. REXX-native oder Dialoge, Helfer im Alltag durch Edit Makros. JCL-Beispiele für Batchverarbeitung, wo sie sinnvoll und vernünftiger als eine Foreground-Verarbeitung ist.

Willkommen sind auch alle, welche die Effizienz ihrer Leistung steigern möchten oder die Outputs ihrer Anwendung per Programm überprüfen wollen – oft reichen hierzu ein paar Lines of Code und REXX ist oft schneller kodiert, als Cobol compiled.

Für diejenigen, welche schon alles wissen, schlägt mein Herz natürlich auch. Allerdings habe ich eine Bitte: Der Host lebt! Wann habt ihr euch das letzte Mal mit den Neuerungen auseinandergesetzt? Ich möchte nicht den Eindruck erwecken, alles zu wissen. Es hat mich aber oft erstaunt, wie häufig Neuerungen an Anwendern vorbeigehen und wie leicht man auf einem bestimmten Know How stehen bleibt. Es geht ja auch immer irgendwie. Wenn ein Problem nicht sofort lösbar ist, umfliegt man es. Für euch versuche ich, alle Fragen zu beantworten, die ich mir oft selbst gestellt habe. Und: Es gab in z/OS 1.8 viele Neuerungen, auf die man sehr lange gewartet hat.

Versprechen

Wenn Sie für das Eine oder Andere mal Unterstützung brauchen und in Ihrer unmittelbaren Umgebung keine finden, schreiben Sie mir einfach: rexxikon@spgmbh.de. Ich hätte gerne Ihren Code, eine kurze Beschreibung der Problemstellung und was ihrer Meinung nach falsch läuft. Sie bekommen sicher Antwort. Versprochen.

Die Struktur des Buches

Generell sollen hier unterschiedliche Anforderungen erfüllt werden. In den verschiedenen Kapiteln soll für jeden schnell auffindbar das angeboten werden, was gerade gebraucht wird.

Kapitel 1 widmet sich der Erklärung der allgemeinen Dinge. Die Sprache REXX wird erläutert, die Umgebung erklärt. Es werden diverse Beispiele aus der täglichen Praxis aufgeführt.

Kapitel 2 und 3 setzen sich mit Edit Makros und LM-Services auseinander. In der täglichen Praxis sind REXX-native Anwendungen eher selten. Ich frage mich oft, wie jemand vernünftig im Editor arbeiten kann, ohne die vielen kleinen wiederkehrenden Dinge des Lebens über ein Makro abzudecken. Schnell mal eine Datei sortieren, den Cursor auf einen Dateinamen zu stellen, um in diese reinzuschauen oder einen Auto-Vervollständiger zu benutzen. Im Internet sind wir dies gewohnt und empfinden es als praktisch. Warum nicht auch im z/OS?

Kapitel 4 beschreibt ISPF (Dialog Manager). Neben der Erklärung der Services wird das Wirken des Dialog Managers anhand einer kleinen Anwendung demonstriert, welche zeigt, wie die einzelnen Komponenten zusammenspielen. Darin sind auch die letzten größeren Erweiterungen aus z/OS 1.8 enthalten (REXX-Coding in Panels und Skeletons etc.).

Kapitel 5 liefert einen Überblick über Job Control. Sinniger Weise befassen wir uns hier nur mit den gängigen Parametern. Beschrieben werden die alltäglichen Operanden der JOB-, EXEC- und DD-Anweisung. Darüber hinaus die nun schon seit längerem verfügbaren aber zu selten benutzten Special-Statements wie IF-THEN-ELSE, private PROCLIB-Zuweisungen, INCLUDES, SETs und dergleichen mehr.

Die weiteren Kapitel liefern eine umfassende Beschreibung der Syntax für alle Teilbereiche. Die Syntaxbeschreibung liegt nach REXX/TSO (Kapitel 6), Edit Makros (Kapitel 7) LM-Services (Kapitel 8), Dialog Manager (Kapitel 9) und Job Control Language (Kapitel 10) getrennt vor. Da oftmals gar nicht so einfach zu erkennen ist, ob es sich bei einer bestimmten Komponente um eine REXX-Instruktion, eine Built-in-Function, eine Extended-TSO-Function oder ein TSO-Kommando handelt, wurde die Beschreibung all dieser Teilbereiche im Kapitel 6 vereint und in alphabetischer Folge beschrieben. Auf diese Weise soll die Suche erleichtert werden. Jeder beschriebene Operand wird über ein Syntaxdiagramm abgebildet. In der darüber liegenden Zeile wird seine Herkunft beschrieben (TSO-Kommando, REXX-Instruktion, etc.).

Literaturhinweis

Ich möchte an dieser Stelle keine Hinweise auf Bestellnummern von Manuals der IBM geben. Häufig ändern sich die Titel und Buchnummern. Generell möchte ich auf den Server der IBM verweisen, auf dem Manuals zum Download zur Verfügung stehen. Viele Manuals stehen als Reference und Users Guide zur Verfügung. In der Reference wird im Allgemeinen die Syntax beschrieben. Im Users Guide stehen wertvolle Hinweise für die Anwendung.

Verweise auf das Internet

Die gängigsten Google-Groups. Hier treffen sich die absoluten Größen der Fachwelt zu den jeweiligen Themen. Sehr Empfehlenswert:

Group	**Thema**
bit.listserv.ibm-main	Generelle Mainframe-Themen
bit.listserv.tsorexx	Alles zu REXX unter TSO/E (und mehr)
bit.listserv.ispf-l	Alles zu ISPF
bit.listserv.db2-l	Alles zu DB2

Interessante Homepages (alphabetisch):

URL	**Inhalte / Betreiber**
www.cbttape.org	MVS-Freeware
www.cbttape.org/links.phtml	Links von der Freeware-Seite
www.mzelden.com/mvsutil.html	Mark Zelden
www.ibm.com/systems/z/os/zos/bkserv/	BookServer der IBM
www.mvsforums.com	Manuals und Links
www.planetmvs.com	David Alcock
www.rexxla.org	Rexx Language Association
www.spgmbh.de	REXX-Beispiele deutsch
www.theamericanprogrammer.com	Viele Links zu Literatur
www.tsotimes.com	TSO und ISPF

Inhaltsverzeichnis

Kapitel 1 – REXX native

Historisches

REXX hatte seinen Ursprung im VM/CMS, einem Mainframe-Betriebssystem der IBM. Als Interpreter-Sprache fand es dort in sehr kurzer Zeit viele Freunde. Entwickelt wurde REXX von Mike Colishaw (auch bekannt als Michael REXX). Die Sprache zeichnet sich durch einfache Strukturen und eine Vielzahl von eingebetteten Funktionen aus, ist leicht erlernbar. Viele Produkte aus dem Bereich Mainframe offerieren ein API zu REXX.

Ende der 80er Jahre des letzten Jahrhunderts fand REXX den Weg ins MVS und hat dort die Interpretersprache CLIST verdrängt. CLISTen sind zwar auch heute noch in der TSO- und NetView-Umgebung anzutreffen, stammen aber meist aus der grauen Vorzeit. Neuentwicklungen werden in der Regel nicht mehr in CLIST geschrieben.

Auf einen Nenner gebracht: REXX kann alles, was CLIST kann. REXX kann auch alles, was CLIST nicht kann.

IBM lieferte sein PC-Betriebssystem OS/2 erstmals mit REXX anstelle von Basic aus. Auch zu DOS war REXX erhältlich. Unter UNIX/Linux wurden REXX-Derivate angeboten und heute noch kann mit Regina-REXX ein Interpreter für Linux und Windows kostenlos aus dem WEB geladen werden. REXX ist damit eine der wirklich wenigen Sprachen, die vollkommen plattformunabhängig ist.

Drei kleine Wermutstropfen

IBM hat den SAA-Standard (Konzept zur Standardisierung der von IBM entwickelten Software) gebildet. Befehle, welche dem SAA-Standard entsprechen, sind plattformunabhängig. Leider entsprechen nicht alle Funktionen in TSO/E-REXX dem SAA-Standard.

Die zweite gewöhnungsbedürftige Situation ist die Dateiverarbeitung. Hier stehen in REXX unter TSO die Befehle ALLOC und EXECIO zur Verfügung. Auf anderen Plattformen wird an Stelle dieser Befehle die Dateiverarbeitung mit den Funktionen STREAM(), LINEIN(), LINEOUT() und LINES() durchgeführt. Generell gäbe es diese auch unter TSO, sie werden im Allgemeinen aber nicht mitinstalliert.

Wer PIPEs aus NetView kennt, wird diese im TSO vergebens suchen. Es gibt dort die sogenannten BatchPipes. Diese sind aber nicht direkt vergleichbar und zudem ein kostenpflichtiges Zusatzprodukt, welches die meisten Unternehmen nicht gesondert anmieten. Generell können die PIPE-Funktionen käuflich erworben werden, allerdings nicht über die IBM. Die meisten Unternehmen lehnen dies ab, da die Pipes unter TSO exotisch sind und keinen Standard darstellen. Verständlich.

Synonyme

Aktivitäten werden auf dem Mainframe über Synonyme durchgeführt. Anders als auf dem PC, wäre es hier unvorstellbar, die Namen von Dateien hart in ein Programm zu kodieren. Dieses müsste jedesmal geändert werden, wenn sich der Dateiname ändert.

Diese Systematik wird auch im Systemumfeld benutzt. Ladeaktivitäten für Programme, Panels, Tabellen etc. werden über Synonyme. Wir sprechen in diesem Zusammenhang auch von den sog. DD-Namen – eine Art Alias. Welche physischen Dateien sich dahinter verbergen, ist für eine Verarbeitung ohne Bedeutung.

Oft stehen mehrere Dateien mit einem DD-Namen in Verbindung. In diesem Fall wird in der Reihenfolge der Verkettung in den verschiedenen Dateien gesucht.

Für eine funktionierende Umgebung, in welcher mit REXX und ISPF-Services gearbeitet werden soll, ist es unerlässlich, ein sauberes Umfeld aufzubauen. Dies geht in aller Regel bereits während der Logon-Phase vonstatten. Leider kann in Zuge des Logons nicht jede Anforderung für jede beliebige Applikation erfüllt werden. Oftmals muss die Applikation das von ihr benötigte Umfeld selbst aufbauen. Aus diesem Grund sollen hier zunächst eine Beschreibung der für eine Dialoganwendung benötigten DD-Namen und deren Funktionalität erfolgen. Diese Dateien haben PO-Format (partitioned). Um Ärger zu vermeiden sollten die Dateien das Format Library (PDS/E) haben.

DDNAME	Funktion
SYSEXEC	Aufruf von REXX-Programmen
SYSPROC	Aufruf von CLISTen
ISPPLIB	Aufruf von ISPF-Panels (Bildschirmmasken)
ISPMLIB	Aufruf von Nachrichten in ISPF-Dialogen (Messages)
ISPPROF	Der Variablenpool von ISPF
ISPTLIB	Zum Lesen von ISPF-Tabellen
ISPTABL	Zum Schreiben von ISPF-Tabellen. Sollte eindeutig sein
ISPSLIB	Zum Lesen von ISPF-Skeletons
ISPFILE	Zum Schreiben von FileTailoring Ergebnissen
ISPLLIB	Für den Aufruf von LOAD-Moduln in ISPF

SYSEXEC und SYSPROC

Der DD-Name SYSEXEC wurde mit der Einführung von REXX unterstützt. Bis zu diesem Zeitpunkt war CLIST die Interpretersprache im TSO. Um Alt und Neu besser voneinander trennen zu können, wurde von IBM empfohlen, CLISTs unter dem bisherigen DD-Namen SYSPROC weiterhin zu rufen. REXX dagegen sollte über den DD-Namen SYSEXEC aufgerufen werden.

Generell können natürlich auch REXXe aus SYSPROC sowie CLISTen gerufen werden. Aber an dieser Stelle eine persönliche Frage: Würden Sie Ihre Sportschuhe nach dem Joggen ins Gemüsefach neben Frischobst legen, nur weil es geht?

ISPPLIB

Ist in ISPF-Dialogen zuständig für das Laden von Bildschirmmasken. Es ist unerheblich, über welchen Service die Masken geladen werden sollen. Es kann sich dabei um Datenpanels, Tabellenpanels oder Auswahlpanels handeln.

ISPMLIB

Wird für den Aufruf von Nachrichten in einem ISPF-Dialog benutzt. Die Membernamen, in denen die Nachrichten gesucht werden, ergeben sich aus dem Alpha-Prefix der Nachricht und den ersten beiden numerischen Werten dahinter. Eine Nachricht AKUV001A beispielsweise wird innerhalb der ISPMLIB-Verkettung in einem Member AKUV00 gesucht. Mehr dazu bei der Beschreibung der Nachrichtenbehandlung.

ISPTLIB und ISPTABL

Diese DD-Namen sind für Tabellenverarbeitung vorgesehen.

ISPTLIB wird zum Lesen von Tabellen benutzt. Dahinter können sich beliebig viele Dateien verbergen. Die Tabellen werden unter gleichem Namen als Members in ISPTLIB gesucht.

ISPTABL wird für Schreiboperationen von Tabellen benutzt. Diese Zuweisung sollte eindeutig sein. Es macht nicht viel Sinn, für eine Schreiboperation mehrere Ausgaben zu definieren und dem TSO zu überlassen, irgendeine von diesen zu benutzen.

ISPSLIB und ISPFILE

Diese DD-Namen sind für File Tailoring (Skeleton-Verarbeitung) vorgesehen.

ISPSLIB wird zum Lesen von Skeletons benutzt. Dateiverkettung ist möglich. Die Skeletons werden als namensgleiche Member in ISPSLIB gesucht.

ISPFILE wird für Schreiboperationen im File Tailoring benutzt, wenn die Ergebnisse nicht temporär erstellt werden sollen. Diese Zuweisung sollte immer eindeutig sein.

ISPPROF

Hier wird der Applikationsspezifische ISPF-Profile-Variablenpool abgebildet. Der Membername bildet sich aus der Applikationskennung als Prefix und dem Suffix PROF.

Der Variablenpool für ISPF-Standard ist ISPPROF.

ISPILIB

Die Image-Library. Von dieser haben viele ISPF-Programmierer noch nie gehört. Ich selbst bin auch erst vor einiger Zeit darüber gestolpert. Unter ISPILIB können Images

im GIF-Format hinterlegt werden die in graphischen Bereichen eines Panels abbildbar sind.

Besonderheiten:

Die DD-Namen ISPTLIB/ISPTABL und ISPSLIB/ISPFILE können umgangen werden, indem die entsprechenden Befehle mit dem Operanden LIBRARY() kodiert werden und über davon abweichende DD-Namen Schreib-/Lesevorgänge durchführen. Dies verkompliziert aber meist eine Anwendung unnötig.

Dateinamen

Dateinamen sind Schall und Rauch, aber es gibt gewisse Empfehlungen, an die man sich halten sollte. Nicht, dass es nicht auch anders ginge, was ist schon schlimm daran, sprechende Namen zu benutzen. Am Ende würde es ja jeder verstehen…

Bitte informieren Sie sich über die Regeln für Dateinamen in Ihrem Unternehmen. Heute wird überall DFSMShsm eingesetzt. Unter Umständen verlieren Sie Ihre Dateien schneller, als Sie diese erstellt haben.

DSNAME	Inhalt
Userid().?.EXEC	Aufruf von REXX-Programmen
Userid().?.CLIST	Aufruf von CLISTen
Userid().?.PANELS	ISPF-Panel-Bibliothek
Userid().?.MSGS	ISPF-Message-Bibliothek
Userid().?.TABLES	ISPF-Tabellen-Bibliothek
Userid().?.SKELS	ISPF-Skeletons

Programmaufrufe

Explizite Programmaufrufe

Für den expliziten Aufruf von REXXen steht das TSO-Kommando EXEC zur Verfügung. Damit kann eine REXX aus einer beliebigen Datei aufgerufen werden, einerlei, ob diese unter SYSEXEC/SYSPROC im System definiert ist, oder nicht. Einzige Bedingung: der Aufrufer muss die erforderlichen RACF-Rechte besitzen.

Bei der Ausführung des Befehles werden die TSO-Richtlinien bezüglich Dateinamen berücksichtigt. Vollqualifizierte Dateien werden zwischen Apostrophe gesetzt. Teilqualifizierte Dateien werden ohne Apostrophe und ohne FLQ (First Level Qualifier) kodiert (siehe Auswirkungen des TSO-Befehles PROFILE).

```
EXEC 'USERXY.MVS.EXEC(KALENDER)' EXEC
```

Auf diese Weise kann eine REXX aus jeder beliebigen Bibliothek gestartet werden.

```
EXEC MVS(KALENDER) EXEC
```

Ergänzt den Namen der Bibliothek nach folgenden Standards: Als FirstQualifier wird die Prefix-Angabe des TSO-Profils benutzt. Der 2. Qualifier wird übernommen. Als dritter Qualifier wird EXEC eingesetzt, da es sich aufgrund des Parameters EXEC um eine REXX handelt und EXEC der Standard-Low Level Qualifier LLQ) ist.

Implizierte Aufrufe

Sie sind möglich, sofern die Bibliothek, aus welcher die REXX gerufen werden soll, unter SYSEXEC zugewiesen ist (wir wollen uns für den Rest des Buches davon trennen, dass die Aufrufe auch über SYSPROC möglich sind, es sei denn, Sie legen Ihre Joggingschuhe… wir lassen das lieber). Dementsprechend kann der Aufruf des Programms unter TSO durch Angabe des Programmnamens erfolgen:

```
KALENDER<enter>
```

Dabei kommt es zu folgender Situation: Der TSO Befehlsprozessor kann in dieser Form Aktivitäten aus drei verschiedenen Bereichen erzeugen und hält dabei eine bestimmte Suchreihenfolge ein:

1. KALENDER wird im Bereich der TSO-Kommandos gesucht. Wird ein gleichnamiges gefunden, wird es ausgeführt.

2. Gibt es kein gleichnamiges TSO-Kommando, werden die SYS1.LINKLIB (Standard-Lademodulbibliothek im MVS) und alle mit ihr verketteten Dateien nach einem gleichnamigen Modul durchsucht. Dies ist in der Regel ein großer Aufwand, da die Suche über eine Vielzahl von Bibliotheken erfolgt, in denen jeweils mehrere Tausend Member vorhanden sind. Gibt es eine namentliche Übereinstimmung, wird intern der TSO-Befehl CALL ausgeführt und das Lademodul wird aufgerufen.

3. Ist die Suche innerhalb der LINKLIB-Verkettung erfolglos, wird die Verkettung von SYSPROC/SYSEXEC durchsucht. Erfolgt die Zuweisung mit Verstand (via EXECUTIL) wird die SYSEXEC-Verkettung vor SYSPROC durchsucht.

4. Ist die Suche auch hier erfolglos, wird eine Fehlermeldung ausgegeben.

An dieser Stelle kann jeder ein wenig zu mehr Leistung zum Nulltarif beitragen. Ist bekannt, dass eine REXX aufgerufen werden soll, kann dies mit einem führenden %-Zeichen kenntlich gemacht werden. In diesem Fall wird nicht nach TSO-Befehlen oder Programmen gesucht. Das hat auch den Vorteil, dass REXXe namensgleich zu TSO-Kommandos benannt werden können. Erfolgt der Aufruf ohne %-Zeichen, wird der TSO-Befehl ausgeführt. Erfolgt er mit, wird die REXX aufgerufen.

```
%KALENDER<enter>
```

Weiterer Vorteil beim Aufruf mit % ist, dass bei Programmstart intern ein Clear Screen Kommando abgesetzt wird.

Merkmale von REXX

REXX zeichnet sich durch einige nicht selbstverständliche Merkmale aus:

Leichte Anwendung

REXX-Programme sind leicht zu lesen, die Sprache ist schnell erlernbar. Die meisten Befehle sind simple Worte aus dem Englischen, keine Abkürzungen, keine Hieroglyphen (SAY, PULL, IF-THEN-ELSE, DO, END, EXIT).

Freies Format

Bei der Kodierung gibt es kaum formale Zwänge. Die Anweisungen können in Groß- oder Kleinbuchstaben oder gemixt geschrieben werden. Zeilen können nach Belieben eingerückt kodiert werden. Weder für den Beginn einer Instruktion, noch für die Spalte einer Fortsetzungszeile gibt es vorgeschriebene Spaltenpositionen. Auch innerhalb einer Zeile können Spaltenbereiche durch Kommentardefinitionen übergangen werden. Generell könnten sogar mehrere Befehle in einer Zeile kodiert werden. Von dieser Möglichkeit ist in der Praxis wegen der schlechten Lesbarkeit abzuraten (außer für drei Leerzeilen: SAY;SAY;SAY).

Eine Instruktion kann, sofern sie in einer Zeile keinen Platz findet, über mehrere Zeilen fortgesetzt werden. Als Ankündigung der Fortsetzung genügt es, eine Zeile mit Komma(,) enden zu lassen. Die Fortsetzung kann in der nächsten Zeile an beliebiger Position beginnen.

Ob REXX-Instruktionen in Sätzen fester oder variabler Länge hinterlegt sind, spielt für den Interpreter keine Rolle.

Variablen

Variablen müssen nicht vordefiniert werden und können während des Programmverlaufes ihre inhaltliche Länge, und ihren Datentyp (character oder numeric) ändern. Genau genommen gibt es in REXX gar keine numerischen Felder, sondern nur Strings. Ein Abbruch wegen unbekannter Variablen in REXX ist nicht denkbar, es sei denn, dies ist über eine entsprechende NOVALUE-Condition so gewollt. Eine nicht belegte Variable in einem REXX-Programm zeigt inhaltlich immer auf ihren zur Großschreibung konvertierten Eigennamen.

Testhilfen

Betrachtet man REXX, drängt sich immer wieder der direkte Vergleich zu CLIST auf. In Bezug auf Testhilfen bietet REXX hier weit mehr und komfortablere Verfahren, Fehler im Ablauf zu erkennen. Böse Zungen behaupten, viele Leute schreiben ihre REXX-Programme absichtlich falsch, weil es mit den Möglichkeiten des Ablaufverfolgers Freude bereitet, diese Fehler zu lokalisieren und zu beheben.

Eingebettete Funktionen

REXX bietet eine Vielzahl von Funktionen. Gerade in den Anfängen gewinnt man den Eindruck, für jedes Problem die Lösung durch die Funktionsvielfalt bereits fertig präsentiert zu bekommen. Tritt nach der ersten Euphorie die Phase der Ernüchterung ein, stellt man fest, dass die eine oder andere Funktion mehr durchaus wünschenswert gewesen wäre (typisch Anwender: gibt es sehr wenig Komfort, gibt man sich damit zufrieden. Ist eine Sprache sehr komfortabel, verlangt man das Unmögliche von ihr). Kein Grund zur Frustration. REXX erlaubt auf eine sehr simple Weise, den Standard-Funktionsumfang durch eigen geschriebene Funktionen nach Belieben zu erweitern (UDF's - User Defined Functions). Beachtet man dabei einige Regeln, kann so ein Eigenprodukt wahlfrei als Unterprogramm, Funktion oder als Kommando benutzt werden. Mehr dazu im Abschnitt »Modularisierung«.

Arithmetik

REXX erlaubt alle gängigen Formen arithmetischer Operationen. Dabei werden Divisionen mit Dezimalergebnissen ebenso unterstützt wie integer Ergebnisse, Potenzrechnungen und Exponentialschreibweisen (siehe unter Arithmetische Operatoren).

Zeichenketten-Manipulationen

Unter den vielfältigen Funktionen werden viele komfortable Möglichkeiten zur Zeichenmanipulation angeboten. Hier hat man sogar häufig den Eindruck, dass des Guten zu viel getan wurde. Oftmals können, um zum gleichen Ergebnis zu kommen, unterschiedliche Funktionen eingesetzt werden, die sich oft nur in der Reihenfolge der Parameterangaben unterscheiden. Der Grund hierfür ist vermutlich nicht darin zu suchen, dass die geistige Flexibilität der Anwender unter Beweis gestellt werden sollte, sondern liegt wohl eher an einem Mangel an Kommunikation zwischen den Entwicklern bei der Programmierung des Interpreters.

Befehlsausführung unter einem Interpreter

Dies hat Vor- und Nachteile. Einerseits ist das Schreiben von Programmen, das Austesten und Ändern sehr simpel. Die einzelnen Befehle werden zum Laufzeitpunkt vom Interpreter in Maschinencode umgewandelt. So entfällt der bei Sprachen wie Assembler, Cobol, PL/I oder Fortran nötige Übersetzungs- und Bindelauf (Compile und Link). Die Programme können getestet werden, sobald der erste Befehl kodiert ist.

Andererseits bringt die Ausführung unter einem Interpreter Nachteile in punkto Performance mit sich. Nicht zu übersehen ist auch, dass für die Ausführung immer lesbarer Quellcode verfügbar sein muss, was unter den TSO-Hackern oft ungeahnte kriminelle Energien freisetzt. Beides lässt sich in der Praxis befriedigend lösen, da von IBM auch für REXX ein Compiler angeboten wird. Kompilierte REXX-Programme sind schneller im Ablauf (wenn natürlich nicht so schnell, wie übersetzte Cobol- oder Assembleranwendungen). Für viele Programme spielt dies eine untergeordnete Rolle. Im

Zweifelsfall kann eine REXX-Anwendung auch in einem Batch-Adressraum zur Ausführung kommen, sofern einige Dinge beachtet werden (siehe REXX im Batch). Viel wichtiger ist dagegen, dass REXX-Programme nicht im Quellcode zur Verfügung gestellt werden müssen, und so für den Anwender nicht mehr les- oder änderbar sind.

Komponenten in einem REXX-Programm

In einem REXX-Programm können unterschiedliche Sprachkomponenten vorkommen:

- REXX-Instruktionen
- Wertzuweisungen
- eingebettete Funktionen
- benutzerdefinierte Funktionen
- Sprungziele (Label oder Marken)
- STACK-Funktionen, wie die Aufnahme von Daten aus I/O-Operationen
- Befehle und Funktionen aus dem Umfeld (System Environment) von REXX.

Dieses Umfeld ist im allgemeinen TSO. Generell kann aber auch Netview, ISPF oder DB2 ankodiert werden.

Literale

Literale sind Zeichenketten, die von Apostrophen(') oder Anführungszeichen(") eingegrenzt werden. Grundsätzlich wäre eine der beiden Möglichkeiten ausreichend, da die Zeichen durch doppeltes Tippen zum Textbestandteil innerhalb des Literals werden. Die Optik sieht aber auch für den geübten Leser eigenartig aus. Die Anforderungen, Apostrophe(') innerhalb der Zeichenketten als Textbestandteil zu benutzen, treten oft auf, da TSO-Kommandos innerhalb eines REXX-Programms im Regelfall als Literale hinterlegt sind, damit sie nicht vom REXX-Interpreter aufgelöst werden. TSO-Kommandos enthalten Apostrophe('), wenn innerhalb der Befehle vollqualifizierte Dateinamen anzugeben sind. Werden diese Namen über Variable geliefert, wird das Chaos der Schreibweise perfekt und nur durch die richtige Anwendung der Literal-Begrenzer gemildert. Wird ein Literal mit einem Begrenzungszeichen begonnen, wird alles, was folgt, als Teil des Literals interpretiert, bis das zur Einleitung der Zeichenkette benutzte Begrenzungszeichen ein zweites Mal kodiert wird.

```
"Zeichenkette"
'Zeichenkette aus mehreren Token gebildet'
"Das wäre ebenso falsch'
'wie das"
"Heinzi's kleine Lotterie und"
'.....sagte:"BLABLA" sind dagegen OK.'
```

Im folgenden Beispiel soll eine Dateizuweisung durchgeführt werden, bei welcher der Name der Datei durch eine Variable geliefert wird. Entscheiden Sie selbst, welcher der beiden nachfolgenden Varianten für die Schreibweise eines TSO-Kommandos Sie den Vorzug geben würden. Richtig sind beide:

```
"ALLOC DD(TEMP) DA('"datei"') SHR REUSE"
'ALLOC DD(TEMP) DA(''datei'') SHR REUSE'
```

In REXX können Literale auch als HEX-Strings gekennzeichnet werden. In diesem Fall dürfen nur die Buchstaben A bis F, sowie die Ziffern 0 bis 9 im Literal vorkommen. Ob die Buchstaben groß oder klein geschrieben sind, spielt dabei keine Rolle. In der Zeichenkette enthaltene BLANKs, sofern sie an der Bytegrenze stehen, verbessern die Lesbarkeit, werden vom Befehlsprozessor aber ignoriert. Zur HEX-Zeichenfolge wird das Ganze durch ein unmittelbar hinter dem Literal-Begrenzungszeichen stehendes großes oder kleines X. Beispiele für HEX-Zeichenfolgen sind:

```
'c1b4'x  oder  'C0F'x  oder  " C 03 FF"X
```

Worte (Token)

WORTE oder Token im Sinne von REXX sind Zeichenketten beliebiger Länge, wobei es natürlich theoretische Grenzen gibt, die in der Praxis eine untergeordnete Rolle spielen. Begrenzt werden Worte durch Komma(,), BLANK( ) oder durch sich selbst. Inhaltlich können Worte aus Zahlen, Buchstaben oder Sonderzeichen (auch gemischt) zusammengesetzt sein. Alle folgenden Beispiele bestehen aus jeweils fünf Worten:

```
"Der Pferd hat vier Bein"
summe = 5 + 7
"A entspricht dem EBCDI-Code 'C1'x"
```

Variablen

Variable sind Platzhalter. Sie stehen für Werte, die erst zur Laufzeit gebildet werden. Eine Variable wird als solche erkannt, weil sie außerhalb von Literal-Begrenzungszeichen (' oder") steht.

Die Namen von Variablen sind praktisch unbegrenzt (250 Byte). Die inhaltliche Länge ist nicht begrenzt, sofern sie im virtuellen Adressraum der TSO-Sitzung Platz finden.

Zur Namensvergabe dürfen Buchstaben (nicht Case-sensitiv), Ziffern und die Sonderzeichen Underscore(_), Hash(#), At(@), Punkt(.) und Questionmark(?) genutzt werden. Die erste Stelle des Namens darf keine Zahl und kein Punkt sein. Wie groß auch immer der Anreiz ist, Variablennamen unter REXX sehr lang mit Einbeziehung der Sonderzeichen zu schreiben, -oft ergibt sich dadurch eine deutlich verbesserte Lesbarkeit-, so ekelhaft kann dies werden, wenn man sich im Nachhinein entschließt, ISPF-Services zu implementieren. Dort sind die Namen von Variablen auf acht Byte begrenzt. Außerdem dürfen sie nur aus Buchstaben, Zahlen und den nationalen Zei-

chen gebildet werden, wobei das erste Byte ein Buchstabe sein muss. Berücksichtigt man diese Restriktionen bereits von Anfang an, sind Programmerweiterungen zu einem späteren Zeitpunkt unproblematisch. Gibt es bei der Namensvergabe von Variablen zwischen REXX und ISPF Unverträglichkeiten, kann die entsprechende Komponente mit REXX nicht unmittelbar über die Variablen kommunizieren.

Wie immer Sie sich bei der Vergabe von Variablennamen entscheiden, benutzen Sie sprechende Namen. Aus dem Namen der Variablen sollte auch für einen Außenstehenden abzuleiten sein, wozu sie genutzt werden soll.

Empfehlung für Variablennamen

Da es in Unternehmen oft keine starren Regeln für die Benennung von Variablen gibt, oder sich die Mitarbeiter nicht daran halten, hier einige Angebote zur Namensvergabe:

Sxxxxxxx für Variable, die eine Schalterfunktion haben (der Inhalt variiert nur zwischen 0 und 1, oder J und N)

Cxxxxxxx für Variable, die eine Zählfunktion übernehmen (C wie COUNT; natürlich könnte auch Z wie ZÄHLER genutzt werden, entscheiden Sie sich aber konsequent für eine Lösung).

Xxxxxxxx für temporäre Variable, die nur kurzfristig als Träger eines Zwischenergebnisses genutzt werden und zu einem späteren Zeitpunkt im Programm bedenkenlos überschrieben werden können.

Vxxxxxxx für symbolische Programmvariable innerhalb REXX. Je nach Bedarf und Einbindung von ISPF könnte man den Ursprung eines Wertes mit

Pxxxxxxx einem ISPF-Panel oder durch

Txxxxxxx einer Dialog Manager Tabelle zuweisen.

Die restlichen sieben Byte sollten den Inhalt der Variablen beschreiben.

Spezielle Variablen

Der Interpreter füllt situationsabhängig drei spezielle Variablen mit Werten, die im Programm abgefragt werden können. Deren Namen und Bedeutung sind:

Variable	Bedeutung
RC	Returncode der letzten ausgeführten Instruktion
RESULT	Aus einem Unterprogramm durch RETURN rückgereichter Wert
SIGL	Quellencode-Zeilennummer, die aufgrund einer SIGNAL-Bedingung verlassen wurde (siehe SIGNAL, "Test und Fehlerbehandlung). Keine dieser Variablen hat einen Grundwert.

Wertzuweisungen

Variablen können in REXX auf unterschiedliche Weisen gefüllt werden. Die häufigste Art ist die direkte Zuweisung. Wertzuweisungen für Variable können aber auch das Ergebnis eines PARSE-Befehles, einer Dateiverarbeitung (Lesen) oder eines Funktionsaufrufs sein. Sehen wir uns einige Formen der Variablenzuweisung an:

Zuweisung	Variableninhalt
`A = 5`	`5`
`B = 3.7`	`3,7 (Dezimal)`
`C = 3E6`	`3000000 (= 3 mal 10^6`
`D = A + 7`	`12`
`E = D + 3 * 2`	`18`
`F = (D + 3) * 2`	`30`
`G = name`	`NAME`
`H = "Name"`	`Name`
`I = "Wun"`	`Wun`
`J = "der"`	`der`
`K = 'bar'`	`bar`
`L = I"der"K`	`Wunderbar`
`M = I J K`	`Wun der bar`
`N = IJK`	`IJK`

An dieser Stelle treffen wir auf ein: Werden die Variablen I,J und K durch BLANK getrennt aneinander gereiht, werden sie als jeweils eigene Variable erkannt. Das BLANK zwischen den Variablen wird übernommen. Wird auf die trennenden BLANKs verzichtet, kann REXX nicht mehr drei unabhängige Variablen erkennen, sondern unterstellt, das IJK eine neue, nicht gefüllte Variable ist, die somit auf ihren zu Großbuchstaben konvertierten Eigennamen zeigt. Um die Inhalte zweier Variablen unmittelbar zu verbinden, wird als Verkettungszeichen zweifaches Ausrufezeichen(!!) oder NULLSTRING kodiert. Woraus folgt:

Zuweisung	Variableninhalt
`O = I !! J!!K`	`Wunderbar`
`P = I""J""K`	`Wunderbar`
`Q = I''J''K`	`Wunderbar`
`R = 2 b7"x`	`02B7(hex)`

Rücksetzen von Variablen

Soll eine Variable auf ihren Urzustand zurückgesetzt werden, ist dies nicht durch `var=""` möglich. In diesem Fall wäre der Inhalt der Variablen NULLSTRING, Ausgangssituation war aber: Die Variable zeigt auf ihren groß geschriebenen Eigennamen. Diese Situation kann über den Befehl `DROP var` herbeigeführt werden.

Compound-Variablen (Stems)

Compound-Variablen bilden eine Besonderheit in REXX. Ihre Namen sind mehrgliedrig. Die einzelnen Teile werden durch Punkt(.) getrennt. Aus wie vielen Namensteilen eine Compound-Variable besteht, spielt keine Rolle. Die einzige Einschränkung besteht in der Gesamtlänge des Namens, der 250 Byte nicht überschreiten darf. Über die Compound-Variablen können, zumindest logisch, Speichertabellen nachvollzogen werden. Am besten sehen wir uns das nachfolgende Beispiel einer Tabellendemo an:

Tabelle TEXT				
1	2	3	4	5
Hallo	Servus	Bye, bye	Salü	Ciao

```
/* REXX * TABDEMO
*/
TEXT.1="Hallo"
TEXT.2="Servus"
TEXT.3="Bye, bye"
TEXT.4="Salü"
TEXT.5="Ciao"
Pos = random(1,5)
say TEXT.pos
exit
```

Soll ein Stem als Gesamtes gefüllt oder zurückgesetzt werden, kann dies über ihren gemeinsamen Namensteil geschehen. Um alle denkbaren Variablen eines Stammes PLZ mit einer 0 vorzubelegen, genügt die Zuweisung **`PLZ.=0`**. Jede denkbare Compound-Variable, sofern sie mit PLZ. beginnt (PLZ.A.B.TRALLALA.IRGENDWAS, etc.) enthält den Wert 0. Ebenso lassen sich sämtliche Variablen eines gemeinsamen Stammes durch **`DROP PLZ.`** wieder auf Grundstellung bringen.

Operanden

Operanden werden in drei Gruppen eingeteilt:

- Arithmetische Operanden
- Vergleichsoperanden
- Boolesche Operanden

Arithmetische Operanden

Sie werden zur Durchführung von Rechenbefehlen genutzt. In größeren arithmetischen Ausdrücken erfolgt die Auflösung des Ausdruckes gemäß der Gewichtung der Operanden. Die nachfolgende Aufstellung zeigt die möglichen arithmetischen Operatoren und ihre Bedeutung. Die Reihenfolge von oben nach unten entspricht der Gewichtung. In dieser Reihenfolge erfolgt die Auflösung des Ausdruckes. Soll davon

abgewichen werden, können um Teile des Ausdruckes runde Klammern gesetzt werden. Klammern werden von innen nach außen gelöst.

Operator	Bedeutung	Ausdruck	Ergebnis
**	Potenzierung	2**4	16
//	Divisionsrest	29 // 4	1
*	Multiplikation	3 * 7	21
%	Division integer (ganzzahlig)	23 % 5	4
/	Division (Dezimalbruch)	10 / 4	2.5
+	Addition	4 + 7	11
-	Subtraktion	13 – 5	8

Der Ausdruck 49-2**5*3/2 liefert 1 weil 49-(((2**5)*3)/2)

Um hier auch für das Leben lernen zu können, möchte ich Ihnen einen Ausspruch eines Bekannten nicht vorenthalten, der meinem Sohn immer sagte: "Die wichtigen Rechenarten sind PLUS und die MULTIPLIKATION, sie mehren den Inhalt des Geldbeutels. Auf MINUS und die DIVISION kannst Du ruhig verzichten, denn nehmen lassen wir uns nichts und teilen tun wir auch nicht gerne".

Vergleichsoperanden

Sie werden benutzt, um zwei Ausdrücke miteinander zu vergleichen. Anders als in vielen anderen Sprachen können in REXX keine Buchstabenkürzel als Vergleichsoperanden genutzt werden (EQ für =). Dies ist gerade für CLIST-Umsteiger sehr lästig, aber man gewöhnt sich ja an alles. Die möglichen Vergleichsoperanden und ihre Bedeutung können aus der nachfolgenden Tabelle entnommen werden:

Aussage	Operator
Gleich	=
Ungleich	^=
Identisch	==
Nicht identisch	^==
Kleiner als	<
Strikt kleiner	<<
Kleiner oder gleich	<=
Nicht kleiner	^<
Strikt nicht kleiner	^<<
Größer	>
Strikt größer	>>
Größer oder gleich	>=
Strikt größer oder gleich	>>=
Nicht größer	^=
Strikt nicht größer	^>>

Offenbar trifft auch in REXX zu: Alle sind gleich, manche sind gleicher! Der Unterschied zwischen GLEICH und IDENTISCH liegt bei Characterstrings in führenden BLANKs, bei numerischen Werten in führenden Nullen. Bei Prüfung auf GLEICH werden sie ignoriert. Bei IDENTISCH muss eine vollständige Übereinstimmung der Zeichen erkannt werden. Gleiches gilt auch für Vergleiche wie STRIKT KLEINER.

Vergleichsoperatoren werden in Befehlen wie **IF**, **WHEN**, **DO WHILE** und **DO UNTIL** zur Prüfung einer Bedingung oder zur Steuerung einer Schleife eingesetzt. Da die Vergleiche nach dem EBCDI-Code durchgeführt werden, können Äpfel mit Birnen verglichen werden (Character und numerische Werte). Dabei ist aber zu beachten:

```
ANTON      <     BERTA
berta      <     Anton
hugo       <     Hugo
Oskar      <     1
Robert     <     Roberta
```

Oder generell:

```
Sonderzeichen < LowerCase < UpperCasse < Zahlen
```

Boolesche Operanden

Sie bilden die dritte Gruppe und werden benutzt, um mehrere Ausdrücke miteinander zu verknüpfen. Mögliche Angaben sind:

Operator	Bedeutung
&	**UND-Verknüpfung**
!	**ODER-Verknüpfung**
&&	**EXCLUSIVE-ODER-Verknüpfung**
^ als Prefix	**Negation**

Durch diese Operatoren können die Ergebnisse von Vergleichen (TRUE(1) oder FALSE(0)) miteinander verbunden werden, und man erhält wiederum ein RICHTIG(1) oder FALSCH(0). Aus der nachfolgenden Tabelle kann die Funktionsweise der Verknüpfungen abgelesen werden:

	UND (&)		ODER(!)		EX-ODER(&&)		Negation
	Richtig	Falsch	Richtig	Falsch	Richtig	Falsch	
Richtig	1	0	1	1	0	1	0
Falsch	0	0	1	0	1	0	1

Anders ausgedrückt: Eine UND-Verknüpfung ist wahr, wenn beide Bedingungen zutreffend sind. Eine ODER-Verknüpfung wird bejaht, wenn zumindest eine Bedingung erfüllt ist. Ein TRUE für die EXKLUSIV-ODER-Verknüpfung erhält man, wenn ausschließlich eine Bedingung erfüllt ist.

Wendet man die Booleschen Regeln konsequent im täglichen Leben an, wäre interessant zu erfahren, wie die deutsche Rechtssprechung hierüber denkt, denn Schilder wie Betteln und Hausieren verboten, oder Zelten und Feuermachen verboten, würden grundsätzlich das Betteln erlauben, sofern man nicht gleichzeitig hausiert, oder es wäre nichts gegen das Zelten einzuwenden, sofern man dabei kein Feuer macht (und umgekehrt). Im Sinne des Herrn Boole müssten diese Vorschriften lauten: Betteln oder Hausieren, sowie: Zelten oder Feuermachen verboten. Würden die Bedingungen dagegen mit einem exklusiven ODER verbunden werden, wäre sowohl das Betteln und Hausieren, als auch das Zelten und Feuermachen erlaubt, solange man immer beides in Verbindung und nicht getrennt voneinander durchführt.

Kommentare

Kommentare werden vom Befehlsprozessor ignoriert und müssen immer zwischen /* und */ eingeschlossen werden. Es spielt dabei keine Rolle, ob als Kommentar eine Zeile oder ein Teil davon definiert wird, oder ob sich der Kommentar über mehrere Zeilen erstreckt.

Befehlsbegrenzer

Das Zeichen Semikolon(;) wird in REXX als Befehlsbegrenzer eingesetzt. Wenn wir davon ausgehen, dass in einer Zeile nicht mehr als ein Befehl steht, kann bei der Kodierung auf das Zeichen verzichtet werden. Wird es kodiert, ist es nicht störend. Werden dagegen mehrere Befehle in einer Zeile kodiert, müssen sie zwingend mit Semikolon voneinander getrennt werden. Die Lesbarkeit eines Programmes verschlechtert sich durch solche Unarten ganz radikal.

Instruktionen

Instruktionen sind Anweisungen an den Interpreter, die ein bestimmtes Ergebnis erzeugen. Im Endeffekt sind Instruktionen kleine Programme, die ablaufen. Das Ergebnis einer Instruktion kann im Programm frühestens in der nächsten Zeile ausgewertet werden. Formal werden Instruktionen in Form von Schlüsselworten geschrieben. Abhängig von der Instruktion und dem gewünschten Ergebnis, können Parameter an die Instruktion angefügt werden.

```
SAY "Hier steht beliebiger Text"
```

Funktionen

Funktionen sind letztlich nichts wesentlich anderes als Instruktionen. Auch bei einem Funktionsaufruf wird im Endeffekt ein Programm ausgeführt. Funktionen haben aber eine grundsätzlich andere Schreibweise und können immer nur Teil einer Instruktion sein. Eine Funktion außerhalb einer Instruktion wird vom Interpreter mit einer Fehlermeldung und der Ausgabe eines Returncodes(-3) quittiert (Ein Returncode -3 wird gemeldet, wenn der Interpreter eine Instruktion oder Funktion nicht ausführen kann, sie an die aktive Systemumgebung weitergibt, und die Aktion auch dort unbekannt ist).

An den Namen der aufzurufenden Funktion werden immer eine öffnende und schließende Klammer angehängt. Abhängig von der Funktion und den speziellen Bedürfnissen, können innerhalb der Klammer eine oder mehrere Angaben erfolgen. Durch die Angaben in der Klammer kommuniziert der Aufrufer der Funktion mit der Funktion (Datenübergabe).

Ein weiterer Unterschied besteht in der Ablaufsteuerung des Interpreters bei Funktionsausführungen. Wird nach ausgeführter Instruktion die nächste Zeile des Quellencodes verarbeitet, sieht es bei Funktionen anders aus. Liest der Interpreter in einer Instruktionszeile eine Funktion, ruft er sie auf und bringt das Ergebnis der Funktion (den von der Funktion zurückgelieferten Wert) innerhalb der Instruktionszeile an die Stelle der Funktion. Im weiteren Sinn könnten wir dieses Vorgehen mit der Substitution von Variablen vergleichen.

```
SAY DATE()       liefert tt mmm jjjj   (24 MAR 2009)
SAY DATE('E')    liefert tt/mm/jj      (24/03/04)
```

Diese Unterschiede kann man sich zunutze machen, wenn man selbst Funktionen oder Unterprogramme schreibt. Berücksichtigt man bei der Kodierung einige grundsätzliche Dinge, kann einzig durch die Art des Aufrufes entschieden werden, ob eine Funktion oder ein Unterprogramm gestartet werden soll. Mehr dazu im Abschnitt Modularisierung.

Systemumgebungen (Environments)

REXX kann unter verschiedenen Umgebungen laufen. Abhängig vom jeweiligen Umfeld können unter REXX die Dienstleistungsangebote der Umgebung genutzt werden. Generell wären möglich MVS, TSO, ISPEXEC, ISREDIT, LINK, ATTACH und NETVIEW.

Abhängig davon, in welchem Adressraum ein REXX-Programm läuft, kann es bestimmte Umgebungen erreichen oder nicht, wobei es grundsätzlich immer eine Standardumgebung gibt. Soll ein Befehl aus einem REXX-Programm an ein erreichbares Umfeld zur Ausführung übergeben werden, das nicht das Standardumfeld ist, muss der Befehl an das gewünschte Umfeld adressiert werden. Die folgende Tabelle zeigt die möglichen Adressräume, in denen REXX-Programme laufen können, die erste

genannte Systemumgebung ist die standardgemäße, alle weiteren Umgebungen können erreicht werden.

Adressraum	Umgebungen
MVS/Batch	MVS, LINK, ATTACH
TSO/E native	TSO, MVS, LINK, ATTACH
TSO/E ISPF	TSO, MVS, ATTACH, ISPEXEC, ISREDIT
NETVIEW	NETVIEW/VTAM, MVS

Befehlsübergabe an die Systemumgebung

Wir wollen hier nicht diskutieren, unter welchen Umständen Instruktionen, die nicht vom REXX-Interpreter ausgeführt werden, sondern an das Umfeld von REXX zur Ausführung übergeben werden, mit oder ohne Eingrenzung zwischen Apostrophe(') oder Anführungszeichen(") funktionieren oder nicht. Wir gehen hier immer davon aus, dass Befehle, die nicht vom Interpreter ausgeführt werden, REXX-seitig als Literale zu hinterlegen sind. Dabei ist zu beachten, dass das Literal unterbrochen und wieder fortgesetzt werden muss, wenn innerhalb des Befehles REXX-Variable eingesetzt werden sollen. Da in der Standardumgebung von REXX ein Apostroph(') häufig als Teil eines Kommandos genutzt wird (immer wenn TSO weiß, das es sich bei einem Operanden um einen Dateinamen handelt, und diese Datei vollqualifiziert angegeben wird, muss das Voranstellen der Benutzerkennung des Anwenders verhindert werden. Der Name der Datei ist in diesem Fall zwischen Apostrophe(') einzugrenzen), gehen wir im weiteren Verlauf davon aus, dass TSO-Kommandos als Literale, eingegrenzt zwischen Anführungszeichen("), zu schreiben sind. Ich bitte an dieser Stelle alle Freaks und Minimalisten untertänigst um Verzeihung (man würde sicher das eine oder andere Byte bei der Kodierung einsparen können). Wird ein Befehl vom Interpreter an das Umfeld zur Ausführung übergeben, wird auch die Ablaufsteuerung für den weiteren Programmverlauf auf das Umfeld übertragen. Nach erfolgter Ausführung des Befehles meldet die entsprechende Umgebung (TSO) den Returncode des ausgeführten Befehles an REXX. Der Interpreter hinterlegt diesen Returncode in der Variablen RC und übernimmt wieder die weitere Steuerung des Programmablaufes.

```
SAY "Datei vollqualifiziert (ohne Hochkomma)"
PARSE EXTERNAL DATEI
"DEL '"datei"'"
IF rc <= 8 THEN SAY "alles klar"
ELSE SAY "Dateizugriff nicht möglich"
```

Läuft REXX in einem TSO-Adressraum, ist das Standard-Environment TSO. In diesem Fall können ohne Adressierung TSO-Kommandos ausgeführt werden. Auch die TSO/E-REXX-Kommandos (Befehle, für deren Ausführung letztlich TSO bemüht wird, die aber nur aus REXX ausführbar sind) wie: NEWSTACK, QSTACK, DELSTACK, MAKEBUF, QBUF, QELEM, DROPBUF, EXECIO und SUBCOM sind verfügbar.

Parsing (Das Zerlegen von Zeichenketten)

Durch Parsing ist ein REXX-Programm in der Lage, Wortketten auf eine sehr komfortable Art in mehrere Variable zu zerlegen. Grundsätzlich ist dies natürlich mit einer der vielfältigen Funktionen von REXX möglich, aber die Variante PARSE drängt sich in vielen Fällen einfach auf. Sehen wir uns einige der gängigen Verfahren an (eine detaillierte Syntaxbeschreibung lesen Sie bitte im Kapitel »Befehlssyntax« nach):

PARSE EXTERNAL

Weist die Eingabe des Anwenders über die Terminaltastatur einer oder mehreren Variablen zu.

```
/* REXX ***************************************/
SAY "Geben sie Ihren Namen an"
PARSE EXTERNAL name
```

Die Bildschirmeingabe wird in der Variablen NAME hinterlegt.

```
/* REXX ***************************************/
SAY "Name und Alter ? (durch BLANK getrennt)"
PARSE EXTERNAL name alter
SAY "Hallo," name","
SAY "Sie sind" alter "Jahre alt."
```

Vorausgesetzt, ein Bediener macht, was man von ihm erwartet (diese Annahme war schon so manchen Programmes vorzeitiges Ende), wird das erste Wort der Tastatureingabe in der Variablen NAME, das zweite Wort in der Variablen ALTER hinterlegt. Beide Werte können beliebig weiterverarbeitet werden.

PARSE PULL

Parsing über den Operanden PULL durchzuführen, ist sicher die meistgenutzte aller Varianten. Je nach Situation verhält sich die Anweisung aber unterschiedlich. Sind Sie sich im Programmablauf nicht absolut über bestimmte Dinge im klären, führt die Anweisung unter Umständen nicht zu dem von Ihnen erwarteten Ergebnis. Beliebt ist diese Form wegen der erlaubten, verkürzten Schreibweise. Soll eine Wortkette einer Variablen zugewiesen und dabei zu Großbuchstaben konvertiert werden, hieße die Syntax hierfür

```
PARSE UPPER PULL variable.
```

Zum gleichen Ergebnis kommen wir mit der Angabe **`PULL variable`**. Nach dem viele EDV-Geschädigte sich zu Minimalisten entwickelt haben, steht diese Kurzform in der Beliebtheitsskala der Programmierer natürlich ganz oben. Nun aber zu den Tücken des Befehles:

Für REXX steht im TSO-Adressraum ein dynamisch verwalteter Speicherbereich zur Verfügung - der Stack. In diesen Bereich können Datensätze durch Instruktionen oder durch Lesen aus einer Datei eingearbeitet werden. Details über den Stack wollen wir hier nicht erörtern. Wie auch immer, PULL prüft immer zuerst den Stack. Liegt dort ein Datensatz, wird er entnommen (der Satz verlässt den Bereich physisch) und in der hinter PULL angegebenen Variablen abgelegt. Ist der Stackbereich leer, wendet sich PULL an die Tastatur. Das Programm wartet auf Bedieneraktion. Tippt der Anwender beliebige Daten, werden sie bei <Enter> der Variablen hinter PULL zugewiesen. Unter dieser Voraussetzung verhält sich der PULL-Befehl identisch zu **`PARSE UPPER EXTERNAL variable`**. Die Versuchung, durch die Angabe **`PULL variable`** zum gleichen Ergebnis zu kommen ist verständlicherweise groß. Dies macht aber nur Sinn, wenn sicher ist, dass beim Lesen von der Tastatur keine Daten im Stack liegen.

PARSE NUMERIC

Über **`PARSE NUMERIC variable`** können die Standards für die Verarbeitung numerischer Werte ermittelt werden. Der Reihe nach werden die aktuellen Werte für DIGITS FUZZ und FORM in der Variablen hinterlegt (könnte beispielsweise sein: 9 0 SCIENTIFIC). Näheres ist im Kapitel »Befehlssyntax« unter NUMERIC nachzulesen.

PARSE SOURCE

Diese Variante liefert eine Reihe interessanter Informationen über das laufende Programm. Details siehe im Kapitel Befehlssyntax unter PARSE SOURCE. Ein kleiner Vorgeschmack aber durch nachfolgendes Beispiel

```
/* REXX ***************************************/
PARSE SOURCE . . progname .
SAY "Hi, Fan - es grüßt Dich Dein Programm:" ,
progname
```

Das dritte Wort der Wortkette enthält den Namen des laufenden Programmes in Großbuchstaben und wird in der Variablen PROGNAME abgelegt. Diese Information ist sehr hilfreich, wenn ein größerer Dialog Meldungen auszugeben hat, und der Name des Moduls, das die Meldung erzeugt, ebenfalls ausgegeben werden soll (Programmnamen direkt in das Programm zu schreiben ist gefährlich. Sollte das Programm einmal umbenannt werden, gibt es viel Aufregung).

PARSE VALUE

PARSE VALUE ist eine komfortable Art, einen String zu zerlegen. Soll beispielsweise die Systemzeit in drei Variablen (Stunde, Minute und Sekunde) zerlegt werden, kann die Trennung des Zeit-Strings über das Erkennen der Doppelpunkte(:) innerhalb der Zeit erfolgen. Die Ausgabe der Systemzeit erfolgt im Format hh:mm:ss.

```
/* REXX ***************************************/
PARSE VALUE TIME() WITH STD ':' MIN ':' SEC
```

Die aktuelle Systemzeit wird über die Funktion TIME() ermittelt. Die Zeichenfolge wird von links nach rechts gelesen. Alle Zeichen bis vor das definierte Trennzeichen (Doppelpunkt) werden der Variablen STD zugewiesen. Das Trennzeichen selbst wird ignoriert. Nach dem Trennzeichen werden alle bis zum nächsten Trennzeichen vorkommenden Daten der Variablen MIN zugewiesen, das Trennzeichen selbst wird wieder ignoriert. Der Rest der durch TIME() gelieferten Zeichenfolge wird in der Variablen SEC hinterlegt.

PARSE VAR

Die Aufteilung einer Wortkette, welche in einer Variablen liegt, ist auf unterschiedliche Arten möglich. Auch hier bietet das Parsing wohl den höchsten Komfort. Nehmen wir an

```
TEXT="Sein, oder nicht sein"
PARSE VAR TEXT WORT1 WORT2 WORT3 WORT4
```

Variable	Inhalt
WORT1	Sein,
WORT2	oder
WORT3	nicht
WIRT4	sein.

Das Komma, das direkt an das erste Wort angrenzt, wird bei der Zerlegung der Wortkette als Bestandteil des ersten Wortes übernommen. Prinzipiell wäre hier auch möglich, ein bestimmtes Zeichen (nehmen wir am besten das Komma) zum Trennzeichen zu erklären. Das Trennzeichen muss in einer Variablen abgelegt werden und die Variable bei der Auftrennung der Wortfolge zwischen runde Klammern gesetzt sein. Das Ganze nochmal in einem Beispiel

```
TEXT="Sein, oder nicht sein"
TRENN=","
PARSE VAR TEXT WORT1 (TRENN) WORT2 WORT3 WORT4
```

Variable	Inhalt
WORT1	Sein
WORT2	oder
WORT3	nicht
WORT4	sein.

Das Komma wird bei Separation der Wortkette nicht übernommen, da es als Trennzeichen definiert wurde.

Durch die Angabe definitiver Bytepositionen kann die Trennung einer Zeichenkette über Parsing auch dann durchgeführt werden, wenn dies auf der Wortbasis nicht möglich ist. Auch hierzu eine kleine Demonstration:

```
:
TEXT="Dasistdochmalwasanderes"
PARSE VAR TEXT 1 WORT1 4 WORT2 7 WORT3 11 ,
               WORT4 14 WORT5 17 WORT6
:
```

Variable	Inhalt
WORT1	Das
WORT2	ist
WORT3	doch
WORT4	mal
WORT5	was
WORT6	anderes

PARSE VERSION

Kann die Versionsnummer und das Datum des REXX-Interpreters ermitteln.

PARSE ARG

Diese Variante erlaubt neben PARSE PULL eine verkürzte Schreibweise. So ist **`ARG var`** gleichbedeutend mit **`PARSE UPPER ARG var`** und bewirkt die Übernahme von Daten, die beim Programmstart mitgeliefert wurden. Die Daten werden dabei zu UpperCase konvertiert. Die Datenübergabe kann auf drei verschiedene Arten erfolgen:

Expliziter Programmaufruf

```
EX TEST(DEMO1) 'Der Pferd hat vier Bein' EXEC
EX 'XWITT.TEST.EXEC(DEMO1)' 'Der Pferd hat vier Bein' EXEC
```

Implizierter Programmaufruf

```
%DEMO1 Der Pferd hat vier Bein
```

Programmaufruf aus einem REXX-Programm

```
CALL DEMO1 'Der Pferd hat vier Bein'
```

Alle diese Varianten bedingen, dass im Programm DEMO1 die Wortkette "Der Pferd hat vier Bein" aufgenommen wird. Wie dies auszusehen hat, sehen wir uns am besten in einigen Beispielen an:

```
/* REXX ****************************************/
ARG DATEN
```

Der Inhalt der Variablen DATEN wäre bei der verkürzten Schreibweise (gleichzusetzen mit PARSE UPPER ARG DATEN) die Wortkette zu Großschreibung konvertiert.

Variable	Inhalt
DATEN	DER PFERD HAT VIER BEIN

```
/* REXX ****************************************/
ARG WORT1 WORT2 WORT3 WORT4 WORT5
```

Hier wird die Wortkette auf Wortbasis zerlegt. Das erste Wort wird der ersten Variablen hinter ARG zugewiesen, das zweite Wort der zweiten Variablen, und so weiter.

Variable	Inhalt
WORT1	DER
WORT2	PFERD
WORT3	HAT
WORT4	VIER
WORT5	BEIN

```
/* REXX ****************************************/
ARG WORT1 WORT2 WORT3
```

Gehen wir vom gleichen Startaufruf aus. Zunächst erfolgt wieder eine Zuweisung Wort nach Variablen von links nach rechts. Sind weniger Variablen hinter ARG angegeben als Worte im Startaufruf übergeben wurden, führt dies nicht zum Datenverlust, sondern alle überzähligen Worte werden in der letzten Variablen gesammelt.

Variable	Inhalt
WORT1	DER
WORT2	PFERD
WORT3	HAT VIER BEIN

```
/* REXX ****************************************/
ARG WORT1 WORT2 WORT3 WORT4 WORT5 WORT6 WORT7
:
```

Ebenso könnten in der ARG-Instruktion mehr Variablen vorgesehen sein, als übergeben werden. Für die überzähligen Variablen erfolgt eine NULLSTRING-Zuweisung.

Variable	Inhalt
WORT1	DER
WORT2	PFERD
WORT3	HAT
WORT4	VIER

WORT5	BEIN
WORT6	Nullstring
WORT7	Nullstring

```
/* REXX ***************************************/
PARSE ARG WORT1 WORT2 WORT3 WORT4 WORT5
```

Wird die Klausel PARSE kodiert, wird der Befehl nicht mehr in seiner ursprünglichen Form (PARSE UPPER ARG) interpretiert. Die Konvertierung der Variablen auf Großschreibung unterbleibt.

Variable	**Inhalt**
WORT1	Der
WORT2	Pferd
WORT3	hat
WORT4	vier
WORT5	Bein

Der Punkt als Begrenzungszeichen

Häufig ergibt sich das Problem, dass ein Programm nur eine begrenzte Anzahl von Parametern übernehmen soll, der Anwender aber bei Programmaufruf seine ganze Lebensgeschichte liefert. Alle überzähligen Worte würden in der letzten Variablen hinter ARG gesammelt werden. Gehen wir davon aus, dass zwei Worte übernommen werden sollen, der unter Umständen gelieferte Rest soll nicht berücksichtigt werden. Dieses Problem kann gelöst werden, indem ein Begrenzungszeichen die Zuweisung des Parsings beendet.

```
/* REXX ***************************************/
PARSE ARG WORT1 WORT2 .
```

Variable	**Inhalt**
WORT1	Der
WORT2	Pferd

Der Punkt als Platzhalter

Der Punkt kann auch als Platzhalter bei der Wertzuweisung des Parsings eingesetzt werden. Erfolgt die Zuweisung der einzelnen Parameter-Worte zu den der Reihe nach angegebenen Variablen, kann der Punkt an die Stelle einer Variablen treten. Das dazu analoge Parameterwort wird bei der Zuweisung entsprechend ignoriert.

```
/* REXX ***************************************/
ARG WORT1 . WORT2 . WORT3
```

Variable	**Inhalt**

WORT1	Der
WORT2	hat
WORT3	Bein

oder

```
/* REXX ****************************************/
ARG WORT1 . . WORT2 .
```

Variable	**Inhalt**
WORT1	Der
WORT2	vier

Nachdem wir nun mehrfach einen kleinen Einzeiler als Demonstration für Datenübergabe an ein Programm benutzt haben, hier die vollständige Fassung des Gedichtes von Heinz Erhardt:

Der Pferd hat vier Bein,

an jedem Eck ein.

Wenn Pferd drei Bein hätt' -umfall'n tät'.

Testhilfen, Ablaufverfolger, Fehlerbehandlung, Aktionen auf die PA1-Taste

Es gibt nicht umsonst eine Reihe mehr oder minder intelligente Aussprüche wie: ***Ein Programm macht was man kodiert, nie was man will!*** Leistungsfähige, moderne Hochsprachen wie REXX verleiten dazu, im Hau-Ruck-Stil zu programmieren (erst mal alles hinschreiben, was einem so einfällt; Programm laufen lassen; angezeigte Fehler korrigieren; neuer Versuch). Vieles lässt sich auf diese Weise bewältigen. Manche Fehler sind aber etwas komplizierter. Tritt ein Fehler in einem Statement als Folge einer falschen Zuweisung in einem vorangegangenen Befehl auf, oder handelt es sich nicht um syntaktische Mängel im Programm, sondern um logische Fehler, die zu einem anderen als dem gewünschten Programmverlauf führen, ist die Suche nach der Ursache oft nicht so einfach. REXX bietet an dieser Stelle einen Ablaufverfolger in Form des Befehles TRACE (eine detaillierte Beschreibung der Möglichkeiten und Syntax des Befehles finden Sie im Kapitel »Befehlssyntax«). Die gängigste Form, den Ablaufverfolger zu starten, ist die Angabe **`TRACE '?R'`**.

Man sieht in der Praxis häufig eine leicht abartige Vorgehensweise im Umgang mit TRACE. Läuft ein Programm nicht wie es soll, wird der TRACE-Befehl in das Programm kodiert. Testlauf. Man glaubt, den Fehler gefunden zu haben, korrigiert ihn und löscht den TRACE-Befehl wieder. Nächster Versuch - neuer Fehler. Einfügen des TRACE-Befehles. Programmstart. Fehlererkennung - Korrektur - TRACE löschen...

Sinnvoller ist, das Programm so zu schreiben, dass das Aktivieren des Ablaufverfolgers über die Art des Programmaufrufes geschehen kann. Wir gehen im nachfolgenden Beispiel davon aus, dass das Programm sich normal verhält, wenn bei Programmstart keine Parameter angegeben werden. Wird bei Programmaufruf als Parameter dagegen ein T (wie Test) übergeben, soll der Ablaufverfolger eingeschaltet werden.

```
/* REXX
*/
ARG TEST .
IF TEST = "T" THEN TRACE "?R"
```

Im Modus ?R ist dem Anwender während des Programmablaufs die Kommunikation mit dem Interpreter möglich. Die Befehle werden interaktiv ausgeführt. Der Interpreter zeigt einen Befehl, führt ihn aus, teilt das Ergebnis des Befehles mit und setzt einen Haltepunkt. Am Haltepunkt bestimmt der Anwender, wie es weitergeht. Drückt der Anwender <enter>, ohne Daten einzugeben, geht der Interpreter an die Ausführung des nächsten Programmbefehles in gleicher Weise. Nach Befehlsausführung setzt der Interpreter wieder einen Haltepunkt und der Anwender ist wieder am Zug. Prinzipiell ist dem Anwender am Haltepunkt jede beliebige Befehlseingabe möglich, die der Interpreter verarbeiten kann. Dies können REXX-Instruktionen oder Befehle an ein erreichbares Umfeld von REXX sein. Solange der Anwender mit <Enter> eine Eingabe übergibt, versucht der Interpreter diese Eingabe als Befehl auszuführen, ohne mit der

eigentlichen Programmverarbeitung fortzufahren und setzt aufs Neue einen Haltepunkt. Der Anwender hat so die Möglichkeit, während des Programmablaufes die Inhalte von Variablen abzufragen und gegebenenfalls temporär im Ablauf zu verändern. Generell ist ihm jeder Befehl möglich. Wird ein Fehler erkannt, der eine weitere Programmverarbeitung sinnlos macht, wäre eine mögliche Aktion des Anwenders auch die Eingabe des Befehles EXIT, der zum unmittelbaren Programmende führen würde. Achtung bei Dialoganwendungen: Alle Tabellen bleiben geöffnet, alle File Tailoring Outputs verbleiben im virtuellen Speicher.

An dieser Stelle müssen wir uns kurz die standardgemäße Fehlerbehandlung des Interpreters ansehen. Tritt beispielsweise ein Syntaxfehler auf, meldet der Interpreter sehr präzise, was ihm nicht passt (ich frage mich oft, warum er den Fehler nicht gleich selbst korrigiert, wenn er ihn schon so differenziert erkennen kann). Bei Syntaxfehler liefert der Interpreter die Nummer der Zeile aus dem Quellencode, die zum Fehler führte, zeigt deren Inhalt und gibt eine Fehlernummer (RC) und die dazugehörende Fehlermeldung aus. Was will man mehr erwarten? Nun, demonstrieren wir es in einem Beispiel

```
/* REXX
*/
ARG TEST .
IF TEST = "T" THEN TRACE "?R"
A="BlaBla"
B=3
C=7
D=5
ERGEBNIS = A + B * C - D
SAY "Ergebnis:" ERGEBNIS
EXIT
```

Der Interpreter zeigt beim Ablauf des Programmes folgende Meldung:

```
8 +++ ERGEBNIS = A + B * C - D
IRX0041I Error running PROG1, line 8:
Bad arithmetic conversion
```

und beendet das Programm unmittelbar. Die gemeldete Ursache (Bad arithmetic conversion) liegt in einem Fehler bei der Umsetzung der Variablen in der Befehlszeile. Mindestens eine davon hat keinen numerischen Inhalt. Die einzige Variable, die nicht Verursacher des Problems sein kann, ist ERGEBNIS, da sie mit dem Resultat der Auflösung des Ausdruckes rechts vom Gleichheitszeichen hätte gefüllt werden sollen. Welche der Variablen A, B, C oder D aber Grund für den Abbruch war (es können auch mehrere oder alle Variablen falsch gefüllt sein), bleibt verborgen. Werden die Variablen an unterschiedlichsten Stellen des Programmes gebildet und ist ihr jeweiliger Inhalt situationsabhängig, ist die Suche nach der Ursache oft mühsam. Es wäre für uns eine große Hilfe, wenn nach Auftreten des Fehlers, vor Programmende, die

Inhalte der Variablen abgefragt werden könnten. Fehlerbehebung wäre eine Kleinigkeit. Hier ist nun angesagt, die Fehlerbehandlung selbst in die Hand zu nehmen. Dies hat den bitteren Beigeschmack, dass sich der Interpreter um nichts mehr kümmert (Quellencodezeilennummer und -inhalt sowie eine Fehlerursache werden nicht gemeldet). Was soll's, machen wir es eben selbst (Es gibt viel zu tun, gehen wir heim).

SIGNAL heißt das Zauberwort. Über die SIGNAL-Anweisung können wir den Interpreter veranlassen, bei Auftreten einer bestimmten Situation (in unserem Fall SYNTAX-FEHLER) ein Sprungziel innerhalb des Programmes anzusteuern. Was dort zu geschehen hat, können wir selbst veranlassen. Sinnvollerweise muss das Sprungziel der Fehlerbehandlung hinter dem regulären Programmende liegen, damit es bei normalem Ablauf nicht angesteuert wird.

```
/* REXX
*/
ARG TEST .
IF TEST = "T" THEN TRACE "?R"
SIGNAL ON SYNTAX NAME FEHLER
A="BlaBla"
B=3
C=7
D=5
ERGEBNIS = A + B * C - D
SAY "Ergebnis:" ERGEBNIS
EXIT
FEHLER:
PARSE UPPER SOURCE . . progname .
SAY ">>" progname "Zeile" SIGL":" SOURCELINE(SIGL)
SAY ">> RC:" RC "-" ERRORTEXT(RC)
TRACE "?R"
NOP
EXIT
```

Tritt nun im Ablauf der Syntaxfehler auf, kümmert sich der Interpreter nicht um Ursachenforschung, sondern verzweigt zum Label FEHLER: und führt die Befehle aus, die dort kodiert stehen. Der PARSE-Befehl ermittelt den Namen des laufenden Programmes und hinterlegt ihn in der Variablen PROGNAME. Die speziellen Variablen RC und SIGL enthalten den Returncode und die Zeilennummer des fehlerverursachenden Befehles (SIGL ist die Zeilennummer eines Befehles im Quellencode, der aufgrund einer Sprungbedingung (SIGNAL) verlassen wurde. Wurde der Sprung durch SIGNAL ON SYNTAX herbeigeführt, ist es der Befehl, in dem der Syntaxfehler auftrat). Der Syntaxfehler meldet sich für den Anwender mit folgender Optik:

```
>> PROG1 Zeile 9: ERGEBNIS = A + B * C - D
>> RC: 41 - Bad arithmetic Conversion
```

Hier ist das Programm aber nicht unmittelbar zu Ende, die Ablaufsteuerung behält der Interpreter. Der Ablaufverfolger wird eingeschaltet und in den interaktiven Modus versetzt. Bei dieser Gelegenheit eine Charakteristik des Trace-Kommandos und ein neuer Befehl: Nachdem der Interpreter den Quellencode Zeile für Zeile liest und interpretiert, wird der Ablaufverfolger nicht in der Zeile TRACE aktiv (der Interpreter weiß erst was zu tun ist, wenn er die Zeile gelesen hat), sondern zusammen mit dem nächsten Befehl. Wäre dieser nächste Befehl nun EXIT (Programmende), wäre dies ausgesprochenes Pech. Der Ablaufverfolger könnte nicht aktiviert werden. Letztlich spielt es keine Rolle, wie der nächste Befehl heißt, um den Ablaufverfolger einzuschalten, außer EXIT oder RETURN. An dieser Stelle bietet sich eine Instruktion, die ihrerseits nichts tut, direkt an (NOP für No OPeration). Diese Instruktion genügt, um den Ablaufverfolger zu aktivieren und einen Programmhaltepunkt zu setzen. Jetzt ist der Anwender an der Reihe. Solange am Haltepunkt nicht nur <enter> gedrückt wird, versucht der Interpreter die Eingabe als Befehl auszuführen und bleibt in der Programmverarbeitung stehen. Um die problemverursachende Variable zu erkennen ist jetzt nur die Eingabe **`SAY A B C D`** nötig. Die Inhalte der Variablen werden am Bildschirm ausgegeben. Man kann unmittelbar feststellen, wo die Ursache des Problems liegt. Wenn keine Abfragen mehr durchgeführt werden sollen, drückt der Anwender <enter>, ohne Daten einzugeben. Der Interpreter führt den nächsten Befehl aus (EXIT) und beendet das Programm. Gleichgültig, ob es theoretisch möglich wäre, wieder in das Programm an die fehlerverursachende Stelle zurückzukehren oder nicht, es gehört sich einfach nicht.

Wird im Ablauf eines REXX-Programmes <PA-1> gedrückt (ATTENTION-INTERRUPT-KEY), unterbricht der Interpreter zunächst die Verarbeitung und erlaubt dem Anwender eine Eingabe. An dieser Stelle wäre möglich:

Eingabe	**Bedeutung**
HI	Halt Interpretation Programmende, siehe auch SIGNAL ON HALT
HT	Halt Typing Bildschirmausgaben werden unterdrückt
RT	Resume Typing Bildschirmausgaben werden wieder angezeigt
TE	Trace End Der Trace-Modus wird beendet
TS	Trace Start Startet den Ablaufverfolger

Empfehlung

Die einfachste und effektivste Art, ein komplexes Programm auf Fehler hin zu prüfen, ist der Compile! Natürlich werden auch hier nicht alle Fehler gefunden aber die meisten.

Dialog mit dem Anwender

Soll ein REXX-Programm während seines Ablaufes mit dem Anwender kommunizieren, stehen hierfür zwei Befehle zur Verfügung: SAY und PARSE.

Gleich vorweg: Formatierte Bildschirmein-/Ausgaben über Zeilen/Spalten-Koordinaten sind in REXX nicht möglich (geht nicht, gibt's nicht – ist aber mit regulären REXX-Mitteln nicht machbar).

SAY gibt eine Zeile am Terminal aus. Jeder SAY-Befehl erzeugt eine neue Zeile. Sollen nur Leerzeilen erzeugt werden, wird hinter SAY nichts kodiert. Zeichenketten und Variable können in loser Folge im SAY-Befehl ausgegeben werden.

PARSE erlaubt das Lesen von der Tastatur und weist die Eingabe einer oder mehreren Variablen zu.

```
/* REXX
*/
SAY "Geben Sie Ihren Namen ein"
PARSE EXTERNAL name
SAY "Geben Sie Ihr Alter an"
PARSE EXTERNAL alter
SAY "Geben Sie Ihre Körpergröße in cm an"
PARSE EXTERNAL groesse
SAY "Stellen Sie sich aufrecht vor das Terminal"
SAY "Durch mein eingebautes Infrarot-Auge"
SAY "muss ich Ihre Größe prüfen"
SAY "Nachdem Sie sich wieder hingesetzt haben,"
SAY "drücken Sie bitte auf ENTER"
PARSE EXTERNAL
SAY "Ergebnis nach eingehender Prüfung:"
SAY "Sie sind höchstens" groesse-3 "cm groß."
SAY "3 Kilo weniger würden Ihnen sicher guttun..."
:
```

Abfragen und Abfragetechniken

Eines der wichtigsten Sprachelemente in jeder Programmiersprache ist die Entscheidungsfindung. REXX bietet zwei verschiedene Möglichkeiten, eine beliebige Situation zu hinterfragen. Prinzipiell wäre eine Abfrageform zwar ausreichend, um jedes noch so komplizierte Problem zu lösen. Wir werden aber anhand einiger Beispiele erkennen, dass die zwei Varianten in REXX sehr viel Komfort in die Kodierung bringen und jede von ihnen ihre Berechtigung hat.

Einfache Abfragen (IF-THEN-ELSE)

Die einfache Abfrage ist immer dann angebracht, wenn eine Frage mit einem klaren JA(THEN) oder NEIN(ELSE) zu beantworten ist. Wird die Bedingung der Abfrage bejaht, wird der Befehl hinter THEN ausgeführt. Die Kodierung des ELSE-Zweiges ist syntaktisch nicht erforderlich, wenn sie von der Ablauflogik her nicht benötigt wird.

```
SAY "Gib eine Zahl an"
PULL zahl
IF zahl // 2 = 0 THEN SAY zahl "ist gerade"
/* Weitere Verarbeitung */

SAY "Gib eine Zahl an"
PULL zahl
IF zahl // 2 = 0 THEN SAY zahl "ist gerade"
ELSE SAY zahl "ist ungerade"
/* Weitere Verarbeitung */
```

Grundsätzlich gilt für beide Zweige: Es darf nur ein Befehl an die JA- oder NEIN-Bedingung geknüpft werden. Sollen abhängig von einer Bedingung mehrere Befehle ausgeführt werden, sind sie zu einer Befehlsfolge zusammenzufassen, die zwischen DO und END eingegrenzt wird.

```
SAY "Gib Dein Alter an"
PULL alter
IF DATATYPE(alter) = "CHAR" THEN DO
   SAY "Eingabe nicht numerisch"
   SAY "Vorzeitiges Programmende"
   EXIT
END
ELSE DO
   SAY "Bravo"
   /* Weitere Verarbeitung */
END
```

Komplexere Probleme lassen sich durch simple Abfragen oft nicht lösen. Grundsätzlich kann ein IF-Konstrukt eine sehr komplexe Abfragelogik enthalten (Boolesche Verknüpfungen), die Lesbarkeit wird dadurch aber nicht gefördert. Wir wollen versuchen eine Datumsprüfung als abschreckendes Beispiel über ein IF-Gebilde durchzuführen und bedienen uns hierzu dreier Varianten.

Das erste Beispiel zeigt eine Folge von einfachen Abfragen. Nachteil hierbei ist: Wenn eine Fehlerbedingung erkannt ist, hat eine weitere Prüfung im Grunde keinen Sinn mehr und belastet nur unnütz die Systemleistung:

```
/* REXX */
SAY "Gib Dein Geburtsdatum an (Format: tt.mm.jj)"
PULL gebdat
PARSE VAR gebdat 1 tt 3 x1 4 mm 6 x2 7 jj
IF LENGTH(gebdat) >< 8 then say "Falsche Länge"
IF x1!!x2 >< ".." THEN SAY "Falsches Format"
IF DATATYPE(tt!!mm!!jj) = "CHAR" THEN
   SAY "Falsches Format"
IF mm < 01 ! mm > 12 THEN SAY "Monat falsch"
IF POS(mm,"01 03 05 07 08 10 12") > 0 THEN tt_max=31
IF POS(mm,"04 06 09 11") > 0 THEN tt_max = 30
IF mm = 02 THEN IF jj // 4 = 0 THEN tt_max = 29
ELSE tt_max = 28
IF tt > tt_max THEN SAY "Tag falsch"
```

Angenommen, der Anwender hätte 17.aaa.1970 eingetippt, würden wegen der voneinander unabhängigen Prüfungen folgende Meldungen ausgegeben werden:

```
Falsche Länge       /* Länge nicht acht Byte */
Falsches Format     /* Byte 3 und 6 nicht .. */
Falsches Format     /* Tag/Monat/Jahr nicht numerisch */
Monat falsch        /* mm ist kleiner 01 */
Tag falsch          /* TT_MAX wegen falschen Monats nicht
                       Gefüllt, zeigt auf sich selbst, ist
                       somit CHAR, TT>TT_MAX ist true */
```

Versuchen wir es mal mit einem in sich geschachtelten IF-THEN-ELSE-Konstrukt:

```
SAY "Gib Dein Geburtsdatum an (Format: tt.mm.jjjj)"
PULL gebdat
PARSE VAR gebdat 1 tt 3 x1 4 mm 6 x2 7 jjjj
IF LENGTH(gebdat) >< 8 then say "Falsche Länge"
ELSE
   IF x1!!x2 >< ".." THEN SAY "Falsches Format"
   ELSE
      IF DATATYPE(tt!!mm!!jjjj) = "CHAR" THEN ,
         SAY "Falsches Format"
      ELSE
         IF mm < 01 ! mm > 12 THEN SAY "Monat falsch"
         ELSE DO
         IF POS(mm,"01 03 05 07 08 10 12") > 0 THEN ,
            tt_max=31
         ELSE
            IF POS(mm,"04 06 09 11") > 0 THEN ,
```

```
            tt_max = 30
         ELSE
            IF mm = 02 THEN ,
               IF jjjj // 4 = 0 THEN tt_max = 29
               ELSE tt_max = 28
            IF tt > tt_max THEN SAY "Tag falsch"
END
```

Unter gleicher Voraussetzung wird hier nur noch falsches Format aufgrund der Längenprüfung gemeldet. Alle anderen Prüfungen werden übersprungen, da die jeweils nachfolgende Prüfung nur erfolgt, wenn die vorangegangene Prüfung ein logisches FALSE(unwahr) geliefert hat. Aber entscheiden Sie selbst. Schön und lesbar ist etwas anderes.

Für die Enthusiasten der Booleschen Verknüpfungen wäre auch die folgende Lösung denkbar:

```
SAY "Gib Dein Geburtsdatum an (Format: tt.mm.jj)"
PULL gebdat
PARSE VAR gebdat 1 tt 3 x1 4 mm 6 x2 7 jj
IF LENGTH(gebdat) >< 8 ! x1!!x2 >< ".." ! ,
   DATATYPE(tt!!mm!!jj) = "CHAR" THEN
      SAY "Falsches Format"
ELSE
   IF mm < 01 ! mm > 12 THEN SAY "Monat falsch"
   ELSE
      IF POS(mm,"01 03 05 07 08 10 12")>0 & tt > 31 ! ,
         POS(mm,"04 06 09 11") > 0 & tt > 30 ! ,
         mm = 02 & jj // 4 = 0 & tt > 29 ! ,
         mm = 02 & jj // 4 > 0 & tt > 28 THEN ,
            SAY "Tag falsch"
```

Die Änderungsfreundlichkeit bei dieser Version ist gleich Null. Durch die mehrfache Verknüpfung von Ausdrücken bleibt die Lesbarkeit eines Programmes häufig auf der Strecke. Im Ablauf des Programmes liefern die Verknüpfungen keinerlei Vorteile, da eine komplexe, verknüpfte Abfrage intern vom Interpreter in mehrere einzelne Abfragen zerlegt wird. Darüber hinaus ist bei Verknüpfungen eine gezielte Meldungsausgabe nicht möglich (warum ist die Tagesangabe falsch). Beurteilungskriterium, ob ein Programm gut oder schlecht ist, ist ganz sicher nicht, ob es funktioniert oder nicht. Dass ein Programm ein richtiges Ergebnis liefert, ist eine Grundbedingung und das Mindeste, was man erwarten kann. Erfüllt es noch nicht mal diese Bedingung, ist es kein Programm, sondern höchstens ein Versuch. Im Vordergrund stehen die Lesbarkeit, die Änderungsfreundlichkeit, die Ablaufgeschwindigkeit und die Größe des Programmes, in genannter Reihenfolge.

Mehrfachabfragen (SELECT)

Mehrfachabfragen sind sinnvoll, wenn zur Entscheidungsfindung eine Fragestellung mit einem simplen JA/NEIN nicht ausreicht. Von der Interpretation her funktioniert eine Mehrfachabfrage wie ein ineinander verschachteltes IF-THEN-ELSE-Konstrukt, ist aber wesentlich einfacher zu lesen. Wie bei der IF-THEN-ELSE-Verschachtelung wird die zweite Frage der Mehrfachabfrage nur gestellt, wenn die erste Frage verneint wurde. Die dritte Frage wird nur berücksichtigt, wenn die ersten beiden Fragen negativ beantwortet wurden, und so weiter. Trifft eine Fragestellung zu, erfolgt die vorgesehene Aktion. Auch hier gilt: An das Bejahen einer Frage kann nur ein Befehl geknüpft werden. Sollen mehrere Befehle abhängig vom JA einer Frage ausgeführt werden, sind sie als Befehlsfolge zwischen DO und END einzugrenzen.

Ist die Aktion des JA-Falles durchgeführt, wird das Mehrfachabfragengebilde verlassen, ohne dass weitere Fragestellungen berücksichtigt werden.

Auch für die Mehrfachabfrage kann ein NEIN-Zweig kodiert werden. Dieser NEIN-Zweig wird durchlaufen, wenn von allen gestellten Fragen innerhalb der Mehrfachabfrage keine zutreffend war.

In REXX wird die Mehrfachabfrage mit dem Befehl **SELECT** eingeleitet und durch **END** abgeschlossen. Die einzelnen Fragen werden über das Verb **WHEN** gestellt, der JA-Zweig wird durch **THEN** definiert. Der NEIN-Zweig, bezogen auf alle vorausgegangenen WHEN-Klauseln, wird mit **OTHERWISE** kodiert. Sehen wir uns ein kleines Beispiel dazu an:

```
SAY "Wähle unter folgenden Möglichkeiten"
SAY;SAY
SAY "1 <=> Black Jack"
SAY "2 <=> Knobeln"
SAY "3 <=> Mastermind"
SAY "4 <=> Poker";SAY
SAY "9 <=> Programmende"
PULL auswahl
SELECT
   WHEN auswahl = 9 THEN EXIT
   WHEN auswahl = 1 THEN CALL BLK_JACK
   WHEN auswahl = 2 THEN CALL KNOBEL
   WHEN auswahl = 3 THEN CALL MST_MIND
   WHEN auswahl = 4 THEN CALL POKER
   OTHERWISE SAY "Auswahl falsch"
END
```

Stellen wir dem SELECT-Gebilde eine IF-Konstruktion mit gleicher Auswirkung gegenüber, erübrigt sich jeglicher Kommentar:

```
IF auswahl = 9 then exit
ELSE IF auswahl = 1 THEN CALL BLK_JACK
     ELSE
        IF auswahl = 2 THEN CALL KNOBEL
        ELSE
           IF auswahl = 3 THEN CALL MST_MIND
           ELSE
              IF auswahl = 4 THEN CALL POKER
              ELSE SAY "Auswahl falsch"
```

Das SELECT-Gebilde lässt sich durch seine klare und einfache Struktur wesentlich besser lesen, ist leichter verständlich und macht auch dem, der ein Programm nicht geschrieben hat, viel leichter verständlich, was es bezwecken soll. Lesbarkeit und Änderungsfreundlichkeit gewinnen deutlich. Sehen wir uns an dieser Stelle die Datumsüberprüfung in Form einer Mehrfachabfrage an:

```
SAY "Gib Dein Geburtsdatum an (Format: tt.mm.jj)"
PULL gebdat
PARSE VAR gebdat 1 tt 3 x1 4 mm 6 x2 7 jj
MAX02 = 28+(jj//4=0)
SELECT
   WHEN LENGTH(gebdat) <> 8 THEN SAY "Falsche Länge"
   WHEN x1!!x2 <> ".." THEN
      SAY "Trennzeichen muss Punkt(.) sein"
   WHEN DATATYPE(tt!!mm!!jj) = "CHAR" THEN ,
      SAY "tt/mm/jj müssen numerisch sein"
   WHEN tt < 01 ! tt > 31 THEN SAY "Tag > 31"
   WHEN mm < 01 ! mm > 12 THEN SAY "Monat falsch"
   WHEN POS(mm,"04 06 09 11") > 0 & tt > 30 THEN ,
      SAY "Monat" mm "hat nur 30 Tage"
   WHEN mm = 02 & tt > MAX02 THEN ,
      SAY "Februar hat" MAX "Tage"
   OTHERWISE NOP
END
```

Schleifenverarbeitung

Schleifen sind immer angesagt, wenn Programmbefehle wiederholt auszuführen sind. Wenn wir davon ausgehen, dass die SIGNAL-Anweisung nicht als GOTO-Ersatz missbraucht wird (wer es tut, hat verdient, was er bekommt!), wird die Schleife auch in Verbindung mit Plausibilitätsprüfungen von Eingaben genutzt. Wird eine Eingabe als falsch erkannt, soll eine Fehlermeldung ausgegeben und dem Bediener eine neue Eingabe erlaubt werden. Die Rückkehr an einen Punkt des Programmes nach oben kann nur über eine Schleife erfolgen (der Maximalpunkt für die Rückkehr nach oben ist der Schleifenkopf einer aktiven Schleife). REXX bietet fünf verschiedene Formen der Schleife an. Dies wirft oft die Frage auf, welche Schleife für eine Problemstellung benutzt werden soll. Prinzipiell kann jedes Problem mit jeder der fünf Möglichkeiten gelöst werden, doch zeigt die praktische Anwendung, dass sich meist eine Form geradezu aufdrängt. Gleichgültig, für welche Form man sich entscheidet, eines haben alle Varianten gemeinsam: Der Beginn der Schleife (Schleifenkopf) wird immer mit **DO** eingeleitet, das Schleifenende wird mit **END** markiert. Zwischen Schleifenbeginn und -ende werden die Befehle kodiert, die wiederholbar sein sollen. Das Statement DO steht nie alleine, sondern wird immer von der Schleifenbedingung gefolgt.

Die simple Wiederholung (DO zähler)

Hinter DO wird die Anzahl der Schleifendurchläufe angegeben. REXX vermindert die Zahl bei jedem Durchlauf um eins. Ist der Zähler bei 0 angelangt, wird hinter das zur Schleife gehörende END verzweigt und die Verarbeitung fortgesetzt. Der Schleifenzähler muss Integer sein, kann aber auch über eine Variable geliefert werden.

```
SAY CENTER("< Zentrierte Titelzeile >",79,"-")
DO 3
   SAY
END
```

Der SAY-Befehl in der Schleife wird dreimal ausgeführt und erzeugt nach der Überschriftszeile drei Leerzeilen.

Um ein weiteres praktisches Beispiel für die einfache Schleife zu zeigen, nehmen wir eine Anleihe bei der Dateiverarbeitung, ohne im Detail darauf einzugehen (siehe Dateiverarbeitung im gleichen Kapitel oder EXECIO im Kapitel »Befehlssyntax«).

```
"EXECIO * DISKR EING (FINIS"
DO QUEUED()
   PARSE PULL satz
   SAY satz
END
```

Gibt alle Datensätze, die durch EXECIO in den STACK-Bereich gelesen wurden, am Terminal aus.

Die Zählschleife (DO variable=zahl TO endewert BY ± zahl)

Sie findet immer Anwendung, wenn während der Schleifendurchläufe wichtig ist, zum wievielten Mal die Schleife durchlaufen wird, oder ein Zähler benötigt wird, der bei jedem Durchlauf der Schleife gleichmäßig nach oben oder nach unten verändert werden soll. Vorteil dieser Schleife ist, dass man sich um das Verändern des Zählers (wir sprechen hier auch von einer Schleifensteuervariablen) innerhalb der Schleife nicht zu kümmern braucht. Um die Vorteile zu veranschaulichen nehmen wir ein simples Beispiel und sehen uns die Lösung dazu zunächst in Form der einfachen Schleife an.

Ein Anwender tippt am Terminal seinen Namen ein. Das Programm gibt den Namen gesperrt geschrieben (zwischen je zwei Zeichen wird BLANK eingefügt) wieder aus:

```
SAY "Gib Deinen Namen ein"
PARSE EXTERNAL name
count = 1
neu_name = ""
DO LENGTH(name)
   neu_name=neu_name SUBSTR(name,count,1)
   count = count + 1
END
neu_name = STRIP(neu_name,L)
SAY neu_name
```

Um bei jedem Durchlauf auf ein anderes Byte des Namens zugreifen zu können, müssen wir eine Variable zunächst mit 1 initialisieren (beim 1. Durchlauf wollen wir ja an das erste Byte), und diese Variable in der Schleife nach erfolgter Verarbeitung um 1 erhöhen. So gelangen wir beim zweiten Durchlauf an das zweite Byte des Namens und so weiter. Hier drängt sich die Zählschleife auf. Zum einen kann die Initialisierung der schleifensteuernden Variablen in die Schleife eingebunden werden, zum anderen müssen wir uns um die Erhöhung der Steuervariablen im Programm nicht kümmern:

```
SAY "Gib Deinen Namen ein"
PARSE EXTERNAL name
neu_name = ""
DO count = 1 TO LENGTH(name)
   neu_name=neu_name SUBSTR(name,count,1)
END
neu_name = STRIP(neu_name,L)
SAY neu_name
```

Die Aussage, dass die Steuervariable bei jedem Durchlauf um eins erhöht wird, entspricht dem Standard und muss nicht kodiert werden. Ist die Schrittweite der Erhöhung nicht 1, oder soll von der Steuervariablen bei jedem Durchlauf ein bestimmter Wert subtrahiert werden, muss die Angabe erfolgen. Sehen wir uns das gleiche Beispiel nochmals an und verarbeiten die Zeichenkette hier von rechts nach links:

```
SAY "Gib Deinen Namen ein"
PARSE EXTERNAL name
neu_name = ""
DO count = LENGTH(name) to 1 BY -1
   neu_name=SUBSTR(name,count,1) neu_name
END
neu_name = STRIP(neu_name,T)
SAY neu_name
```

Im nächsten Beispiel wollen wir alle Zahlen zwischen 1000 und 2000, die ohne Rest durch 25 teilbar sind, am Terminal ausgeben:

```
DO count = 1000 to 2000 BY +25
   SAY count
END
```

Das gleiche Ergebnis wäre zu erzielen, wenn kodiert wird:

```
DO count = 1000 TO 2000
   IF count // 25 = 0 THEN SAY count
END
```

Für ein scheinbar gleiches Ergebnis muss die untere Schleife 1001 mal durchlaufen werden, die obere Schleife liefert ihr Ergebnis mit nur 41 Durchläufen. Außerdem erfolgen in der unteren Schleife auch noch 1001 Prüfungen, von deren Ergebnis abhängig eine Bildschirmausgabe erfolgt. Die untere Schleife benötigt etwa die 6fache Zeit.

Schleifensteuernde Befehle

An dieser Stelle wollen wir uns mit zwei Befehlen auseinandersetzen, die nur innerhalb der Programmschleife sinnvoll einzusetzen sind: **`ITERATE`** und **`LEAVE`**.

ITERATE führt zum einmaligen Schleifendurchlauf, ohne dass die Befehle in der Schleife, die nach ITERATE folgen, ausgeführt werden. Subjektiv gewinnt man den Eindruck, dass ITERATE direkt zum Ansprung des Schleifenkopfes führt. Der Befehl wird häufig in einer Schleife für Plausibilitätsprüfungen eingesetzt, wenn ein Eingabefehler erkannt wurde und weitere Befehle nicht ausgeführt werden sollen. Am Schleifenkopf wird dann in aller Regel eine Neueingabe des Bedieners ermöglicht.

LEAVE führt zum Absprung aus der aktiven Schleife. Obwohl LEAVE im weiteren Sinne auch ein Sprungbefehl ist, kann er nicht mit GOTO verglichen werden, da durch LEAVE nicht ein beliebiger Punkt im Programm aufgesucht werden kann. Es wird immer hinter das zur Schleife gehörende END verzweigt, dort wird die Verarbeitung fortgesetzt. Soll aufgrund einer Bedingung aus mehr als einer Schleife gesprungen werden, kann LEAVE nicht genutzt werden (in Ausnahmefällen kann hierzu SIGNAL kodiert werden, mehr dazu aber unter SIGNAL im Kapitel »Befehlssyntax«).

Die Endlosschleife (DO FOREVER)

Die Endlosschleife (DO FOREVER) zwingt einen zu kodieren, was man auf gar keinen Fall will. Sie ist aber in vielen Fällen eine sehr praktische Variante, für die Wiederholbarkeit von Befehlen zu sorgen, ohne sich komplizierte Bedingungen auszudenken, die das Ende der Schleife beschreiben. Drei Situationen werden in der Praxis sehr gerne über die FOREVER-Schleife geregelt:

1. Ein Programm soll beliebig oft wiederholbar sein und durch Bediener-Eingabe beendet werden.
2. Datenerfassungsschleifen mit Terminaleingabe.
3. Bedienereingabe mit Plausibilitätsprüfungen.

Sehen wir uns zu diesen Situationen je ein kleines Beispiel an:

```
/* REXX * GAMEBOX ******************************/
DO FOREVER
   SAY CENTER("< Heinzi's Spielebox >",79,"-");SAY
   SAY "Wähle durch die entsprechende Zahl";SAY
   SAY "1 <=> Black Jack"
   SAY "2 <=> Knobeln"
   SAY "3 <=> Mastermind"
   SAY "4 <=> Poker";SAY
   SAY "9 <=> Programmende"
   SAY;SAY;SAY
   SAY "Eingabe ??"
   PULL auswahl
   SELECT
      WHEN auswahl = 9 THEN LEAVE
      WHEN auswahl = 1 THEN CALL BLK_JACK
      WHEN auswahl = 2 THEN CALL KNOBEL
      WHEN auswahl = 3 THEN CALL MST_MIND
      WHEN auswahl = 4 THEN CALL POKER
      OTHERWISE SAY "Falsche Auswahl, nochmal"
   END
END
EXIT
```

Das Ende des Programmes wird erreicht, wenn der Bediener sich für die Auswahl 9 (Programmende) entscheidet. Grundsätzlich hätte als Aktion auf diese Eingabe hin das Programm auch unmittelbar beendet werden können (**`WHEN auswahl = 9 THEN EXIT`**), es ist aber besser nur ein ENDE im Programm zu definieren. LEAVE verzweigt hinter das END der aktiven Schleife (es gibt ja nur eine) und trifft dort auf den Befehl EXIT. Das Programm wird beendet.

```
/* REXX * TERM_EIN */
cnt=0
SAY "Eingabe (ENTER = Ende)"
DO FOREVER
   PARSE EXTERNAL ein
   IF LENGTH(ein) = 0 THEN LEAVE
   cnt = cnt + 1
   eingabe.cnt = ein
END
SAY "Empfangene Eingaben:" cnt
/* Hier könnten die Daten verarbeitet werden */
EXIT
```

In diesem Beispiel kann der Anwender beliebig viele Eingaben am Terminal durchführen. Drückt er <Enter>, wird die Eingabeschleife beendet. Sofern in Verbindung mit <Enter> Daten eingegeben werden (auch BLANK sind Daten), wird Eingabe für Eingabe ein Variablenstamm aufgebaut. Die erste Eingabe liegt in der Variablen EINGABE.1, die zweite in der Variablen EINGABE.2 und so weiter. Wie viele gültige Eingaben in der Schleife erfolgten, kann aus dem Wert von CNT abgelesen werden.

Im folgenden Beispiel gehen wir davon aus, dass der Bediener den Namen einer zu verarbeitenden Datei einzugeben hat. Der Name kann voll- oder teilqualifiziert eingegeben werden. Wird eine vollqualifizierte Eingabe gemacht, müssen die Apostrophe mit getippt werden. Die Datei soll über IEBGENER ausgedruckt werden. Folgende Problemsituationen werden abgefangen: Datei nicht verfügbar, PO-Datei ohne Memberangabe, Satzlänge zu groß, Dateiformat ungleich PS und PO.

```
/* REXX * DRUCK ********************************/
DO FOREVER
   SAY "Name Druckdatei (ggf. mit Hochkomma)"
   PARSE UPPER EXTERNAL drudat
   IF LENGTH(drudat) = 0 THEN EXIT
   IF SYSDSN(drudat) = "OK" THEN DO
      DMY = LISTDSI(drudat)
      SELECT
         WHEN POS(SYSDSORG,"PS PO") = 0 THEN
            msg = "Falsche Dateiorganisation"
         WHEN SYSDSORG = "PO" & ,
            RIGHT(drudat,1) >< ")" THEN
            msg = "PO-Datei ohne Memberangabe"
         WHEN SYSLRECL > 132 THEN
            msg = "Satzlänge zu groß"
         OTHERWISE LEAVE
      END
```

```
   END
   ELSE msg = drudat "nicht verfügbar"
   SAY msg
END
"ALLOC DD(SYSIN) DUMMY"
"ALLOC DD(SYSPRINT) DUMMY"
"ALLOC DD(SYSUT1) DA("drudat") SHR REUSE"
"ALLOC DD(SYSUT2) SYSOUT(X)"
"IEBGENER"
"FREE DD(SYSIN SYSPRINT SYSUT1 SYSUT2)"
EXIT
```

Die FOREVER-Schleife kann aus zwei Gründen verlassen werden. Der Anwender drückt auf die Frage nach der Druckdatei nur <enter> ohne Dateneingabe (in diesem Fall wird das Programm unmittelbar beendet. Einen Ausstieg in dieser oder ähnlicher Form sollte man immer schaffen, für den Fall, dass der Anwender eigentlich ein anderes Programm starten wollte), oder der eingegebene Dateiname spricht nicht gegen eine Verarbeitung. In diesem Fall ist sichergestellt:

- RACF hat keine Einwände
- Das Dateiformat ist OK.
- Wurde eine PO-Datei angegeben, ist auch eine Memberangabe erfolgt.
- Die Satzlänge übersteigt 132 Byte nicht.

Treffen alle diese Dinge zu, befinden wir uns innerhalb der Mehrfachabfrage im NEIN-Zweig (OTHERWISE) und verlassen die Schleife nach unten, wo die Zuweisungen für den IEBGENER und dessen Aufruf stattfinden.

Die letzten beiden Formen der Programmschleife sehen auf den ersten Blick gleich aus. Scheinbar besteht der Unterschied nur darin, dass die eine Schleife beschreibt, wie lange eine Verarbeitung wiederholt werden soll, die andere dagegen festlegt, wann die Wiederholung als beendet betrachtet wird. Kehrt man die schleifensteuernde Bedingung ins Gegenteil, erreicht man scheinbar Gleiches. Von da her gesehen hätte man auf eine der beiden Formen verzichten können (unter CLIST war die UNTIL-Schleife vor TSO Release 2 nicht möglich). Bei Licht betrachtet gibt es aber einen ganz gravierenden Unterschied, der im Zeitpunkt der Überprüfung der schleifensteuernden Bedingung liegt.

Die WHILE-Bedingung wird am Schleifenanfang geprüft. Trifft die Bedingung zu, wird die Schleife betreten. Anderenfalls wird hinter das zur Schleife gehörende END verzweigt. Wir sprechen von einer PRE-abweisenden Schleife.

Die UNTIL-Schleife wird zunächst in jedem Fall einmal durchlaufen. Die schleifensteuernde Bedingung wird am Schleifenende überprüft. Ist die Bedingung erfüllt, wird die Schleife verlassen, anderenfalls wird sie aufs Neue durchlaufen. In diesem Fall sprechen wir von einer POST-abweisenden Schleife.

Beide Schleifen finden zu ihrem geregelten Ende, sobald eine Bedingung wahr (UNTIL) oder unwahr (WHILE) ist. Unabhängig davon können beide Schleifen aber auch über LEAVE verlassen werden. Am besten veranschaulichen wir die beiden Schleifenvarianten mit ein paar Beispielen, die verdeutlichen, in welchen Situationen die jeweilige Schleife sinnvoll eingesetzt werden kann, wo unter Umständen beide Schleifen zum Erfolg führen können, oder nur eine der beiden Varianten die sinnvollere ist.

Die PRE-abweisende Schleife (DO WHILE bedingung)

Gehen wir von folgender Situation aus: Ein Programm benötigt für die Verarbeitung eine Zahl vom Anwender. Diese Zahl sollte bei Programmstart bereits mitgeliefert werden können. Wurde bei Aufruf des Programmes keine Zahl übergeben, muss das Programm nachfragen.

```
/* REXX * PZ
*/
ARG zahl
DO WHILE DATATYPE(zahl) = "CHAR"
   SAY "Gib eine Zahl ein"
   PARSE EXTERNAL zahl
END
/* Hier liegt die Verarbeitung */
EXIT
```

Hier macht die UNTIL-Schleife keinen Sinn. Sie würde auch dann einmal durchlaufen werden, wenn die benötigte Zahl durch die Instruktion ARGument bereits zugewiesen worden wäre.

Wir können die WHILE-Schleife auch für vereinfachte Plausibilitätsprüfung benutzen. Soll eine Datei im Programm über den ALLOCATE-Befehl erzeugt werden und der Bediener unter anderem die Dateiorganisationsform, bei der wir nur sequentiell oder gegliedert zulassen wollen, eingeben, könnte das folgendermaßen aussehen:

```
DO WHILE dsorg >< "PS" & dsorg >< "PO"
   SAY "DS-Organisationsform eingeben (PS oder PO)"
   PULL dsorg
END
```

Hier kommt uns das Variablenhandling von REXX sehr gelegen. Sofern die Variable DSORG bis dahin noch nie benutzt wurde, hätten wir in den meisten Sprachen an dieser Stelle Programmabbruch. In REXX dagegen zeigt die Variable auf ihren zu Großschreibung konvertierten Eigennamen, also "DSORG". Aus diesem Grund wäre die Bedingung am Schleifenkopf (Inhalt der Variablen DSORG ist weder "PS" noch "PO") erfüllt, die Schleife wird betreten und so lange durchlaufen, bis der Bediener die Variable mit der Zeichenfolge "PS" oder "PO" füllt.

Die POST-abweisende Schleife (DO UNTIL bedingung)

Nehmen wir an, ein Bediener soll in einer Schleife durch Eingabe entscheiden, ob eine bestimmte Aktion durchgeführt werden soll oder nicht. In vielen Fällen wird hier eine Ja/Nein Eingabe am Terminal ermöglicht. Häufig wird einer der beiden Fälle (JA oder NEIN) geprüft und behandelt. Alle anderen möglichen Eingaben werden gleich (negativ) behandelt.

```
/* REXX * DEMO */
DO FOREVER
   /* Programmverarbeitung */
   SAY "Noch ein Durchlauf (J/N)"
   PULL antwort
   IF antwort = "N" THEN LEAVE
END
EXIT
```

Ein gängiges Verfahren. Wie sinnvoll es ist, möchte ich hier nicht kommentieren. Was dabei herauskommen kann, zeigt am besten das folgende kleine Beispiel

```
/* REXX * DEMO */
CALL OUTTRAP "datei."
"LISTC"
CALL OUTTRAP "OFF"
DO cnt = 2 TO datei.0
   SAY datei.cnt "löschen oder behalten (L/B)"
   PULL antwort
   IF antwort = "B" THEN ,
      SAY datei.cnt "bleibt erhalten"
   ELSE DO
      "DEL '"datei.cnt"'"
      SAY datei.cnt "gelöscht"
   END
END
EXIT
```

Antwortet der Bediener auf die Frage nach dem Löschen mit B, bleibt die Datei erhalten. Für alle anderen Fälle, auch Tippfehler, erfolgt das Löschen der Datei. Prima! Sinnvoller wäre hier im Programm bereits sicherzustellen, dass die Bedienereingabe nur die Werte liefern kann, die das Programm erwartet:

```
DO UNTIL antwort = "L" ! antwort = "B"
   SAY datei.cnt "löschen oder behalten (L/B)"
   PULL antwort
END
```

"HALT" wird der aufmerksame Leser hier einwenden. Es geht genauso gut mit:

```
DO WHILE antwort >< "L" & antwort >< "B"
   SAY datei.cnt "löschen oder behalten (L/B)"
   PULL antwort
END
```

was bedingt richtig ist. Da die Abfrageschleife in unserem Beispiel aber in eine äußere Programmschleife für die Abarbeitung der einzelnen Dateinamen integriert ist, tritt ein Problem auf: Wird die WHILE-Schleife zum ersten Mal angegangen, ist der Inhalt der Variablen ANTWORT ihr großgeschriebener Eigenname und erfüllt damit die Bedingung ungleich L und ungleich B. Die Schleife wird betreten, der Anwender kann seine Eingabe machen. Gelingt es ihm, ein L oder B zu tippen, wird die Schleife verlassen und die entsprechende Aktion (Datei löschen oder nicht löschen) wird durchgeführt. Das Programm kehrt an den äußeren Schleifenkopf zurück. Steht der Interpreter erneut am Kopf der WHILE-Schleife, ist ihre Bedingung nicht erfüllt (ANTWORT enthält immer noch das L oder B der ersten Eingabe). Die Schleife wird übersprungen. Auch für die zweite und alle weiteren Dateien erfolgt die gleiche Aktion wie für die erste. Im Falle Löschen der ersten Datei wird dies dem Anwender sicher Tränen in die Augen treiben. Andererseits schafft es natürlich enormen Platz auf dem Plattenpool!

Damit die WHILE-Schleife hier vernünftig funktionieren kann, muss die Variable ANTWORT vor erneutem Anlauf des Schleifenkopfes wieder zurückgesetzt werden. Theoretisch könnte der Befehl dazu an beliebiger Stelle innerhalb der äußeren Schleife liegen. Im Hinblick auf spätere Programmänderungen ist es am sinnvollsten, das Rücksetzen der Variablen direkt über dem Schleifenkopf durchzuführen. An dieser Stelle erklärt sich der Befehl selbst.

```
DROP antwort
DO WHILE antwort >< "L" & antwort >< "B"
   SAY datei.cnt "löschen oder behalten (L/B)"
   PULL antwort
END
```

Wird dagegen die UNTIL-Schleife eingesetzt, wird sie zunächst einmal sicher durchlaufen, gleichgültig, ob beim ersten oder einem folgenden Durchlauf der äußeren Schleife. Der Anwender bekommt so in jedem Fall bei jedem Schleifendurchlauf die Möglichkeit, seine Antwort auf die Frage nach der Aktion einzugeben. Ein Rücksetzen der Variablen ANTWORT ist nicht nötig.

An dieser Stelle möchte ich eine eindringliche Bitte an Sie richten: Verzichten Sie auf jeden Befehl, den Sie nicht unbedingt benötigen. Alles, was kodiert wird, muss stimmen. Nur die Befehle, die nicht geschrieben werden, können nicht falsch sein. Der eine oder andere Befehl mehr ist nur sinnvoll, wenn die Lesbarkeit des Programmes dadurch verbessert wird, wobei dies ein relativer Begriff ist, abhängig vom Kenntnisstand des Programmierers.

Kombinationen

Achtung, wir bewegen uns auf eine gefährliche Stelle zu. Grundsätzlich können die verschiedenen Schleifen auch miteinander kombiniert werden. Wenn Ihr Seelenheil als Programmierer nicht davon abhängt, versuchen Sie es zu vermeiden. Nicht alles, was theoretisch machbar ist, ergibt in der Praxis einen Sinn. Ein Schleifenkopf wie

```
DO §=1 TO 1E6 WHILE z < 98 & do_it = "J" UNTIL x > y
```

ist, was den reinen Formalismus angeht, ohne weiteres möglich. Ich bedauere aber bei derartigen Konstrukten immer den armen Mitarbeiter, der ein so schlimmes Gebilde im Nachhinein zu pflegen oder zu ändern hat. Leider sind der Erzeuger und der Änderer eines Programmes in aller Regel verschiedene Personen. Es ist sehr aufwendig, sich in eine Programmlogik mit komplexen und komplizierten Verknüpfungen und Bedingungen einzulesen, wenn man sie nicht selbst geschrieben hat. Und selbst dann fällt es einem oft schwer, nachzuvollziehen, was man vor Jahren gedacht hat, als man derartige, monströse Gebilde ersonnen hatte. REXX-Programme werden häufig nicht oder nur sehr spärlich dokumentiert. Die Kunst der Programmierung besteht nicht darin, möglichst komplizierte Programme zu schreiben, sondern einen Ablauf in simple, leicht verständliche Schritte zu zerlegen und in möglichst einfache und leicht lesbare Befehle umzuwandeln. Kompliziert wird ein Programm oft von ganz alleine. Programmierer, die durch eine gewollt komplizierte Kodierung ihren Arbeitsplatz sichern wollen, sollten heute eigentlich der Vergangenheit angehören.

Keine Regel ohne Ausnahme

Es gibt Situationen, in denen eine Vermengung unterschiedlicher Schleifenkonstruktionen von großem Vorteil sein kann. Wir sehen uns mal eine an. Gehen wir von folgender Situation aus: Ein Programm benötigt vom Anwender Informationen direkt bei Aufruf. Natürlich sollte der Anwender darüber informiert sein, doch oftmals sieht die Praxis eben ein wenig anders aus als die Theorie. Auf PC-Ebene hat sich durchgesetzt, dass ein Programm durch Startparameter wie /H oder /? eine kleine Hilfestellung über seine Bedienung liefert. Vergleichbares wollen wir auch in unsere REXX-Programme integrieren. Ausgehend davon, dass ein Programmkopf kodiert wird, in dem die wichtigsten Informationen über die Bedienung eines Programmes stehen, wollen wir erreichen, dass, wenn ein Bediener keine oder falsche Angaben beim Startaufruf übergibt, oder über Startaufruf ein Fragezeichen(?) geliefert wird, dieser Programmkopf als Hilfestellung für den Bediener ausgegeben wird. In unserem Beispiel erfolgt die Ausgabe des Schleifenkopfes ab der Zeile, in der die Zeichenfolge "Start" vorkommt.