Das REXX Lexikon

Begriffe, Anweisungen, Funktionen

von
Gerhard Leibrock

R. Oldenbourg Verlag München Wien 1997

Für G., H. und S.

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Leibrock, Gerhard:
Das REXX-Lexikon : Begriffe, Anweisungen, Funktionen / von Gerhard Leibrock. - München ; Wien : Oldenbourg, 1996
ISBN 3-486-23672-5
NE: HST

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0, Internet: http://www.oldenbourg.de

Lektorat: Margarete Metzger
Herstellung: Rainer Hartl
Umschlagkonzeption: Mendell & Oberer, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: R. Oldenbourg Graphische Betriebe GmbH, München

Vorwort

Lieben Sie REXX? Fehlt Ihnen ein umfassendes Nachschlagewerk, in dem Sie alles über REXX konzentriert und wohlgeordnet vorfinden? Möchten Sie aus einem Fundus von 200 handfesten und kommentierten Praxisbeispielen (und 1000 Syntaxbeispielen) schöpfen? Wenn Sie diese Fragen mit Ja beantworten, ist dieses Buch für Sie richtig.

Das REXX-Lexikon behandelt sämtliche Begriffe, Anweisungen und Funktionen, die für die Betriebssysteme PC-DOS, OS/2 und MVS/TSO von Bedeutung sind. Die Stichworte, die sich auf den "Grundwortschatz" von REXX beziehen, sind darüberhinaus auch für alle anderen Betriebssysteme geeignet (OS/400, UNIX, VM/CMS, VSE/ESA, Windows),

Seit ich REXX kenne, ist es eine Fron für mich, wenn ich in C oder PL/I programmieren muß, von anderen Programmiersprachen nicht zu reden. REXX ist die erste Programmiersprache, die sich nicht vor dem Entwerfer der Sprache und ihren Implementierern verbeugt, sondern vor ihrem Anwender. *Mike F. Cowlishaw*, der Erfinder der Sprache REXX, hat seine Philosophie in einem bezeichnenden Kernsatz formuliert: *"Der Benutzer der Sprache hat üblicherweise recht."*. Wer will, kann Cowlishaws Entwurfsmotive in seinem Buch *"Die Programmiersprache REXX"* nachlesen.

Wenn Sie sich für dieses Nachschlagewerk entscheiden, gehe ich davon aus, daß Sie ein "working knowledge" der REXX-Sprache besitzen. Vielleicht geht es Ihnen genauso wie mir: Das Benutzerhandbuch legt man schnell zur Seite, wenn man seine ersten Gehversuche mit einer Sprache gemacht hat. Dann kommt das Referenzbuch (papieren oder elektronisch) zum Tragen. Aber leider sind diese Werke meist systematisch aufgebaut, und oft findet man die Informationen zu einem Stichwort über das ganze Buch verstreut, so daß man zum Suchen verdammt ist. Und die Beispiele erst! Sie sind, falls vorhanden, häufig mehr als dünn und nehmen auf die praktische Anwendung kaum Rücksicht.

Dieses Buch, das wir REXX-Lexikon getauft haben, ist eigentlich ein Referenzhandbuch, so wie es der Anwender in der täglichen Praxis benötigt. Unter jedem der einigen hundert Stichwörter wird - wie in einer Enzyklopädie - alles Wissenswerte zu einem Begriff, einer Anweisung oder Funktion versammelt. Die ausführliche Beschreibung von Syntax und Wirkung wird ergänzt um Tips zur praktischen Anwendung und durch Hinweise auf Zusammenhänge mit anderen Themen. Neben reichhaltigen Syntaxbeispielen werden über 100 ausführlich kommentierte Anwendungsbeispielen geboten, die den Reichtum und die Praxisnähe der Sprache erst richtig zur Geltung bringen. Alle Anwendungsbeispiele und die im Text erwähnten externen REXX-Programme finden Sie auf der beiliegenden CD-ROM. Ein herkömmliches Stichwortverzeichnis erschließt die Begriffe, die nicht in eigenen Stichwortartikeln beschrieben werden können.

REXX ist auch für die Zukunft bestens gerüstet:

- Das Programmieren der OS/2-Arbeitsoberfläche ist mit VxRexx von Watcom und anderen Produkten von Drittherstellern ein Kinderspiel und in dieser Form in kaum einer der bekannten Programmiersprachen oder Programmierumgebungen auch nur annähernd so elegant gelöst.
- Für OS/2 Warp 3 liegt bereits eine objektorientierte Fassung von REXX vor (OREXX), die einfacher als andere OOP-Implementierungen den Umstieg in eine neue Welt ermöglicht. Für Windows 95, Windows NT und Linux wird OREXX im Laufe des Jahres 1996 verfügbar. Die verschiedenen OREXX-Versionen finden Sie im Internet unter:
  http://www2.hursley.ibm.com./rexx bzw. orexx
- Im Laufe des Jahres 1996 wird NETREXX entwickelt, eine REXX-Version, mit der auf einfache Weise Java-Programme mit den hochentwickelten Möglichkeiten von OREXX erzeugt werden können. NETREXX finden Sie im Internet unter:
  http://www2.hursley.ibm.com./netrexx

Der Autor nimmt gerne Kritik, Verbesserungsvorschläge, Tips und Ergänzungen entgegen.

Gerd Leibrock
Reinsburgstr. 99
70197 Stuttgart

Danksagung

Eine Reihe von Leuten hatten Anteil an dem Zustandekommen dieses Buches:

Manfred Blaschke hat wie kein anderer in Hunderten von Programmen REXX nicht nur als Prozedursprache, sondern als Programmiersprache eingesetzt, wodurch mir zum ersten Mal die Mächtigkeit dieser Sprache klar geworden ist.

Jürgen Klaiber, Thilo Matthies und *Rudi Dettling* haben mir durch Probelesen und durch ihre kritischen Anmerkungen Mut zum Weitermachen gemacht.

Angelika Michaelis, Walter Pachl, Dr. Robert Spencer, Dietmar Stroh und den anderen Mitgliedern der *REXX-Arbeitsgruppe im GSE* (GUIDE/SHARE Europa) verdanke ich viele Detail-Informationen, Walter Pachl insbesondere einen reichhaltigen Beispielfundus.

Den Mitarbeitern der REXX-Gruppe im *IBM-Labor Böblingen* unter der Leitung von *Christoph Dibon*, in deren Händen die Weiterentwicklung von REXX liegt, danke ich für Hintergrundinformationen und für Einblicke in die zukünftige Entwicklung von REXX.

Dem Chefredakteur *Frank-Martin Binder* von *Inside OS/2* danke ich für die Genehmigung, die Artikel zum *REXX Award* auszuwerten.

Meinem Freund *Klaus-Waldemar Hennig* danke ich für viele anregende Gespräche, auch wenn er sich bisher zu meinem Kummer beharrlich geweigert hat, REXX zu erlernen.

Meiner erfrischend unkonventionellen Lektorin *Margarete Metzger* danke ich für ihre nette Unterstützung und ihr freundliches Verständnis.

Schließlich danke ich all den ungenannten REXX-Programmautoren, bei denen ich abgeschrieben oder von denen ich gelernt habe.

Inhaltsverzeichnis

CD-ROM

Die beiliegende CD-Rom enthält sämtliche Anwendungsbeispiele, die Sie in diesem Buch unter der Überschrift *Anwendung* zu einem Stichwort finden.

Installation unter DOS und OS/2

Zum Installieren rufen Sie in dem Verzeichnis *\lex\dos* bzw. *\lex\os2* das REXX-Programm *install* auf der CD-ROM auf. Sie werden nacheinander nach den folgenden Angaben gefragt:

- Zielsystem (DOS, OS/2)
- Ziellaufwerk
- Name eines existierenden, leeren Zielverzeichnisses

Anschließend werden die Beispielprogramme und -dateien auf das Ziellaufwerk kopiert. Damit die Installation beendet.

Installation unter TSO

Auf der CD-ROM finden Sie die folgenden Stapeldateien zum Senden der Beispielprogramme und -dateien an den Hostrechner:

- **\lex\tso\tsosend.bat:** für DOS-Recchner, wenn die REXX-Zeichen | und \ auch auf dem Hostrechner gelten.
- **\lex\tso\tsosend.cmd:** für OS/2-Recchner, wenn die REXX-Zeichen | und \ auch auf dem Hostrechner gelten.
- **\lex\tso\tsoconv.bat:** für DOS-Recchner, wenn die REXX-Zeichen | und \ auf dem Hostrechner durch ! bzw. ^ ersetzt werden sollen.
- **\lex\tso\tsoconv.cmd:** für OS/2-Recchner, wenn die REXX-Zeichen | und \ auf dem Hostrechner durch ! bzw. ^ ersetzt werden sollen.

Die Stapeldateien bestehen aus *send*-Befehlen, mit denen die Beispielprogramme und -dateien zum Hostrechner gesendet werden können. Zum Aufruf der Stapeldatei müssen Sie den Namen des Ziel-PDS als Parameter übergeben, z. B.

```
tsosend sptgl01.t.ispf
```

oder:

```
tsoconv sptgl01.t.ispf
```

Ausführung der Beispiele

Unter DOS und OS/2 müssen Sie sich in dem von Ihnen bei der Installation angegebenen Zielverzeichnis befinden und das Programm *bsp* aufrufen.

Unter TSO müssen Sie zuerst das Ziel-PDS mit dem TSO-Befehl *altlib* als EXEC-Datei anmelden. Wenn Ihr Ziel-PDS z. B. *"sptgl01.t.ispf"* heißt, geben Sie im TSO-Eingabemodus den folgenden Befehl ein:

```
altlib act application(exec) dataset('sptgl01.t.ispf')
```

Anschließend rufen Sie das Programm *bsp* im TSO-Eingabemodus durch Eingabe von *bsp* auf. Nach dem Start des Programms *bsp* geben Sie zuerst ein Stichwort aus dem Lexikon ein:

```
Stichwort eingeben, z. B. "abbrev"
letztes Stichwort kopieren = Enter, Ende = X):
schlüsselwortparameter
```

und anschließend die Nummer des Beispiels, das zu einem Stichwort unter der Überschrift *Anwendung* im Lexikon abgedruckt ist. Falls nur ein Beispiel vorhanden ist, geben Sie als Nummer *1* ein:

```
Nummer des Anwendungsbeispiels eingeben, z. B. 2:
1
```

Das Beispielprogramm wird ausgeführt. Anschließend können Sie ein neues Stichwort eingeben oder das Programm *anw* mit einer leeren Eingabe beenden.

Benutzungshinweise

Typographische Konventionen

Leerstellen

Wenn es darauf ankommt, daß Leerstellen eindeutig als solche erkannt werden können, werden sie durch ein Kästchen (□) dargestellt:

```
dateibez2 = translate(dateibez1, "□□□", ":\.")
```

Anführungszeichen und Hochkommas

Doppelte Anführungszeichen, doppelte Hochkommas und Kombinationen von Anführungszeichen und Hochkommas werden zur besseren Lesbarkeit durch eine scheinbare Leerstelle getrennt und unterstrichen:

```
"", '', "'", '"'
```

Wenn zwischen diesen Zeichen eine wirkliche Leerstelle dargestellt werden soll, wird ein Kästchen (□) verwendet:

```
"□", '□', "□'□", '□"□'
```

Zeilenumbrüche in REXX-Anweisungen

Wenn eine REXX-Anweisung nicht in eine Zeile paßt, wird die Zeile umbrochen und die nächste Zeile durch Auslassungspunkte (...) eingeleitet:

```
translate("45.12.78", datum,
..."12345678")
```

Das Komma in der ersten Zeile ist kein Fortsetzungskennzeichen, sondern ein Argumenttrennzeichen: es trennt die Argumente datum und "12345678" voneinander. Die Auslassungspunkte in der zweiten Zeile kennzeichnen den technischen Zeilenumbruch im Unterschied zu einer Fortsetzungszeile:

```
translate("45.12.78", datum,,
  "12345678")
```

In diesem Fall wird die erste Zeile mit zwei Kommas beendet: dem Argumenttrennzeichen und dem Fortsetzungskennzeichen. Da die erste Zeile in der nächsten Zeile (im Sinne von REXX) fortgesetzt wird, wird sie nicht durch Auslassungspunkte eingeleitet.

Zeilen mit mehreren REXX-Anweisungen

In den Beispielen werden bisweilen mehrere REXX-Anweisungen (durch Semikolon getrennt) in einer Zeile zusammengefaßt. Dies hat rein technische Gründe. Wir tun dies aus Gründen der Platzersparnis, um Ihnen mehr Beispiele bieten zu können. In der Praxis sollte man in der Regel darauf verzichten, mehrere Anweisungen in einer Zeile zusammenzufassen, weil dadurch die Lesbarkeit und die Änderbarkeit eines Programms beeinträchtigt werden.

Schriftart von REXX-Anweisungen

Programmbeispiele werden in einer serifenlosen Schriftart dargestellt:

```
name = "Gisela"
say left (name, 4)
```

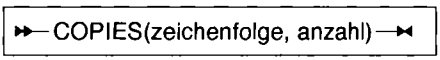
Einzelne Bestandteile von REXX-Anweisungen (z. B. Zeichenfolgen, Variablennamen, Schlüsselwörter, Funktionsnamen), Parameter, Syntaxdiagramme und formale Darstellungen von Anweisungen und Aufrufen werden ebenfalls in einer serifenlosen Schriftart dargestellt:

> die verschachtelte endgültige Lösung *(Kurzform)* besser verständlich wird. Mit bitxor (text1, text2) wird auf text1 und text2 die Operation des logischen Exklusiv-Oder angewandt, d. h. zwei gleiche Quellbits führen...

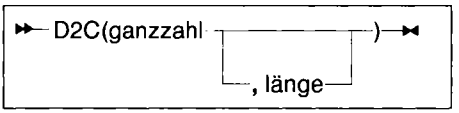
Syntaxdiagramme

Die Beschreibung von Anweisungen, Befehlen und Funktionen wird durch Syntaxdiagramme eingeleitet. Diese sind nach folgenden Regeln aufgebaut:

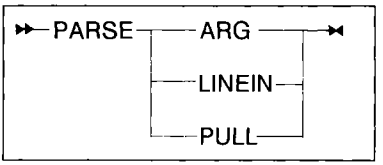
- Syntaxdiagramme werden von links nach rechts und von oben nach unten gelesen.
  - ▸▸— kennzeichnet den Anfang eines Diagramms (Startpfeil).
  - —▸ bedeutet, daß das Diagramm in der Folgezeile fortgesetzt wird.
  - ▸— setzt die vorige Zeile fort.
  - —▸◂ beendet ein Diagramm (Endepfeil).
- *Obligatorische Einträge* sind immer erforderlich und erscheinen im Hauptpfad des Diagramms (auf der waagerechten Linie zwischen Start- und Endepfeil):



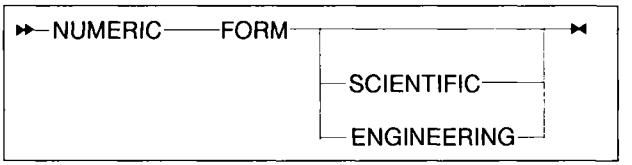
- *Optionale Einträge* können wahlweise angegeben werden und erscheinen unterhalb des Hauptpfads (im Beispiel: , länge):



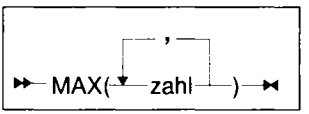
- Eine *obligatorische Auswahl* besteht aus mehreren Einträgen, von denen einer ausgewählt werden *muß*. Ein Eintrag erscheint im Hauptpfad und die übrigen Einträge erscheinen in Nebenpfaden (im Beispiel: ARG, LINEIN und PULL):



- Eine *optionale Auswahl* besteht aus mehreren Einträgen, von denen einer ausgewählt werden *kann*. Die wählbaren Einträge erscheinen in Nebenpfaden (im Beispiel: SCIENTIFIC und ENGINEERING):



- Ein *wiederholbarer Eintrag* wird durch einen rückläufigen Pfeil gekennzeichnet:



- *Schlüsselwörter* erscheinen in Großbuchstaben, können aber in Groß-/Kleinschreibung angegeben werden. *Variable Einträge* werden in Kleinbuchstaben dargestellt und müssen durch entsprechende Variablennamen oder Werte ersetzt werden.
- *Sonderzeichen* wie Satzzeichen, Klammern und Operatoren sind Bestandteil der Syntax und müssen angegeben werden.

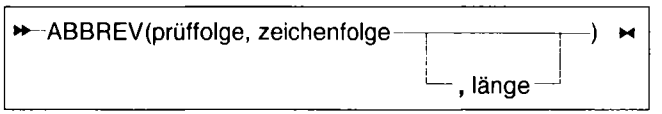
Syntaxmodelle

Der formale Aufbau von Anweisungen, Befehlen und Funktionen wird in Übersichtstabellen und im Text durch Syntaxmodelle dargestellt. Diese sind nach folgenden Regeln aufgebaut:

- Einträge, die nicht in eckigen oder geschweiften Klammern stehen, müssen angegeben werden:
  ```
  LENGTH(zeichenfolge)
  ```
- Einträge in eckigen Klammern [] sind optional und können weggelassen werden.
  ```
  LINES([datei])
  ```
- Mehrere durch | getrennte Einträge in geschweiften Klammern {} stellen eine Auswahl dar. Einer der Einträge muß ausgewählt werden:
  ```
  CALL {ON | OFF}
  ```
- Ein Eintrag, dem Auslassungspunkte folgen, kann wiederholt werden:
  ```
  CALL name [ausdruck,...]
  ```
- *Schlüsselwörter* erscheinen in Großbuchstaben, können aber in Groß-/Kleinschreibung angegeben werden. *Variable Einträge* werden in Kleinbuchstaben dargestellt und müssen durch entsprechende Variablennamen oder Werte ersetzt werden.
- *Sonderzeichen* wie Satzzeichen, Klammern und Operatoren sind Bestandteil der Syntax und müssen angegeben werden.

LEXIKON

abbrev



Die eingebaute SAA-Funktion abbrev (abbreviation = Abkürzung) liefert den logischen Wert 1 (wahr) zurück, wenn zeichenfolge mit den ersten n (n = länge) Zeichen der Zeichenfolge prüffolge übereinstimmt, andernfalls wird der logische Wert 0 (falsch) zurückgeliefert.

prüffolge und zeichenfolge sind beliebige Zeichenfolgen. Wenn länge fehlt, gilt die Länge von zeichenfolge, andernfalls die angegebene Länge länge. Die Funktion liefert 1 zurück,

- wenn länge gleich 0 ist,
- oder wenn länge fehlt und zeichenfolge die Nullzeichenfolge ist.

Die Funktion liefert 0 zurück,

- wenn länge größer als die Länge von zeichenfolge ist,
- wenn länge fehlt und die Länge von prüffolge größer als die Länge von zeichenfolge ist,
- wenn prüffolge die Nullzeichenfolge ist, es sei denn, zeichenfolge ist ebenfalls die Nullzeichenfolge.

länge	Nichtnegative Ganzzahl. Anzahl der Zeichen von prüffolge, die linksbündig mit zeichenfolge übereinstimmen sollen.
prüffolge	Beliebige Zeichenfolge, die zur Prüfung verwendet werden soll.
zeichenfolge	Beliebige Zeichenfolge, die geprüft werden soll.

Beispiele

REXX-Anweisungen	Erläuterung
befehl = "edit" eingabe = "ed" ok1 = abbrev(befehl, eingabe) ok2 = abbrev(befehl, eingabe, 3)	Abkürzung prüfen: Da bei ok1 die Länge fehlt, ist eingabe in der Länge von eingabe (2) mit befehl identisch, daher wird ok1 der Wert 1 zugewiesen. Bei ok2 werden 3 Zeichen verglichen. Da eingabe aber nur aus zwei Zeichen besteht, ergibt der Vergleich von "ed" mit "edi" ungleich und ok2 wird 0 zugewiesen.
befehl = "edit" eingabe = " " ok1 = abbrev(befehl, eingabe) ok2 = abbrev(befehl, eingabe, 1)	Zeichenfolge = Nullzeichenfolge: Da bei ok1 die Länge fehlt, ist die Nullzeichenfolge in eingabe in der Länge von eingabe (0) mit befehl identisch, daher wird ok1 der Wert 1 zugewiesen. Bei ok2 wird 1 Zeichen verglichen. Da eingabe aber nur aus der Nullzeichenfolge besteht, ergibt der Vergleich von " " mit "e" ungleich und ok2 wird 0 zugewiesen.

Anwendung

1	Abkürzungen tabellarisch auf Eindeutigkeit überprüfen
2	Abkürzungen selektiv auf Eindeutigkeit überprüfen
3	Hilfeoption abfragen

1. Abkürzungen tabellarisch auf Eindeutigkeit überprüfen

Aufgabe: Ein auf den Wortanfang verkürztes Wort, z. B. ein Befehl, ein Parameter, eine Funktion oder eine Option, soll in einer Wortliste identifiziert und durch das vollständige Wort ersetzt werden.

```
tab = "move copy browse move edit compare"
tab = preptab(tab)
lgtab = getlgtab(tab)
fun = "com"
name = checkfun(tab, lgtab, fun, "edit")
if name = "_" then say "Funktion "fun" ist falsch"
exit

preptab: procedure
  parse arg tab
  tab = wordsort(tab)
  do i = 1 to words(tab) - 1
    do while word(tab, i) = word(tab, i + 1) & i < words(tab)
      tab = delword(tab, i, 1)
    end
  end i
  return tab
```

```
➔ name: "COMPARE"
```

Die Wortliste tab definiert die Wörter, mit deren Hilfe die Prüfung vorgenommen werden soll. Mit der internen Funktion preptab bereiten wir diese Wortliste so vor, daß die Erstellung der Längentabelle mit der internen Funktion getlgtab einwandfrei möglich ist. Da es bei längeren Wortlisten mühselig und fehleranfällig wäre, die Wörter von Hand zu sortieren, sortieren wir zuerst tab mit der externen Funktion wordsort. Mehrfach vorkommende Wörter würden in getlgtab eine Sonderbehandlung erfordern, deswegen merzen wir evtl. Doubletten in der folgenden do-Schleife aus und geben die bereinigte Wortliste mit return tab an den Aufrufer zurück.

```
getlgtab: procedure
  parse arg tab
  lgtab = "_"
  lastpos = 0
  do i = 1 to words(tab)
    pos = compare(word(tab, i), word(tab, i + 1))
    if pos < lastpos then min = lastpos; else min = pos
    lgtab = lgtab min
    lastpos = pos
  end i
  return lgtab
```

Die interne Funktion getlgtab wird (sinngemäß) unter *compare, Anwendung,* beschrieben. Der Aufruf lgtab = getlgtab(tab) erstellt die Tabelle "1 3 3 1 1". Die i-te Zahl in lgtab bezeichnet die Anzahl der Zeichen, die mindestens angegeben werden müssen, damit das i-te Wort in tab eindeutig identifiziert werden kann.

```
checkfun: procedure
  parse upper arg tab, lgtab, fun, standard
  if fun = "" then fun = standard
  do i = 1 to words(tab)
    name = word(tab, i)
    lg = word(lgtab, i)
    if abbrev(name, fun, lg) then do
      fun = name; leave i
    end
  end i
  if i > words(tab) then fun = ""
  return fun
```

Der zu prüfende Wortanfang (die Abkürzung) ist in der Variablen fun enthalten ("com"). Mit dem Funktionsaufruf checkfun(tab, lgtab, fun, "edit") übergeben wir an die interne Funktion checkfun die Wortliste tab, die Längentabelle lgtab, die zu prüfende Abkürzung fun und den Standardwert "edit", der verwendet werden soll, wenn fun die Nullzeichenfolge ist.

Die interne Funktion checkfun beginnt mit dem Einlesen der Parameter mit parse upper arg. Sie werden in Großbuchstaben (upper) eingelesen, so daß unterschiedliche Schreibweisen keinen Einfluß auf die Prüfung haben. Die Parameter sollten allerdings keine Umlaute und ß enthalten, da diese nicht in Großbuchstaben umgewandelt werden. Auch das anstelle der Abkürzung zurückzuliefernde vollständige Wort wird daher immer in Großbuchstaben geliefert, so daß im weiteren Verlauf des Hauptprogramms das Wort name nur in Großbuchstaben abgefragt werden muß. Wenn fun die Nullzeichenfolge ist, wird fun der Standardwert standard zugewiesen.

In der do-Schleife weisen wir der Variablen name zuerst das i-te Wort aus tab zu. lg = word(lgtab, i) liefert die Mindestlänge, die erforderlich ist, um das i-te Wort in tab eindeutig zu identifizieren. Mit abbrev(name, fun, lg) prüfen wir, ob der Wortanfang fun mit mindestens lg Zeichen linksbündig mit name übereinstimmt. Wenn ja, wird die Abkürzung fun durch das vollständige Wort name ersetzt und die Schleife verlassen. Nach dem Verlassen der Schleife muß noch geprüft werden, ob überhaupt eine Übereinstimmung gefunden wurde. Wenn i größer als words(tab) ist, wurden alle Wörter von tab geprüft. Dann wurde noch einmal der Schleifenkopf ausgeführt, d. h. i wurde um 1 auf words(tab) + 1 erhöht und die Prüfung der to-Klausel to words(tab) führte zum Abbruch der Schleife. Wir geben daher mit say eine Fehlermeldung aus, wenn i größer als words(tab) ist. In unserem Beispiel enthält fun nach dem Verlassen der Schleife das komplette Fundwort "COMPARE". Dieses liefern wir mit return fun an das Hauptprogramm zurück, wo es dann der Variablen name zugewiesen wird.

2. Abkürzungen selektiv auf Eindeutigkeit überprüfen

Aufgabe: Es soll geprüft werden, ob ein eingegebenes Befehlskürzel eindeutig einem der Befehle browse, compare, copy, edit oder move zuzuordnen ist. Wenn nichts, also die Nullzeichenfolge eingegeben wurde, soll standardmäßig der Befehl "edit" angenommen werden.

```
kuerzel = "com"
kuerzel = translate(kuerzel)
select
  when abbrev("EDIT", kuerzel) then ok = 1
```

```
    when abbrev("BROWSE", kuerzel) then ok = 1
    when abbrev("COMPARE", kuerzel, 3) then ok = 1
    when abbrev("COPY", kuerzel, 3) then ok = 1
    when abbrev("MOVE", kuerzel) then ok = 1
    otherwise ok = 0
end
```

➔ ok: 1

Das Befehlskuerzel kuerzel wird zuerst mit translate in Großbuchstaben umgewandelt, damit unterschiedliche Schreibweisen keinen Einfluß auf die Prüfung haben. Die erste when-Klause in der select-Auswahl fragt den Standardbefehl "EDIT" ab. Wenn eine beliebige linksbündige Teilfolge von "EDIT" eingegeben wurde, liefert abbrev 1 (wahr). Dies trifft auch dann zu, wenn die Nullzeichenfolge eingegeben wurde. Da der Längenparameter in dem abbrev-Aufruf ausgelassen wurde, wird in der Länge des Prüflings kuerzel geprüft, und da dieser 0 Zeichen lang ist, wird immer Übereinstimmung festgestellt. Wir haben also mit der ersten when-Klausel gleichzeitig das Problem der Standardzuweisung gelöst. Dies funktioniert aber nur dann so, wenn der Standardbefehl in der *ersten* when-Klausel abgefragt wird und wenn keine Längenangabe notwendig ist, um Eindeutigkeit zu erzielen.

Die übrigen when-Klauseln fragen analog ab, außer daß die Nullzeichenfolge nicht mehr in Frage kommt. Bei den beiden mit "C" beginnenden Befehlen wird die Länge 3 im abbrev-Aufruf mit angegeben, weil diese beiden Befehle sich erst im dritten Zeichen eindeutig unterscheiden. Wenn eine der when-Klauseln zutrifft, wird ok auf 1 (wahr) gesetzt und die Auswahl verlassen, andernfalls wird ok im otherwise-Zweig der Wert 0 (falsch) zugewiesen.

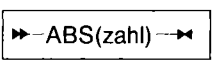
3. Hilfeoption abfragen

Aufgabe: In vielen Programmen wird die Übergabe *keines* Parameters oder eines Fragezeichens als Hilfeoption interpretiert, d. h. in diesem Fall zeigt das Programm eine Hilfe an.

```
parse arg parms
if abbrev('?', parms) then do...
```

Wenn beim Aufruf des Programms keine Parameter angegeben wurden, enthält parms nach parse arg die Nullzeichenfolge. In diesem Fall wird mit abbrev parms in der Länge 0 mit "?" verglichen. Dies ergibt 1 (wahr), wird also als Hilfeanforderung interpretiert. Wenn parms aus dem Fragezeichen besteht (und zwar nur aus dem Fragezeichen) ergibt der Vergleich von parms mit "?" Gleichheit und abbrev liefert ebenfalls 1. In allen anderen Fällen liefert abbrev 0 (falsch), d. h. es wurde keine Hilfe angefordert.

abs



Die eingebaute SAA-Funktion abs (Absolutwert) liefert den absoluten Wert von zahl zurück. Die zurückgelieferte Zahl hat kein Vorzeichen und wird entsprechend dem aktuellen Wert von numeric digits formatiert.

zahl Beliebige Zahl.

Beispiele

REXX-Anweisungen	Erläuterung
abs1 = abs(100.25) abs2 = abs(-100.25)	Absolutwert: abs1: 100.25, abs2: 100.25.
REXX-Anweisungen	**Erläuterung**
numeric digits 3 z1 = 1234 z2 = "□□-□1234□□" abs1 = abs(z1) abs2 = abs(z2)	numeric digits wird berücksichtigt: Der Absolutwert von z1 und z2 ist in beiden Fällen 1234. Dieser Wert wird nach numeric digits formatiert, d. h. auf 3 Stellen gerundet. Die Funktion liefert daher für abs1 und abs2 jeweils 1.23E+3 zurück.

Anwendung

1	Uhrzeitdifferenz in Sekunden berechnen
2	Dezimalformat in gepacktes Dezimalformat umwandeln

1. Uhrzeitdifferenz in Sekunden berechnen

Aufgabe: Es soll die Differenz in Sekunden zwischen zwei Uhrzeiten, die in der Form hh:mm:ss vorliegen, berechnet werden. Dabei soll es keine Rolle spielen, in welcher Reihenfolge die beiden Uhrzeiten angegeben werden. Die Differenz zwischen 08:00:00 und 16:00:00 soll also z. B. die gleiche sein wie zwischen 16:00:00 und 08:00:00.

```
zeit1 = "12:24:36"
zeit2 = "08:07:55"
diff1 = timediff(zeit1, zeit2)
diff2 = timediff(zeit2, zeit1)
exit

timediff: procedure
  parse arg h1 ":" m1 ":" s1, h2 ":" m2 ":"  s2
  zeit1 = ((h1 * 60) + m1) * 60 + s1
  zeit2 = ((h2 * 60) + m2) * 60 + s2
  return abs(zeit1 - zeit2)
```

➔ diff1: 15401, diff2: 15401

Die procedure-Anweisung der Funktion timediff schirmt die Variablen des Aufrufers und der Funktion gegeneinander ab. Mit parse arg werden die beiden durch Komma getrennten Argumentfolgen (die beiden Uhrzeiten) eingelesen. Dabei werden sie anhand des Trennzeichens ":" bereits in ihre Bestandteile, nämlich Stunden (h1, h2), Minuten (m1, m2) und Sekunden (s1, s2) zerlegt. Die beiden Uhrzeiten werden in Sekunden umgerechnet. Die absolute Differenz der beiden Sekundendifferenzen wird mit abs(zeit1 - zeit2) berechnet und an den Aufrufer zurückgegeben.

2. Dezimalformat in gepacktes Dezimalformat umwandeln

Aufgabe: Ganzzahlige Dezimalzahlen sollen in gepackte Dezimalzahlen umgewandelt werden, ein Datenformat, das z. B. auf Großrechnern benutzt wird. Eine Dezimalzahl ist gepackt, wenn die einzelnen Dezimalziffern in je einem Halbbyte gespeichert sind, und wenn das rechteste Halbbyte das Vorzeichen enthält. Dabei entspricht "c"x einem Plus, einem Minus entsprechen "b"x oder "d"x (beide Vorzeichendarstellungen sind gleichwertig).

```
    dezimalzahl = -1234
    packzahl = d2p(dezimalzahl)
    exit
  d2p: procedure
    parse arg zahl
    if zahl < 0 then vorzeichen = "D"; else vorzeichen = "C"
    numeric digits length(zahl)
    pos = pos(".", reverse(zahl))
    if pos > 0 then zahl = zahl * (10 ** (pos - 1)) + 0
    zahl = zahl / 1
    return x2c(abs(zahl + 0) || vorzeichen)
➔   packzahl: "01234D"x
```

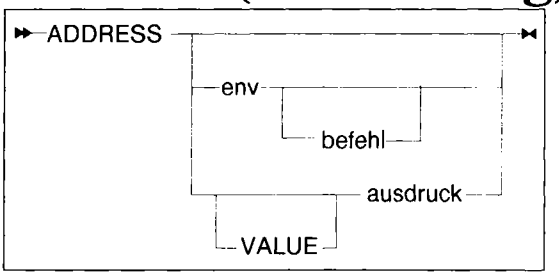
Die interne Funktion d2p wandelt dezimalzahl in eine gepackte Dezimalzahl um. d2p beginnt mit einer procedure-Anweisung, so daß die Variablen des aufrufenden Programms und der Funktion verschiedenen Geltungsbereichen angehören. Mit parse arg wird die umzuwandelnde Zahl in die Variable zahl eingelesen.

Wenn zahl negativ ist, wird das Vorzeichen mit "D", andernfalls mit "C" besetzt. Die Anzahl der maximalen Ziffernstellen wird vorsichtshalber auf die Länge von zahl gesetzt für den Fall, daß diese länger ist als der aktuelle Wert von numeric digits. Mit reverse(zahl) kehren wir zahl um und suchen mit pos(".", ...) die Position des Dezimalpunktes. Wenn pos größer als 0 ist, repräsentiert die um 1 verminderte Position pos - 1 die Anzahl der Nachkommastellen. Wir multiplizieren in diesem Fall zahl mit der Zehnerpotenz 10 ** (pos - 1), so daß zahl zu einer Ganzzahl wird. Die scheinbar überflüssige Division von zahl durch 1 (zahl/1) löscht die Nachkommastellen, wenn diese 0 sind.

Bei der Berechnung der gepackten Zahl werden zuerst durch die Addition zahl + 0 führende Nullen von zahl abgestrippt. Die Funktion abs entfernt ggf. das Vorzeichen von der Zahl. Der so entstandenen Dezimalzahl 4567 wird noch das Vorzeichen angehängt, das bereits vorher zugewiesen wurde. Die sich ergebende Zeichenfolge 4567D wird mit der Funktion x2c in eine Bytezeichenfolge mit einer geraden Anzahl von Stellen umgewandelt ("04567D") und mit return an den Aufrufer zurückgegeben.

Hinweis: Wir haben hier aus Gründen der Übersichtlichkeit nicht den Fall berücksichtigt, daß eine Zahl in exponentieller Schreibweise vorliegen kann.

address (Anweisung)



Siehe auch: address (Funktion); Adreßumgebung; Befehl.

Die Anweisung address (Adresse) stellt die aktuelle Adreßumgebung temporär oder dauerhaft ein oder schaltet zwischen zwei Adreßumgebungen um.

Standardadreßumgebung. Beim Eintritt in ein Programm oder ein internes oder externes Unterprogramm gilt automatisch die Standardadreßumgebung. Mit

```
ADDRESS ""
```

kann diese Adreßumgebung nach einer Änderung wieder eingestellt werden.

Temporäre Adreßumgebung. Wenn die Adreßumgebung vorübergehend, d. h. nur für die Ausführung eines Befehls, geändert werden soll, muß die address-Anweisung in der folgenden Variante verwendet werden:

```
ADDRESS env befehl
```

Dabei muß env eine Zeichenfolge sein, die eine gültige Adreßumgebung darstellt, befehl ein Ausdruck, der nach der Auswertung einen gültigen Befehl für die angegebene Adreßumgebung ergibt. Nach der Ausführung des Befehls wird wieder zur vorherigen Adreßumgebung zurückgeschaltet.

Permanente Adreßumgebung. Wenn die Adreßumgebung permanent umgestellt wird, werden alle folgenden Befehle an diese Adreßumgebung gesendet, bis die Adreßumgebung wieder umgestellt wird. Der Name der vorhergehenden Adreßumgebung wird gesichert. Zur permanenten Umstellung der Adreßumgebung gibt es drei Möglichkeiten:

- Die Schlüsselwort address steht zusammen mit dem Namen der Adreßumgebung env allein in einer Zeile:

  ```
  ADDRESS env
  ```

- Die Schlüsselwort address steht zusammen mit dem Unterschlüsselwort value und ausdruck in einer Zeile:

  ```
  ADDRESS VALUE ausdruck
  ```

  ausdruck kann eine Variable oder ein zusammengesetzter Ausdruck sein und muß nach seiner Auswertung den Namen einer gültigen Adreßumgebung ergeben.

- Wenn ausdruck nicht mit einem Symbol oder einer literalen Zeichenfolge beginnt, d. h. wenn ausdruck mit einem speziellen Zeichen wie einem Operator oder einer Klammer beginnt, kann das Unterschlüsselwort value weggelassen werden:

  ```
  ADDRESS ausdruck
  ```

Umschaltung. Wenn address ohne Argumente angegeben wird, wird die letzte gesicherte, permanente Adreßumgebung wieder als permanente Adreßumgebung hergestellt. Dabei wird wiederum die aktuelle Adreßumgebung gesichert. Die mehrfache Anwendung der address-Anweisung ohne Argumente bewirkt daher das Umschalten zwischen zwei Adreßumgebungen.

Geltungsbereich. Die aktuelle Adreßumgebung gilt lokal für ein Programm oder ein internes oder externes Unterprogramm, d. h. beim Eintritt in ein Programm gilt die Standardadreßumgebung und beim Verlassen eines Programms wird die vorherige aktuelle Adreßumgebung des Aufrufers wiederhergestellt.

Hinweis: Die eingebaute SAA-Funktion address liefert den Namen der aktuellen Adreßumgebung.

ausdruck — Ausdruck, der den Namen einer Adreßumgebung ergibt.
- Wenn ausdruck das Unterschlüsselwort VALUE vorausgeht, ist jeder beliebige Ausdruck erlaubt.
- Andernfalls darf ausdruck nicht mit einem Symbol oder einer konstanten Zeichenfolge beginnen, d. h. ausdruck muß mit einem speziellen Zeichen wie einem Operator oder einer Klammer beginnen.

befehl — *Siehe* Befehl.

env — Zeichenfolge, die den Namen einer gültigen Adreßumgebung ergibt.

VALUE — Es folgt ein Ausdruck, dessen Auswertung den Namen einer Adreßumgebung ergibt.

Beispiele

REXX-Anweisungen	Erläuterung
	Aktuelle Adreßumgebung: "tso".
address isredit "macro"	Der Befehl "macro" wird an die temporäre Adreßumgebung "isredit" gesendet, anschließend gilt wieder die aktuelle Adreßumgebung "tso".
address isredit "macro" "(dsn) = dataset"	Die aktuelle Adreßumgebung "tso" wird verlassen und dauerhaft auf die Adreßumgebung "isredit" umgeschaltet. Die folgenden Befehle "macro" und "(dsn) = dataset" (und alle weiteren) werden daher an die Adreßumgebung "isredit" übergeben.
parse source system if system = "OS/2" ...then env = "cmd"; ...else env = "command" address value env address (env)	Je nach zugrundeliegendem Betriebssystem wird der Variablen env der Name der Adreßumgebung der OS/2-Befehlsschnittstelle oder der DOS-Befehlsschnittstelle zugewiesen. Das Unterschlüsselwort value substituiert für die Variable env den Namen der jeweiligen Adreßumgebung. Auf diese Adreßumgebung wird permanent umgeschaltet. address (env) bewirkt das gleiche, ohne daß das Unterschlüsselwort value verwendet werden muß, weil die einleitende Klammer klarmacht, daß es sich nicht um den literalen Namen einer Adreßumgebung handelt.
address isredit address address address ""	Die erste address-Anweisung schaltet dauerhaft auf die Adreßumgebung "isredit" um. Die zweite schaltet zur vorherigen Adreßumgebung, das ist in diesem Fall die Standardadreßumgebung "tso" zurück. Die dritte address-Anweisung wiederum schaltet zu der vorherigen Adreßumgebung "isredit" zurück. Schließlich schaltet address "" auf die Standardadreßumgebung "tso" zurück, gleichgültig welche Adreßumgebung vorher aktiv war.

Anwendung

Datei mit APPC senden

Aufgabe: Mit APPC können Daten zwischen verschiedenen Plattformen, z. B. OS/2, OS/400 und MVS/TSO, gesendet und empfangen werden. Die auf den genannten drei Plattformen verfügbare Adreßumgebung CPICOMM unterstützt die CPI-Schnittstelle, eine Sammlung von einheitlichen Befehlen zur Ausfüh-

rung von APPC-Operationen. Dies bedeutet, daß ein einziges REXX-Programm auf allen drei Plattformen zum Einsatz kommen kann, da die REXX-Sprache und die CPI-Befehle auf allen Plattformen gleich sind. Allerdings müssen dabei Betriebssystemeigenarten berücksichtigt werden, insbesondere solche, die sich auf die Ein-/Ausgabe beziehen.

Es soll ein Programm erstellt werden, das eine beliebige Datei unter OS/2 oder TSO einliest und diese Datei an einen beliebigen Empfänger sendet.

APPC: Advanced Program to Program Communication = Erweiterte Kommunikation zwischen Programmen. CPI: Common Programming Interface = Allgemeine Programmierschnittstelle.

```
parse arg destname, datei
parse source system
address cpicomm
if system = "OS/2" then do
  address cmd "@cpicrexx 1>nul 2>nul"
  if rc <> 30 then call fehler "Fehler "rc" beim Laden von CPICREXX"
end
```

Zuerst werden die übergebenen Parameter mit parse arg eingelesen: der *destination name* (Name des Ziels) destname, der einen Eintrag in einer Datei mit Angaben über den Empfänger bezeichnet, und datei, der Name der zu sendenden Datei. Mit parse source verschaffen wir uns den Namen des Betriebssystems, den wir benötigen, um die betriebssystembedingten Unterschiede zwischen OS/2 und TSO berücksichtigen zu können. Mit address cpicomm stellen wir die Adreßumgebung permanent auf die CPI-Schnittstelle ein.

Wenn wir unter OS/2 arbeiten, müssen wir noch eine Besonderheit beachten: Das CPI-Schnittstellenprogramm für REXX, CPICREXX, muß geladen werden. Die Anweisung address cmd mit folgendem OS/2-Befehl bewirkt die temporäre Umschaltung der aktuellen Adreßumgebung CPICOMM auf CMD, so daß der Befehl "@cpicrexx 1>nul 2>nul" der OS/2-Befehlszeile zur Ausführung übergeben wird. "@" bewirkt, daß keine Bildschirmausgaben erfolgen, und die Umleitungen "1>nul 2>nul" bewirken, daß die Zugriffsnummern (filehandles) 1 und 2 für die Standardausgabe und die Standardfehlerausgabe auf das Nullgerät umgelenkt werden, so daß auch hier keine Ausgaben auf den Bildschirm erfolgen. Den Rückkehrcode rc = 30 ignorieren wir, da er bedeutet, daß CPICREXX schon geladen ist. Alle anderen Rückkehrcodes (auch 0) werden in dem internen Unterprogramm fehler abgehandelt, das wir weiter unten besprechen.

```
"cminit convid destname cpirc"; call fehler
"cmallc convid cpirc"; call fehler
```

Die beiden Befehle richten sich an die aktuelle Adreßumgebung, die nach wie vor CPICOMM heißt, da sie (unter OS/2) nur temporär geändert wurde. Der CPI-Befehl cminit dient der Initialisierung der Konversation und übergibt als Parameter die beiden Ausgabeparameter convid und cpirc sowie den Eingabeparameter destname, den wir bereits am Anfang des Programms mit dem übergebenen Argument bestückt haben. convid wird von der CPI-Schnittstelle mit der zugewiesenen KonversationsID versorgt und cpirc enthält nach Abarbeitung des Befehls den CPI-Rückkehrcode, den wir mit call fehler auswerten. Der Befehl cmallc dient zur Allokierung der Konversation und hat convid als Eingabeparameter und, wie alle CPI-Befehle, cpirc als Ausgabeparameter.

```
select
  when system = "OS/2" then do
    do while lines(datei) > 0; queue linein(datei); end
  end
  when system = "TSO" then do
    address tso
    "alloc dd(eingabe) da('datei') shr"; call fehler
    "execio * diskr eingabe (finis)"; call fehler
    "free dd(eingabe) da('datei') shr"; call fehler
    address
  end
  otherwise call fehler "Betriebssystem "system" nicht vorgesehen"
end
```

In der select-Auswahl lesen wir die Sendedatei ein. Diesen Vorgang selektieren wir je nach Betriebssystem. Unter OS/2 verwenden wir die Funktion lines, um abzufragen, ob noch Eingabesätze vorhanden sind, wenn nicht, wird die do-Schleife beendet. Der eingelesenen Sätze linein(datei) werden mit queue nacheinander in die Warteschlange gestellt. Die Fehlerbehandlung der linein-Funktion vernachlässigen wir hier, weil sie die Systematik unserer einfachen Fehlerroutine sprengen würde.

Unter TSO stellen wir die Adreßumgebung permanent auf TSO um, da wir drei Befehle an TSO übergeben wollen. Der TSO-Befehl alloc allokiert die Eingabedatei. Beachten Sie, daß datei außerhalb der Anführungszeichen steht, weil wir dem TSO-Befehl den Inhalt der Variablen datei übergeben wollen und nicht den Namen. Dies ist bei den CPI-Befehlen grundsätzlich anders. Dort werden immer nur die Namen von Variablen übergeben. Mit execio lesen wir die komplette Eingabedatei auf einmal in die Warteschlange ein und mit free geben wir die allokierte Datei wieder frei. Die folgende address-Anweisung ohne Argumente stellt wieder die zuvor gesicherte, vorhergehende Adreßumgebung her, so daß sich die Folgebefehle wieder an CPICOMM richten.

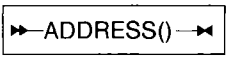
```
do queued()
  parse pull satz
  laenge = length(satz)
  "cmsend convid satz laenge request_to_send_received cpirc"
  call fehler
end
"cmdeal convid cpirc"; call fehler
exit
```

Die do-Schleife führt den Schleifenkörper so oft aus, wie die Funktion queued angibt, also entsprechend der Anzahl der in die Warteschlange eingelesenen Sätze. Neben dem bereits gefüllten Eingabeparameter convid sind noch die beiden Eingabeparameter satz und laenge zu versorgen. Wir tun das, indem wir mit parse pull einen Satz aus der Warteschlange entnehmen und der Variablen laenge die Länge length(satz) von satz zuweisen. Im nachfolgenden Aufruf von cmsend können wir nämlich length(satz) nicht direkt angeben, da die CPI-Schnittstelle in einem Befehl nur Variablennamen zuläßt. Nach diesen Vorarbeiten wird satz mit dem CPI-Befehl cmsend an den Partner gesendet. Die Bedeutung des Ausgabeparameters request_to_send_received behandeln wir hier nicht. Wenn alle Sätze gesendet sind, beenden wir die Konversation mit dem CPI-Befehl cmdeal.

```
fehler:
  parse arg msg
  if msg = "" then do
    if rc <> 0 then msg = "RC "rc" bei dem Befehl: "cmd
    if cpirc <> 0 then msg = "CPI-RC "cpirc" bei dem Befehl: "cmd
  end
  if msg = "" then return
  say msg
  exit
```

Zu Beginn der einfachen Fehlerroutine lesen wir mit parse arg die optionale Fehlermeldung msg ein. Wenn msg nicht die Nullzeichenfolge ist, ist ein spezieller Fehler aufgetreten. Andernfalls fragen wir, ob der Rückkehrcode rc ungleich 0 ist. Wenn ja, konnte der Befehl wegen formaler Mängel nicht an die CPI-Schnittstelle übergeben werden und wir bilden eine entsprechende Fehlernachricht, die wir msg zuweisen. Wenn der CPI-Rückkehrcode cpirc ungleich 0 ist, ist bei der Ausführung der CPI-Operation ein Fehler aufgetreten und wir definieren msg entsprechend. Wenn jetzt msg gleich der Nullzeichenfolge ist, verlassen wir das Unterprogramm mit return und kehren zum Aufrufer zurück. Andernfalls geben wir die Fehlermeldung mit say aus und brechen das Programm mit exit ab.

address (Funktion)



Siehe auch: address (Anweisung); Adreßumgebung; Befehl.

Die eingebaute SAA-Funktion address (Adresse) liefert den Namen der aktiven Adreßumgebung zurück, also den Namen der Adreßumgebung, an die momentan Befehle zur Ausführung übergeben werden. Leerstellen am Ende des Namens werden entfernt.

Hinweis: Mit der address-Anweisung kann eine Adreßumgebung temporär oder dauerhaft eingestellt oder zwischen zwei Adreßumgebungen umgeschaltet werden.

Beispiele

REXX-Anweisungen	Erläuterung
address cpicomm a1 = address() address command a2 = address()	a1 liefert z. B. "CPICOMM", a2 etwa "COMMAND".

Adreßumgebung

Siehe auch: Befehl.

Eine Adreßumgebung ist ein Prozeß in einem Adreßraum außerhalb des REXX-Interpreters, an die von einem REXX-Programm aus Befehle zur Ausführung gesendet werden können. Befehle unterscheiden sich von REXX-Anweisungen dadurch, daß diese vom REXX-Interpreter ausgeführt werden, während Befehle an eine Adreßumgebung zur Ausführung übergeben werden.

Nach Beendigung eines Befehls enthält die spezielle Variable rc den Rückkehrcode des Befehls.

Adreßumgebung der Befehlsschnittstelle. Jede REXX-Implementierung verfügt über mindestens eine aktive Adreßumgebung. In der Regel ist dies die Befehlsschnittstelle des zugrundeliegenden Betriebssystems. Die Befehlsschnittstelle wird auch zur Standardadreßumgebung, die beim Eintritt in ein REXX-Programm gilt, wenn ein REXX-Programm von dieser aus aufgerufen wird. Die folgende Tabelle zeigt die Adreßumgebungen, die für die Befehlsschnittstelle der unterstützten Betriebssysteme gelten.

Adreß-umgebung	System	Befehlsschnittstelle
CMD	OS/2	OS/2-Befehlszeile
COMMAND	DOS	DOS-Befehlszeile
MVS	MVS	Ausführung ausgewählter TSO/E REXX-Befehle und Aufruf von REXX-Programmen. Steht auch unter ISPF und TSO/E zur Verfügung.
TSO	TSO/E	TSO/E-READY-Modus; Ausführung der externen TSO/E-Funktionen und der TSO/E REXX-Befehle. Auch Standardadreßraum unter ISPF.

Adreßumgebungen. Die gültigen Adreßumgebungen sind unter dem zugrundeliegenden Betriebssystem definiert. REXX kann als Makro- oder Skriptsprache eines Anwendungsprogramms verwendet werden, wenn dieses von REXX übergebene Unterbefehle ausführt. Die folgende Tabelle führt die bekanntesten Adreßumgebungen auf.

Adreßraum	System	Funktion
ATTACH ATTACHMVS ATTACHPGM	ISPF, MVS, TSO/E	Aufruf von Programmen auf einem unterschiedlichen Task-Level (die Unterschiede bestehen in der Anzahl der Parameter und der Art der Parameterübergabe)
CONSOLE	ISPF, TSO/E	Ausführung von MVS-Befehlen und MVS-Unterbefehlen
CPICOMM	ISPF, MVS, OS/2, TSO/E	CPI-Schnittstelle für APPC-Befehle
EPM	OS/2	Makroschnittstelle zum Editor EPM
ISPEXEC	ISPF	Ausführung von ISPF-Befehlen und ISPF-Services
ISREDIT	ISPF	Makroschnittstelle zum Editor EDIT
LINK LINKMVS LINKPGM	ISPF, MVS, TSO/E	Aufruf von Programmen auf dem gleichen Task-Level (die Unterschiede bestehen in der Anzahl der Parameter und der Art der Parameterübergabe)
LU62	ISPF, MVS, TSO/E	Schnittstelle für APPC/MVS-Befehle

Standardadreßumgebung. Die Standardadreßumgebung, die beim Eintritt in ein REXX-Programm gilt, hängt vom Aufrufer ab. Wenn ein Programm von der Befehlsschnittstelle des Betriebssystems aus aufgerufen wird, ist die Befehlsschnittstelle die Standardadreßumgebung. Wenn ein Programm von einem

Editor aus aufgerufen wird, der Unterbefehle vom REXX-Interpreter akzeptiert, ist der Editor die Standardadreßumgebung.

Anweisungen und Funktionen. Mit der address-Anweisung kann eine Adreßumgebung temporär oder dauerhaft eingestellt oder zwischen zwei Adreßumgebungen umgeschaltet werden. Die eingebaute SAA-Funktion address liefert den Namen der aktuellen Adreßumgebung. Der TSO/E REXX-Befehl subcom prüft, ob eine Adreßumgebung verfügbar ist.

Anweisung

Siehe auch: Anweisungen - Übersicht; Klausel; Zuweisung.

Anweisungen bestehen aus ein oder mehreren Klauseln und bezeichnen eine Operation, die vom REXX-Interpreter ausgeführt wird. Wir unterscheiden zwischen Zuweisungen und Anweisungen im engeren Sinn, den Schlüsselwortanweisungen. Hier werden nur Schlüsselwortanweisungen behandelt.

Schlüsselwörter. Anweisungen sind Klauseln, die mit einem Schlüsselwort beginnen, das die Anweisung identifiziert. Die Anweisung parse arg parms beginnt z. B. mit dem Schlüsselwort parse. Das Schlüsselwort wird aber nur erkannt, wenn ihm nicht ein Gleichheitszeichen (Zuweisung) oder ein Doppelpunkt (Marke) folgt. Die beiden folgenden Klauseln werden daher nicht als die Schlüsselwortanweisungen parse bzw. drop erkannt, sondern als Zuweisung bzw. als Marke:

```
parse = 12
drop:
```

Eine Anweisung, die mit einem Schlüsselwort beginnt, kann nicht als Befehl erkannt werden, auch wenn dies wie in dem folgenden Fall so gewollt ist:

```
  call upro "e", "tel.lst"
  ...
upro: arg(1) arg(2)
```

Offenbar soll in dem internen Unterprogramm ein Befehl zusammengesetzt werden, der aus der ersten und zweiten Argumentfolge besteht, die an das Unterprogramm übergeben wurden: "e tel.lst" (die Datei tel.lst soll editiert werden). Da der REXX-Interpreter jedoch arg als Schlüsselwort der Anweisung arg und nicht als Aufruf der Funktion arg erkennt, wird (1) für ein variables Zeichenmuster angesehen, und da in diesem Fall in den Klammern ein Symbol stehen müßte, meldet REXX einen Fehler.

Unterschlüsselwörter. Eine Anweisung kann weitere Schlüsselwörter enthalten, die die Anweisung näher qualifizieren. Die Anweisung do i = 1 to 15 by 2... enthält z. B. die beiden Unterschlüsselwörter to und by, die den Endwert bzw. die Schrittweite der Kontrollvariablen i definieren. Die Unterschlüsselwörter else, end, otherwise, then und when haben faktisch die gleiche Funktion wie Klauseln und werden daher wie Schlüsselwörter behandelt, obwohl sie keine Anweisungen oder Klauseln einleiten.

Reservierte Symbole. Schlüsselwörter und Unterschlüsselwörter gelten innerhalb eines Programms bzw. einer Anweisung als reservierte Symbole, wenn sie an einer bestimmten Stelle erwartet werden oder vorkommen können. Ansonsten können Schlüsselwörter auch als Symbole verwendet werden. Es empfiehlt

sich jedoch, Schlüsselwörter grundsätzlich als reservierte Symbole zu betrachten, um Mißverständnissen und Fehlern vorzubeugen.

Groß-/Kleinschreibung. Schlüsselwörter und Unterschlüsselwörter können in Groß-/Kleinschreibung angegeben werden. In den Syntaxdiagrammen und in anderen formalen Darstellungen werden Schlüsselwörter zur Unterscheidung von anderen Elementen einer Anweisung jedoch immer in Großbuchstaben gezeigt.

Arten von Anweisungen. Wir unterscheiden die folgenden Arten von Anweisungen:

- **Steueranweisungen.** Sie beeinflussen durch logische Entscheidungen oder Verzweigungen den Kontrollfluß in einem Programm. Es gibt die folgenden Steueranweisungen:

  call, do, exit, if, iterate, leave, return, select, signal.

  Die Anweisungen do, if und select enthalten selbst wiederum Klauseln und werden daher als *komplexe Anweisungen* bezeichnet.
- **Operative Anweisungen.** Sie führen bestimmte Operationen aus. Darunter fällt z. B. die Einstellung der Adreßumgebung mit address, die Zerlegung von Eingabezeichenfolgen mit parse, die Manipulation von Warteschlangen mit parse, pull, push und queue usw. Es gibt die folgenden operativen Anweisungen:

  address, arg, drop, interpret, nop, numeric, options, parse, procedure, pull, push, queue, say, trace, upper (nur unter REXX TSO/E Level 1).

Anweisungen - Übersicht

Aufruf	Wirkung
ADDRESS {env [befehl] | [VALUE] ausdruck}	stellt die aktuelle Adreßumgebung dauerhaft oder temporär ein
ARG schablone	wandelt die übergebenen Argumente in Großbuchstaben um und zerlegt sie
CALL name [ausdruck,...]	ruft ein Unterprogramm auf und übergibt an dieses ggf. ein oder mehrere Argumentfolgen
CALL {ON | OFF} bedingung [NAME trapname]	setzt eine Fehlerbedingung oder setzt sie zurück und verzweigt im Fehlerfall mit call zur Fehlerroutine
DO [*wiederholung*] [{WHILE | UNTIL} bedingung] [anweisung...] END [name] *wiederholung:* {name = start [TO ende] [BY diff] [FOR anzahl] | FOREVER | anzahl}	gruppiert Anweisungen und führt diese einmal, wiederholt und/oder bedingungsabhängig aus
DROP {name | liste}...	hebt die Definition von Variablen auf
EXIT [ausdruck]	beendet ein Programm (Unterprogramm, Funktion) und gibt ggf. eine Zeichenfolge zurück

Aufruf	Wirkung
IF bedingung [;] THEN [;] anweisung [ELSE anweisung]	führt Anweisungen bedingungsabhängig aus
INTERPRET [ausdruck]	Zeichenfolge als Programmzeile interpretieren und ausführen
ITERATE [name]	do-Schleife mit der nächsten Wiederholung fortsetzen
LEAVE [name]	do-Schleife sofort beenden
NOP	Pseudo-Anweisung ohne Wirkung
NUMERIC DIGITS [ausdruck]	definiert die maximale Anzahl Ziffernstellen
NUMERIC FORM {SCIENTIFIC | ENGINEERING | [VALUE] ausdruck}	definiert das Format der exponentiellen Schreibweise
NUMERIC FUZZ [ausdruck]	definiert die numerische Unschärfe
OPTIONS...	setzt Optionen für DBCS-Zeichenfolgen
PARSE [UPPER] ARG [schablone]	zerlegt die übergebenen Argumente
PARSE [UPPER] LINEIN [schablone]	zerlegt die nächste Zeile des Standardeingabestroms (1)
PARSE [UPPER] PULL [schablone]	zerlegt die nächste Zeile der externen Warteschlange bzw. des Standardeingabestroms
PARSE [UPPER] SOURCE [schablone]	liefert Angaben über das ausgeführte Programm
PARSE [UPPER] VALUE [ausdruck] WITH [schablone]	zerlegt einen Ausdruck
PARSE [UPPER] VAR variable [schablone]	zerlegt eine Variable
PARSE [UPPER] VERSION [schablone]	liefert Angaben über die REXX-Version
PROCEDURE [EXPOSE {name | liste}...]	schützt die Variablen des Aufrufers und eines Unterprogramms gegeneinander und exponiert ggf. bestimmte Variablen
PULL [schablone]	liefert die nächste Zeile der externen Warteschlange bzw. des Standardeingabestroms in Großbuchstaben
PUSH [ausdruck]	fügt eine Zeile am Anfang der externen Warteschlange ein
QUEUE [ausdruck]	fügt eine Zeile am Ende der externen Warteschlange ein
RETURN [ausdruck]	beendet ein Programm (Unterprogramm, Funktion) und gibt ggf. eine Zeichenfolge zurück
SAY [ausdruck]	gibt eine Zeile in den Standardausgabestrom aus
SELECT {WHEN bedingung [;] THEN [;] anweisung}... [OTHERWISE [;] [anweisung...]] END	führt bedingungsabhängig eine von mehreren alternativen Anweisungen aus

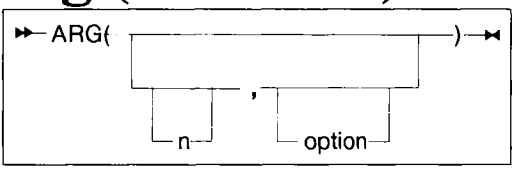
Aufruf	Wirkung
SIGNAL {marke I [VALUE] ausdruck}	verzweigt zu einer Marke
SIGNAL {ON I OFF} bedingung [NAME trapname]	setzt eine Fehlerbedingung oder setzt sie zurück und verzweigt zur Fehlerroutine mit signal
TRACE {[präfix] [option] I zahl I zeichenfolge I [VALUE] ausdruck}	schaltet den Trace ein oder aus und bestimmt den Umfang der Trace-Ausgabe
UPPER name...	wandelt ein oder mehrere Variablen in Großbuchstaben um (2)

(1) erst ab REXX Level 2
(2) nur unter REXX TSO/E Level 1

arg (Anweisung)

Die arg-Anweisung ist identisch mit der Anweisung parse upper arg. *Siehe:* parse arg.

arg (Funktion)



Siehe: Argumentfolge.

Die eingebaute SAA-Funktion arg (Argument) liefert eine Argumentzeichenfolge (kurz: Argumentfolge) oder Angaben über die Argumentfolgen zurück.

Wenn n fehlt, wird die Anzahl der übergebenen Argumentfolgen zurückgeliefert. Die Anzahl der übergebenen Argumentfolgen ist mit der Nummer der letzten ausdrücklich übergebenen Argumentfolge identisch, stimmt also mit der letzten Argumentfolge überein, die für arg(n, "exists") den Wert 1 zurückliefert (*siehe* unten). Wenn keine Argumentfolgen übergeben werden, liefert arg() den Wert 0.

Wenn nur n angegeben wird, wird die n-te Argumentfolge zurückgeliefert. Wenn die n-te Argumentfolge nicht vorhanden ist, wird die Nullzeichenfolge zurückgeliefert. n muß eine positive Ganzzahl sein.

option ist ein beliebiger Ausdruck, der eine der nachstehenden Optionen ergeben muß. Von der Option wird nur das erste Zeichen ausgewertet, Groß- und Kleinbuchstaben sind gleichwertig.

- E (exists = vorhanden). Liefert 1 (wahr), wenn die n-te Argumentfolge beim Aufruf des Programms ausdrücklich angegeben wurde, andernfalls 0 (falsch).
- O (omitted = weggelassen). Liefert 1 (wahr) wenn die n-te Argumentfolge beim Aufruf des Programms *nicht* ausdrücklich angegeben wurde, andernfalls 0 (falsch).

Hinweise:

- Mit der Anweisung parse arg wird die übergebene Zeichenfolge eines Programms in Argumentfolgen zerlegt. parse upper arg und die Anweisung arg leisten das gleiche, wandeln aber die Argumentfolgen in Großbuchstaben um.
- Obwohl das Implementierungslimit die Anzahl der Argumentfolgen in Programmaufrufen auf 20 beschränkt, können die Anweisungen parse arg bzw. arg beliebige Argumentfolgen zerlegen, d. h. diese können beliebig lang sein und aus beliebig vielen kommabegrenzten Argumenten bestehen.

n — Positive Ganzzahl. Nummer der Argumentfolge, die zurückgeliefert werden soll.

option — Nur das erste Zeichen wird ausgewertet, Groß- und Kleinbuchstaben sind gleichwertig.
Mögliche Optionen: *siehe oben.*

Beispiele

REXX-Anweisungen	Erläuterung
parm1 = "dies" parm3 = "jenes" rc = upro(parm1,, parm3) call upro parm1,, parm3 upro: a1 = arg() a2 = arg(1) a3 = arg(2) a4 = arg(1, "e") a5 = arg(2, "o") a6 = arg(2, "e") return "_"	Programmaufruf: Die beiden Aufrufe des internen Unterprogramms upro übergeben die gleichen Argumentfolgen: "dies",, "jenes". Die Zuweisungen a1-a6 erfolgen im aufgerufenen Programm upro. a1 liefert die Anzahl der übergebenen Argumentfolgen: 3. a2 liefert die erste Argumentfolge: "dies". a3 liefert die zweite Argumentfolge. Da diese im Aufruf aber ausgelassen wurde, wird die Nullzeichenfolge zurückgeliefert: "_". a4 liefert 1, da die erste Argumentfolge vorhanden ist. a5 liefert 1, da die zweite Argumentfolge ausgelassen wurde, und a6 liefert aus dem gleichen Grund 0.
parm1 = "dies" parm3 = "jenes" address cmd "upro□" ...parm1",,"parm3 upro: a1 = arg() a2 = arg(1) a3 = arg(2) a4 = arg(1, "e") a5 = arg(2, "o") a6 = arg(2, "e") return "_"	Befehlsaufruf: Das externe Unterprogramm upro wird unter OS/2 mit address cmd als Befehl aufgerufen. Dazu gleichwertig wäre der Aufruf von upro aus der OS/2-Befehlszeile: "upro dies,, jenes". Da es sich um einen Befehlsaufruf handelt, wird nur *eine* Argumentfolge übergeben: "dies,,jenes". Die Zuweisungen a1-a6 erfolgen im aufgerufenen Programm upro. a1 liefert daher als Anzahl der übergebenen Argumentfolgen 1. a2 liefert die erste und einzige Argumentfolge: "dies,,jenes". a3 liefert die zweite, nicht vorhandene Argumentfolge, also die Nullzeichenfolge: "_". a4 liefert 1, da die erste Argumentfolge vorhanden ist. a5 liefert 1, da keine zweite Argumentfolge vorhanden ist, und a6 liefert aus dem gleichen Grund 0.

Anwendung

1	Parameter in eine Meldung einsetzen
2	Hexadezimal konvertieren und rechnen
3	Weitere Anwendungen

1. **Parameter in eine Meldung einsetzen**

Aufgabe: In einer Fehlermeldungsroutine sollen in die angegebene Fehlermeldung statt &1, &2 usw. bis zu 9 variable Parameter eingesetzt werden.

```
msg1 = "Die Zahl &1 ist kleiner &2 oder größer &3."
zahl = 4711
say getmsg(msg1, zahl, 0, 1000)
exit
getmsg: procedure
  parse arg msg
  do argno = 1 to 9 until pos = 0
    pos = pos("&", msg)
    if pos > 0 then msg = left(msg, pos - 1) ||,
      arg(substr(msg, pos + 1, 1) + 1) || substr(msg, pos + 2)
  end argno
  return msg
```

Im Hauptprogramm werden die benötigten Meldungen definiert, hier stellvertretend msg1. zahl definieren wir hier direkt, in der Praxis kommt zahl z. B. als Eingabe, Parameter oder Element einer Datei herein. say gibt die von getmsg ergänzte Fehlermeldung am Bildschirm aus.

Das interne Unterprogramm schottet seine Variablen durch procedure gegen den Aufrufer ab. Die erste Argumentfolge, die Fehlermeldung msg, übernehmen wir mit parse arg. Die do-Schleife wird so oft durchlaufen, wie msg Parameter der Art &n (n = 1-9) enthält, jedoch maximal neunmal. Mit pos wird nach einem "&" in msg gesucht. Wenn pos größer als 0 ist, wurde "&" gefunden und msg wird um den entsprechenden Parameter ergänzt. Die aktualisierte Meldung setzt sich folgendermaßen zusammen:

- Aus dem linken Teil von msg bis vor das gefundene "&":
  left(msg, pos - 1).
- Aus dem Wert des n-ten Parameter &n. n wird mit substr(msg, pos + 1, 1) in msg ermittelt. Der Wert des Parameters &n ergibt sich als das (n + 1)te an getmsg übergebene Argument, also als arg(n + 1).
- Aus dem rechten Teil von msg nach dem gefundenen Platzhalter &n:
  substr(msg, pos + 2).

Nach dem Verlassen der do-Schleife wird mit return msg die aktualisierte Meldung

```
"Die Zahl 4711 ist kleiner 0 oder größer 1000."
```

an den Aufrufer zurückgegeben.

2. **Hexadezimal konvertieren und rechnen**

Aufgabe: Es sollen zwei Funktionen erstellt werden, mit denen eine Dezimalzahl in eine Hexadezimalzahl konvertiert werden kann und umgekehrt, und zwei weitere Funktionen, mit denen hexadezimal addiert und subtrahiert werden kann.

```
h1 = hex(367)
h2 = dec("16f")
h3 = hexadd("16f", "85")
h4 = hexsub("1f4", "85")
exit
hex: return d2x(arg(1))
dec: return x2d(arg(1))
```

```
hexadd: return d2x(x2d(arg(1)) + x2d(arg(2)))
hexsub: return d2x(x2d(arg(1)) - x2d(arg(2)))
```

➔ h1: "16F", h2: 367, h3: 1F4, h4: 16F.

Um die Funktion möglichst kompakt zu halten, übernehmen wir die übergebenenen Parameter direkt mit arg(1) bzw. arg(2). d2x wandelt eine Dezimalzahl in eine hexadezimale Zahl um und x2d bewirkt das Gegenteil. Mit return wird der jeweils ermittelte Wert an den Aufrufer zurückgegeben.

3. **Weitere Anwendungen**

Weitere Anwendungsmöglichkeiten:

- Standardwert vorgeben:

```
parse upper value arg(1)"N" with option 2
```

  Mit parse upper value wird das Ergebnis der Verkettung arg(1)"N" in Großbuchstaben umgesetzt. arg(1) bezeichnet das erste kommagetrennte Argument, das übergeben wird. Wenn arg(1) die Nullzeichenfolge ist, d. h. wenn keine Parameter übergeben werden, ergibt arg(1)"N" die Zeichenfolge "N", in allen anderen Fällen spielt das angekettete Zeichen "N" keine Rolle, weil nur das erste Zeichen des Ausdrucks ausgewertet wird. Auf diese Weise haben wir Vorsorge getroffen, daß der Standardwert "N" richtig behandelt wird. Die Positionsangabe 2 in der parse-Schablone bewirkt, daß nur das erste Zeichen von arg(1)"N" der Variablen option zugewiesen wird.

- Ausgelassene Parameter durch Standardwerte ersetzen:

```
parse arg str, anz, füll
if arg(2, "omitted") then anz = 1
if arg(3, "omitted") then füll = "."
```

  arg mit der Option "omitted" prüft, ob der Parameter im Funktionsaufruf ausdrücklich angegeben wurde, wenn nein, wird er mit dem Standardwert besetzt. Natürlich kann man das gleiche auch mit den beiden folgenden Anweisungen erreichen:

```
if anz = "" then anz = 1
if füll = "" then füll = "."
```

  Beachten Sie aber, daß if füll = "" besser durch if füll == "" ersetzt würde. Denn füll = "" (Nullzeichenfolge) gilt auch dann als wahr, wenn nur Leerstellen übergeben wurden. Wenn für füll eine Leerstelle übergeben wird, würde also das beabsichtigte Füllzeichen "□" fälschlicherweise durch einen Punkt ersetzt. füll == "" hingegen fragt exakt ab, ob füll die Nullzeichenfolge ist.

Argumentfolge

Die Zeichenfolge, die an ein Programm bzw. an ein externes oder internes Unterprogramm übergeben wird, besteht aus einer einzigen Zeichenfolge oder aus mehreren kommagetrennten Zeichenfolgen, die Argumente (Parameter) darstellen. Wir bezeichnen diese Zeichenfolgen als Argumentzeichenfolgen oder kurz als Argumentfolgen.

Programmaufruf. Kommagetrennte Argumentfolgen kann nur der REXX-Interpreter identifizieren. Dies trifft dann zu, wenn ein Programm als internes oder externes Unterprogramm aufgerufen wird.

Befehlsaufruf. Argumentfolgen, die über die Befehlsschnittstelle des Betriebssystems (z. B. TSO, OS/2- oder DOS-Befehlszeile) übergeben werden, können nicht in einzelne, kommagetrennte Argumentfolgen zerlegt werden, da die jeweiligen Befehlsprozessoren diese Art der Parameterübergabe nicht unterstützen. Das gleiche gilt für Programmaufrufe, die über die address-Anweisung an eine Befehlsschnittstelle übergeben werden (z. B. address tso, address command). In diesen Fällen wird die übergebene Zeichenfolge als *eine* Argumentfolge gewertet. Wenn kein Argumente (auch keine Leerstellen) übergeben werden, ist keine Argumentfolge vorhanden.

Aufrufart. Nach welchem Verfahren ein externes Programm aufgerufen wurde, kann mit der Anweisung parse source ermittelt werden, so daß ein Programm die Argumentzerlegung bei Bedarf der Aufrufart des Programms dynamisch anpassen kann. Der zweiten Variablen in der Schablone dieser Anweisung wird einer der Werte COMMAND (Befehlsaufruf), FUNCTION (Aufruf als externe Funktion) oder SUBROUTINE (Aufruf als externes Unterprogramm) zugewiesen.

Anweisungen und Funktionen. Mit der Anweisung parse arg wird die übergebene Zeichenfolge eines Programms in Argumentfolgen zerlegt. parse upper arg und die Anweisung arg leisten das gleiche, wandeln aber die Argumentfolgen in Großbuchstaben um.

Mit dem Funktionsaufruf arg() kann die Anzahl der übergebenen kommabegrenzten Argumentfolgen abgefragt werden, der Funktionsaufruf arg(n) liefert den Inhalt der n-ten Argumentfolge zurück. Mit arg(n, "exists") kann abgefragt werden, ob das n-te Argument angegeben wurde, mit arg(n, "omitted"), ob das n-te Argument weggelassen wurde.

Implementierungslimit. In call- und Funktionsaufrufen sind höchstens 20 Argumentfolgen erlaubt. Die Anweisungen parse arg bzw. arg und die Funktion arg können allerdings theoretisch beliebig viele Argumentfolgen zerlegen.

arithmetische Funktionen

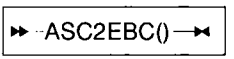
Übersicht

Arithmetische Funktionen sind Funktionen zur Bearbeitung von Zahlen. Syntax und Anwendungsmöglichkeiten der arithmetischen Funktionen können Sie der folgenden Tabelle entnehmen. Legende: *siehe* Funktionen - Übersicht.

*	Aufruf	Wirkung
s	ABS(zahl)	liefert den Absolutwert einer Zahl
s	DIGITS()	liefert die maximale Anzahl Ziffernstellen
s	FORM()	liefert das Format der exponentiellen Schreibweise
s	FORMAT([vk], [nk], [explg], [kipp])	rundet und formatiert eine Zahl
s	FUZZ()	liefert die numerische Unschärfe
s	MAX(zahl, [zahl,...])	liefert die größte von mehreren Zahlen
s	MIN(zahl, [zahl,...])	liefert die kleinste von mehreren Zahlen

*	Aufruf	Wirkung
s	SIGN(zahl)	liefert einen Wert für das Vorzeichen einer Zahl
s	TRUNC(zahl, [anzahl])	liefert eine Zahl mit anzahl Nachkommastellen

asc2ebc



Siehe auch: ebc2asc.

Die externe Funktion asc2ebc (ASCII t[w]o EBCDIC) gibt eine 256stellige Zeichenfolge zurück. Diese enthält die EBCDIC-Codes für die ASCII-Codes von 0-255, und zwar in der gleichen Reihenfolge. Die Funktion benötigt keine Parameter.

Anwendung

Zeichenfolgen vom ASCII-Code in den EBCDIC-Code übersetzen

Die Funktion asc2ebc wird dazu verwendet, um Zeichenfolgen im ASCII-Code in den EBCDIC-Code zu übersetzen. Im folgenden Beispiel wird die Zeichenfolge text1 zuerst aus dem ASCII-Code in den EBCDIC-Code übersetzt und dann mit Hilfe der Zwillingsfunktion ebc2asc wieder in den ASCII-Code zurückübertragen, so daß text2 den gleichen Inhalt hat wie text1.

```
text1 = "Ich weiß nicht, wie mir geschieht"
text2 = translate(text1, asc2ebc())
text2 = translate(text2, ebc2asc())
```

➔ text2: "Ich weiß nicht, wie mir geschieht"

Der erste Parameter im Aufruf der translate-Funktion bezeichnet die Zeichenfolge, die übersetzt werden soll, text1 bzw. text2. Der zweite Parameter gibt die Ausgabetabelle an. Dies sind hier die Übersetzungstabellen, die von asc2ebc() bzw. ebc2asc() geliefert werden. Wenn der dritte Parameter für die Eingabetabelle weggelassen wird, gelten standardmäßig alle Zeichen von "00"x bis "ff"x als Eingabetabelle.

Die translate-Funktion sucht jedes Zeichen aus text1 bzw. text2 in der Eingabetabelle. Anhand der gefundenen Positionsnummer wird dann das entsprechende Zeichen aus der Ausgabetabelle abgegriffen.

Ausdruck

Siehe: Klausel; Operator, Rangfolge; Term.

In Klauseln, d. h. Anweisungen, Zuweisungen und Befehlen, können Ausdrükke vorkommen. Ausdrücke bestehen aus Termen (Gliedern), Operatoren und runden Klammern.

Datentyp. Die Terme eines Ausdrucks werden als Zeichenfolgen interpretiert und haben keinen bestimmten Datentyp. Der Datentyp eines Terms wird vielmehr dynamisch anhand des Operators bestimmt, durch den er mit einem anderen Term verbunden ist. Der Wert, also das Ergebnis der Auswertung eines

Ausdrucks ist wieder eine Zeichenfolge. Da Zeichenfolgen in REXX beliebig lang sein dürfen (soweit es der verfügbare Speicher erlaubt), kann auch das Ergebnis eines Ausdrucks beliebig lang sein.

Auswertungsreihenfolge. Ausdrücke werden von links nach rechts ausgewertet. Klammern und die Rangfolge der Operatoren können jedoch die Reihenfolge der Auswertung ändern:

- Ein Teilausdruck in runden Klammern wird zuerst ausgewertet. Die Klammern in einem Funktionsaufruf haben eine besondere Bedeutung und verändern daher nicht die Reihenfolge der Auswertung.
- Wenn die Reihenfolge der Auswertung nicht durch Klammern verändert wird, werden in einem Ausdruck zuerst die Teilausdrücke ausgewertet, die durch die höchstrangigen Operatoren miteinander verbunden sind. Wenn mehrere gleichrangige Operatoren vorhanden sind, erfolgt die Auswertung der gleichrangigen Operatoren von links nach rechts.

Klammern und die Rangfolge der Operatoren haben allerdings keinen Einfluß auf die Reihenfolge, in der einzelne Terme ausgewertet werden: diese werden immer von links nach rechts ausgewertet.

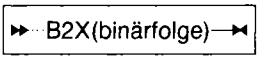
Die Auswertung eines Ausdrucks erfolgt immer vollständig, außer wenn ein Fehler während der Auswertung auftritt.

Nullzeichenfolgen. Die Nullzeichenfolge, also eine Zeichenfolge der Länge 0, kann als Term und als Ergebnis eines Ausdrucks auftreten. Dies gilt nicht für arithmetische und logische Operationen, bei denen die beteiligten Terme und das Auswertungsergebnis eine Zahl oder ein logischer Wert sein müssen bzw. sind.

Beispiele

REXX-Anweisungen	Erläuterung
a = 3; b = 10 s = (a + b) * 3	Die rechte Seite der Zuweisung s = (a + b) * 3 besteht aus den Termen a, b und 3, den Operatoren + und * sowie aus einem Paar von runden Klammern. Da der Ausdruck a + b in Klammern gesetzt ist, wird zuerst der Wert dieses Ausdrucks ermittelt. Da die Variablen a und b durch den Operator + verbunden sind, versucht der REXX-Interpreter, diese Variablen als Zahlen zu interpretieren. Beide Variablen enthalten eine Zahl, daher werden die beiden Zahlen 3 und 10 addiert und liefern als Ergebnis die Zeichenfolge "13". Der soweit aufgelöste Ausdruck lautet jetzt: 13 * 3. Er besteht aus den beiden Termen 13 und 3 und dem arithmetischen Operator *. Die beiden verbleibenden Terme werden nun miteinander multipliziert. Das Ergebnis ist die Zeichenfolge 39.
ort = "Stuttgart" z = ort", den "date("e")	Die rechte Seite der Zuweisung z = ort", den "date("e") besteht aus den Termen ort, ", den " und date("e"), die jeweils durch den Grenzoperator miteinander verbunden sind, d. h. ohne dazwischentretende Leerstellen oder den eigentlichen Verkettungsoperator ||. An den Inhalt der Variablen ort ("Stuttgart") wird die Zeichenfolge ", den " angehängt und an diese wiederum das Ergebnis, das der Funktionsaufruf date("e") zurückliefert, z. B. "25/05/96", so daß sich als Wert des Ausdrucks die Zeichenfolge "Stuttgart, den 25/05/96" ergibt.

b2x



Siehe auch: Konvertierfunktionen.

Die eingebaute SAA-Funktion b2x (binary t[w]o hexadecimal = binäres in hexadezimales Format umwandeln) wandelt die Binärfolge binärfolge in eine Hexfolge um und liefert diese als Funktionsergebnis zurück.

binärfolge wird ggf. durch führende Nullen auf volle Halbbytes ergänzt, so daß die Anzahl der Binärziffern ein Vielfaches von vier ist. Die Ausgabezeichenfolge umfaßt soviele Zeichen, wie binärfolge Halbbytes enthält.

Die Ausgabezeichenfolge wird *ohne* die Kennung X/x zurückgeliefert und enthält keine Leerstellen. Die Hexadezimalziffern A-F werden immer in Großbuchstaben ausgegeben.

Wenn binärfolge aus der Nullzeichenfolge besteht, wird eine Nullzeichenfolge zurückgeliefert.

Hinweis: Die fehlenden Konvertierfunktionen b2c und b2d können Sie durch Kombination von b2x mit x2c bzw. x2d simulieren.

binärfolge — Binärfolge von beliebiger Länge *ohne* die Kennung B/b. *Siehe:* Binärfolge.

Beispiele

REXX-Anweisungen	Erläuterung
hex1 = b2x("0") hex2 = b2x("11") hex3 = b2x("1000□1110")	Binärzahl umwandeln: hex1: "0", hex2: "3", hex3: "8E"
hex = b2x("")	Nullzeichenfolge umwandeln: hex: ""

Häufige Fehler

Keine Binärfolge. Die folgenden Funktionsaufrufe werden mit REXX-Fehler 40 quittiert:

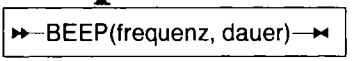
```
b2x("0100 0010"b)
b2x("41"x)
b2x("AB")
b2x(1500)
```

Die obigen Aufrufe liefern als Eingabeparameter (im ASCII-Code) die Zeichenfolgen "B", "A", "AB" und "1500". Diese Zeichenfolgen bestehen nicht oder nicht nur aus den Binärziffern 0 und 1 und sind daher keine gültigen Binärfolgen. Die folgenden Aufrufe führen hingegen zum Erfolg:

```
b2x("0011 0001"b)
b2x("31 30"x)
b2x(1100)
```

Die Argumente liefern nämlich (im ASCII-Code) zufällig die gültigen Zeichenfolgen "31"x = "1", "10" und "1100". Es ist aber zu vermuten, daß hier ein Versehen vorliegt und etwas ganz anderes gewollt wurde.

beep



Die eingebaute OS/2-Funktion beep (Ton) gibt über den Lautsprecher einen Ton mit der Hertz-Frequenz frequenz für die Dauer von dauer Millisekunden aus und liefert die Nullzeichenfolge zurück.

frequenz kann eine beliebige Ganzzahl im Bereich 37-32767 Hertz sein. dauer kann eine beliebige Ganzzahl im Bereich 1-60000 Millisekunden sein.

dauer	Ganzzahl im Bereich 1-60000. Dauer des Tons in Millisekunden.
frequenz	Ganzzahl im Bereich 37-32767. Frequenz des Tons in Hertz.

Beispiele

REXX-Anweisungen	Erläuterung
c_ = 262 sekunde = 1000 call beep(c_, sekunde/ 4)	c_ wird die Frequenz 262 Hertz, das ist die Frequenz des eingestrichenen c (c'), zugewiesen. Die Variable sekunde wird mit 1000 Millisekunden besetzt. beep gibt den Ton c' für sekunde / 4 Millisekunden, also für eine viertel Sekunde lang aus.

Anwendung

1	Akustischen Alarm ausgeben
2	Melodien abspielen

1. **Akustischen Alarm ausgeben**

Aufgabe: In einer Fehlerroutine soll ein akustischer Alarm ausgegeben werden.

```
alarm: call beep 512,300; call beep 256,500; call beep 200,400; return
```

Es werden (ungefähr) die Töne c', c'' und g ausgegeben und zwar für 300, 500 bzw. 400 Millisekunden.

2. **Melodien abspielen**

Aufgabe: Ein Unterprogramm soll beliebige Melodien anhand vorgegebener Töne und ihrer Notenwerte abspielen. Der Grundalgorithmus für die nachfolgende Lösung stammt von *F. J. Schubert* (nomen est omen?) aus Kirchheim und wurde mit dem *OS/2 Inside Rexx-Award 5/95* ausgezeichnet.

```
tempo = 200 /* 1/4 Noten pro Minute (30-300) */
thema = "f a f a f a f a p "
zeit = 1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/2" "
melodie = copies(thema" ", 3)
zeiten = copies(zeit" ", 3)
oktaven = copies("2 ", words(melodie))
call play melodie, oktaven, zeiten, tempo
```

Wir zeigen als Beispiel, wie das Klingeln eines Telefons simuliert werden kann. Das weiter unten erläuterte Unterprogramm play besorgt das Abspielen der Melodie. Wir definieren hier die Parameter, die wir für den Aufruf des Unterprogramms benötigen. Mit tempo geben wir an, wieviele Viertelnoten in einer Minute abgespielt werden sollen, hier z. B. 200. Mit tempo = 10 könnten wir dagegen die Sirene eines Feuerwehrautos nachahmen. Die Variable thema

definiert ein Thema aus den Werten f und a (sie stellen die Töne f" und a" dar, die Oktaven werden jedoch durch den Parameter oktaven angegeben), die sich viermal wiederholen, und einer Pause p. Insgesamt soll thema dreimal abgespielt werden, wir definieren daher melodie als dreifache Kopie von thema: copies(thema" ", 3). Die Notenwerte für thema definieren wir in der Variablen zeit und kopieren auch diese dreifach in die Variable zeiten. Schließlich müssen wir noch für jede Note angeben, welcher Oktave sie angehört. Da die beiden Töne f" und a" beide der zweigestrichenen Oktave angehören, definieren wir oktaven als Wortliste aus so vielen Kopien der Oktavennummer 2, wie Töne in melodie enthalten sind. Anschließend spielen wir die Melodie ab, indem wir die Funktion play mit den definierten Parametern aufrufen.

```
play: procedure
  parse upper arg melodie, oktave, dauer, tempo
  intvl = 196 / 185; a_ = 110
  tontab = "_"; lentab = "_"
  noten = "AIS_ H_ C CIS D DIS E F FIS G GIS A AIS H"
  do i = 1 to words(melodie)
    num = wordpos(word(melodie, i), noten)
    if num > 0 then do
      ton = (a_ * intvl ** num) * 2 ** word(oktave, i)
      tontab = tontab format(ton, 5, 0)
    end
    else tontab = tontab 30000
    len = 60 / tempo * 1000 * word(dauer, i)
    lentab = lentab format(len, 5, 0)
  end
  do i = 1 to words(tontab)
    call beep word(tontab, i), word(lentab, i)
  end
  return
```

In dem internen Unterprogramm, das durch procedure gegen den Aufrufer abgeschottet wird, lesen wir zuerst die Parameter mit parse upper arg in Großbuchstaben ein. Wir definieren die Intervallänge intvl und die Hertzfrequenz a_ des Grundtons a'. tontab und lentab, die Wortlisten für die abzuspielenden Töne und die zugehörigen Notenwerte, werden mit der Nullzeichenfolge initialisiert. Die Noten, die wir interpretieren wollen, werden in der Wortliste noten definiert.

In der folgenden do-Schleife werden die Parameter Ton für Ton abgearbeitet. Zuerst wird die Nummer num des Tones in der Wortliste noten festgestellt. Wenn der Ton in noten enthalten ist, wird intvl zur num-ten Potenz intvl ** num erhoben und mit der Frequenz des Grundtons a', nämlich mit a_, multipliziert. Das Ergebnis wiederum wird mit 2^n (n = Nummer der Oktave) multipliziert und mit format auf 5 Vor- und 0 Nachkommastellen formatiert an die Wortliste tontab angehängt. Wenn num 0 ist, gibt es diesen Ton nicht. Dies trifft z. B. bei der Pause p zu. In diesem Fall wird ein Ton von 30 kHz der Wortliste tontab angefügt. Die Tonlänge berechnen wir mit len = 60 / tempo * 1000 * word(dauer, i). Wenn tempo kleiner als 60 ist, wird der Quotient 60 / tempo größer als 1, d. h. das Tempo wird langsamer, und umgekehrt. Der Quotient wird mit 1000 Millisekunden und dem Notenwert word(dauer, i) multipliziert und an die Wortliste lentab angehängt.

In der zweiten do-Schleife spielen wir die einzelnen Töne mit call beep word(tontab, i), word(lentab, i) ab, wobei word(tontab, i) die Tonfrequenz, word(lentab, i) die Tondauer bezeichnet.

Befehl

Siehe auch: Anweisung; Fehlerbedingungen; Fehlerroutine; Klausel.

Ein Befehl ist eine REXX-Klausel, die aus einem Ausdruck besteht. Der Unterschied zwischen Befehlen und REXX-Anweisungen besteht darin, daß REXX-Anweisungen zum Sprachschatz von REXX gehören und vom REXX-Interpreter ausgeführt werden, während Befehle an eine andere Adreßumgebung zur Ausführung übergeben werden müssen.

Syntax. Ein Befehl an die aktive Adreßumgebung ist ein Ausdruck, der entweder für sich allein in einer Zeile steht oder aber durch Semikolon von anderen Anweisungen oder Befehlen in der gleichen Zeile getrennt ist. Der Ausdruck wird wie üblich ausgewertet, insbesondere werden Variablennamen, sofern sie nicht in Anführungszeichen stehen, durch ihren Wert substituiert.

Beachten Sie, daß alle Bestandteile des Ausdrucks, die nicht ausgewertet werden sollen, in Anführungszeichen gesetzt werden müssen. Wenn in einem Befehl keine Variablen oder Ausdrücke vorkommen, setzt man am besten den ganzen Befehl in Anführungszeichen. Wenn Sie dies unterlassen, laufen Sie Gefahr, daß Bestandteile des Befehls zufällig als Variablen definiert sind und deswegen substituiert werden.

```
del y*.*
```

Dieser Befehl scheitert daran, daß REXX die Sterne (*) als Multiplikationsoperatoren auffaßt und daher REXX-Fehler 41 zurückmeldet.

```
del y"*"."*"
```

Auch diese halbherzige Methode führt nur zum Ziel, wenn weder del noch y als Variablen definiert sind. Wenn etwa del = 0 definiert ist, scheitert der Befehl z. B. unter DOS mit dem Fehlercode 2. Wenn y = "heute" definiert ist, würden alle Dateien, die der Schablone "heute*.*" entsprechen, gelöscht, was offensichtlich nicht gewollt ist. Wenn in einem Befehl aber Variablen auftreten, dürfen diese *nicht* in Anführungszeichen gesetzt werden, damit sie substituiert werden können:

```
schablone = "y*.*"
"dir schablone"
```

Dieser Befehl scheitert oder liefert ein falsches Ergebis, weil die Variable schablone nicht substituiert wird. Sie muß daher außerhalb der Anführungszeichen angegeben werden, z. B. so:

```
"dir" schablone
```

Befehlsausführung. Die Adreßumgebung führt den übergebenen Befehl aus, setzt einen Rückkehrcode und gibt die Steuerung wieder an den REXX-Interpreter zurück. REXX weist den Rückkehrcode der speziellen Variablen rc zu, die dann innerhalb des REXX-Programms abgefragt werden kann.

Der folgende Befehl wird zuerst ausgewertet und ergibt display panel(edbpprt). Diese Zeichenfolge wird an die Adreßumgebung ispexec (unter ISPF) übergeben.

```
maske = "edbpprt"
address ispexec "display panel("maske")"
```

Wenn rc 0 ist, wurde display ordnungsgemäß ausgeführt, andernfalls enthält rc einen Rückkehrcode ungleich 0.

Ausgabeumleitung. Unter DOS und OS/2 kann die Standardausgabe (stdout) im Befehlsaufruf mit >ausgabedatei in die Datei ausgabedatei umgeleitet werden. Unter OS/2 kann auch die Standardfehlerausgabe (stderr) im Befehlsaufruf mit 2>fehlerdatei in die Datei fehlerdatei umgeleitet werden. Wenn die Ausgabe unwerwünscht ist, kann sie mit >nul bzw. 2>nul auf das Nullgerät umgeleitet werden.

Unter TSO wird die Ausgabe von TSO-Befehlen mit der TSO/E-Funktion outtrap in einen Variablenstamm umgeleitet. Mit der TSO/E-Funktion msg kann die Ausgabe von Meldungen durch Befehle und TSO/E-Funktionen unterdrückt werden.

Fehlerbedingungen. Für Fehler, die von Befehlen zurückgemeldet werden, können zwei Arten von Fehlerbedingungen gesetzt werden. Mit call on failure oder signal on failure wird die Failure-Bedingung gesetzt, d. h. wenn der REXX-Interpreter eine Failure-Bedingung abfängt, wird der Fehler an die entsprechende Fehlerroutine zur Behandlung übergeben. Mit call on error oder signal on error hingegen wird die Error-Bedingung gesetzt. Wenn allerdings die Failure-Bedingung *nicht*, aber die Error-Bedingung gesetzt ist, fängt die Error-Bedingung auch alle Failure-Bedingungen ab.

Zum Unterschied zwischen Error- und Failure-Bedingung: *siehe* Fehlerbedingungen.

Trace. Analog dazu werden Failure-Bedingungen von trace f (oder trace failure) verfolgt, Error- und Failure-Bedingungen von trace e (oder trace error). trace n (oder trace normal) ist die Standardoption von trace und identisch mit trace f.

Befehle, TSO - Übersicht

Legende

ta TSO/E REXX-Befehl, in allen MVS-Adreßumgebungen zulässig
tt TSO/E REXX-Befehl, nur in der TSO-Adreßumgebung zulässig

*	Aufruf	Wirkung
ta	DELSTACK	löscht die aktuelle Warteschlange oder den Inhalt der Standardwarteschlange
ta	DROPBUF [nummer]	löscht den aktuellen Warteschlangenpuffer oder alle Puffer ab einem bestimmten Puffer aufwärts

*	Aufruf	Wirkung
ta	EXECIO anzahl DISKR ddname [nummer] ([FINIS] [{LIFO | FIFO | SKIP} | STEM stamm])	öffnet eine Datei zum Lesen und schließt sie, liest Sätze in einen Variablenstamm oder die Warteschlange oder überspringt sie
	EXECIO anzahl DISKRU ddname [nummer] ([FINIS] [{LIFO | FIFO | SKIP} | STEM stamm])	öffnet eine Datei zum Lesen und Ändern und schließt sie, liest Sätze in einen Variablenstamm oder die Warteschlange oder überspringt Sätze
	EXECIO anzahl DISKW ddname ([FINIS] [STEM stamm])	öffnet eine Datei zum Schreiben und schließt sie, schreibt Sätze aus einem Variablenstamm oder der Warteschlange
ta	EXECUTIL EXECDD([NO]CLOSE)	SYSEXEC nach Laden eines REXX-Programms schließen oder nicht
	EXECUTIL HI	stoppt die Ausführung von REXX-Programmen
	EXECUTIL HT	schaltet die Terminalausgabe aus
	EXECUTIL RT	schaltet die Terminalausgabe ein
	EXECUTIL RENAME	ändert Einträge in einem Funktionspaket-Directory
	EXECUTIL SEARCHDD({YES | NO})	Suchfolge für REXX-Programme ändern
	EXECUTIL TE	schaltet den interaktiven Trace aus
	EXECUTIL TS	schaltet den interaktiven Trace ein
ta	HI	stoppt die Ausführung von REXX-Programmen
ta	HT	schaltet die Terminalausgabe aus
ta	MAKEBUF	legt einen neuen Puffer in der aktuellen Warteschlange an
ta	NEWSTACK	erzeugt eine neue Warteschlange
ta	QBUF	liefert die Anzahl der angelegten Puffer in der aktuellen Warteschlange
ta	QELEM	liefert die Anzahl der Elemente des aktuellen Warteschlangenpuffers
ta	QSTACK	liefert die Anzahl der vorhandenen Warteschlangen
ta	RT	schaltet die Terminalausgabe ein
ta	SUBCOM env	prüft, ob eine Adreßumgebung verfügbar ist
ta	TE	schaltet den interaktiven Trace aus
ta	TS	schaltet den interaktiven Trace ein

Bildschirmverarbeitung

Allgemein. Alle REXX-Implementierungen verfügen über einfache Anweisungen zur Bildschirmmanipulation: Mit der Anweisung say können Zeichenfolgen zeilenweise auf dem Bildschirm ausgegeben werden und mit parse pull bzw. pull können über die Tastatur eingegebene Zeichenfolgen eingelesen werden.

DOS und OS/2. Darüberhinaus stehen ab REXX Level 2, das unter DOS und OS/2 implementiert ist, die folgenden Funktionen zur Bildschirmmanipulation zur Verfügung:

- charin() und linein() lesen über die Tastatur eingegebene Zeichenfolgen ein.
- charout(, *zeichenfolge*) und lineout(, *zeichenfolge*) geben Zeichenfolgen auf dem Bildschirm aus.

DOS. Unter OS/2 kommen die folgenden DOS-Dienstprogrammfunktionen hinzu:

- rxscrsiz liefert die Bildschirmgröße
- rxgetpos, rxsetpos und rxcrstat zum Manipulieren des Cursors
- rxcls zum Löschen des Bildschirms
- rxwrscr zur Ausgabe von Zeichenfolgen auf dem Bildschirm
- rxgetkey zum Einlesen von Tastatureingaben

OS/2. Unter OS/2 kommen die OS/2-Funktion beep zur Ausgabe von Tönen über den Lautsprecher sowie die folgenden OS/2-Dienstprogrammfunktionen hinzu:

- systextscreensize liefert die Bildschirmgröße
- syscurpos und syscurstate zum Manipulieren des Cursors
- syscls zum Löschen des Bildschirms
- systextscreenread zum Einlesen von Zeichenfolgen vom Bildschirm
- sysgetkey zum Einlesen von Tastatureingaben

ISPF. Unter ISPF stehen *Panels* zur maskenorientierten Bildschirmverarbeitung zur Verfügung.

Binärfolge

Binäre Zeichenfolgen, oder kurz: Binärfolgen, werden benötigt, um die Bitabfolge innerhalb einer Zeichenfolge darzustellen.

Eine Binärfolge (*binary string*) ist eine Zeichenfolge, die nur aus Binärziffern, also aus den Zeichen "0" und "1" besteht. Je acht Zeichen einer Binärfolge beschreiben ein Byte (Zeichen) einer normalen Zeichenfolge. Aus einer Binärfolge bildet REXX intern eine normale Zeichenfolge, wobei je acht Binärziffern in ein Byte umgewandelt werden.

Konstante Binärfolgen werden in Anführungszeichen oder Hochkommas eingeschlossen und mit der Kennung b oder B beendet.

Beispiel: Die konstante Binärfolge "1000 1110"b wird intern in ein Byte bzw. Zeichen mit dem Wert 142 (Dezimalzahl), 8E (Hexadezimalzahl) oder 10001110 (Binärzahl) umgewandelt.

Hinweis: Die eingebaute SAA-Funktion b2x verlangt als Argument eine Binärfolge. Wenn diese als Konstante angegeben werden soll, muß die Kennung b/B weggelassen werden. Dies ist eine etwas unglückliche Regelung, da Ausnahmen wie diese eine beliebte Fehlerquelle sind. Der Grund für diese Sonderregelung ist darin zu suchen, daß b2x ohnehin nur Binärfolgen erwartet und die Kennung b/B daher überflüssig ist.

Eine Binärfolge wird nach den folgenden Regeln aufgebaut.

Nr.	Regel
1	Eine Binärfolge besteht aus 0, ein oder mehreren *Binärziffern.* Binärziffern sind die Zeichen *0* und *1*.
2	Ein *Byte* wird durch *acht Binärziffern* dargestellt. Das erste Byte in einer Binärfolge kann auch aus *weniger* als acht Ziffern bestehen. REXX füllt in diesem Fall die Zahl mit führenden Nullen auf, so daß intern die Anzahl der Binärziffern immer ein Vielfaches von acht ist.
3	An Byte- oder Halbbytegrenzen können zur besseren Lesbarkeit ein oder mehrere *Leerstellen* eingefügt werden. Hingegen sind am Anfang und Ende einer Binärfolge keine Leerstellen zulässig. Bei der internen Umwandlung der Binärfolge in eine Binärzahl bleiben Leerstellen unberücksichtigt.
4	Eine Binärfolge muß von *Anführungszeichen* (") oder *Hochkommas*(') eingeschlossen werden.
5	Dem abschließenden Anführungszeichen bzw. Hochkomma muß der *Buchstabe B/b* (b = binär) folgen. Dieser Kennung darf nicht unmittelbar ein Symbol folgen, da sonst Kennung und Symbol zusammen als ein Symbol betrachtet werden. **Hinweis:** Die Konvertierfunktion b2x verlangt als Eingabeparameter Binärfolgen *ohne* die Kennung B/b. Die Konvertierfunktion x2b gibt eine Binärfolge *ohne* die Kennung B/b zurück.
6	**Implementierungslimit:** Die Länge einer Binärfolge ist auf *800 Binärziffern* begrenzt. Trennende Leerstellen werden dabei nicht mitgerechnet. Dies ist das SAA-Minimum, unter MVS sind bis zu 2000 Binärziffern zulässig. **Tip:** Wenn Sie plattformneutral programmieren wollen, sollten Sie Binärfolgen auf 800 Binärziffern beschränken.
7	Eine Variable mit dem Namen B/b kann nicht unmittelbar an eine Zeichenfolge angekettet werden. **Tip:** Verwenden Sie *nicht* den Variablennamen B bzw. b. Sie vermeiden dadurch unbewußte Fehlermöglichkeiten.

Die Kodierung der Bytes, die durch eine Binärfolge erzeugt werden, unterscheidet sich in nichts von der Kodierung, die durch eine gleichwertige *Hexfolge* oder eine *Textzeichenfolge* erzeugt wird:

```
"4142"x = "0100 0001 0100 0010"b = "AB"
```

Alle drei Schreibweisen, die Hexfolge, die Binärfolge und die Textzeichenfolge erzeugen die gleiche Kodierung, d. h. die gleiche Bitabfolge im Speicher.

Beispiele

In der folgenden Tabelle finden Sie einige *richtige* Beispiele für Binärfolgen. Die Beschreibung der Regeln finden Sie in in der obenstehenden Regel-Tabelle.

Beispiele	Erläuterung	Regeln
""B "0"b '10001110'B "11"b	Die Zeichenfolgen sind in Anführungszeichen eingeschlossen und enden mit einem B/b. Das erste Beispiel zeigt eine binäre Nullzeichenfolge. Die übrigen Zeichenfolgen bestehen aus den Ziffern 0 und 1.	1-2, 4-5

Beispiele	Erläuterung	Regeln
"1000□1110"b "1□□□1110"b	Die Zeichenfolgen bestehen aus den Ziffern 0 und 1, sind in Anführungszeichen eingeschlossen und enden mit einem B/b. Halbytes sind zur besseren Lesbarkeit durch Leerstellen getrennt.	1-5
"00000000 ...01100011"b ...II"01100100"b	"0000 0000...0110 0011"b steht für eine Binärfolge aus den 100 Binärzahlen von 0-99. Da die Länge einer Binärfolge auf 800 Binärziffern begrenzt ist, müssen längere Zeichenfolgen durch Verkettung erzeugt werden. Daher wird der Binärfolge "0000 0000...0110 0011"b durch Verkettung die Binärfolge "0110 0100" angehängt, so daß eine Tabelle von 101 Binärzahlen entsteht. **Hinweis:** Beachten Sie, daß in älteren Implementierungen von REXX die Länge einer Klausel auf 500 Zeichen beschränkt ist. Dies kann dazu führen, daß eine Verkettung zwar nicht die Regel 7 verletzt, aber an dieser Begrenzung scheitert. Da die maximale Länge für Variableninhalte aber praktisch unbegrenzt ist, entsteht durch diese Einschränkung keine ernsthafte Behinderung.	6

Die folgende Tabelle enthält einige *falsche* Beispiele für Binärfolgen. Die Beschreibung der Regeln finden Sie in der obenstehenden Regel-Tabelle.

Beispiele	Erläuterung	Regeln
"1000 1112"b	Die Binärfolge enthält eine falsche Binärziffer (2). Führt zu REXX-Fehler 15.	1
"□1110"b "1110□□"b	Leerstellen am Anfang oder am Ende einer Binärfolge sind nicht erlaubt. Führt zu REXX-Fehler 15.	3
1110b 1110"b "1110b "1110'b "1110b"	Die Binärfolgen sind nicht von Anführungszeichen oder Hochkommas eingeschlossen. Sie fehlen entweder ganz, am Anfang der Zeichenfolge oder vor der Kennung b, oder es wird am Anfang ein anderes Begrenzungszeichen als vor der Kennung b verwendet. Führt zu REXX-Fehler 15.	4
"1110" "1110"x	Im ersten Fall fehlt die Kennung b. Im zweiten Fall ist die Kennung x falsch, da sie keine Binärfolge, sondern eine Hexfolge kennzeichnet.	5
b = "xyz" y = "110"b	Die Variable b, die unmittelbar an die Zeichenfolge "abc" angekettet werden soll, führt zu einem ungewollten Ergebnis. y wird in diesem Fall nicht wie erwartet die Zeichenfolge "110xyz" zugewiesen, sondern die Binärzahl "0000 0110"b oder "06"x, da REXX "110"b für eine Binärfolge hält.	7
b = "xyz" y = "320"b	Die Variable b soll unmittelbar an die Zeichenfolge "320" angekettet werden. REXX hält aber "320"b für eine Binärfolge, und da sie Zeichen enthält, die keine Binärziffern sind, wird der REXX-Fehler 15 gemeldet.	7
str = "xyz" str = "110"bstr	Da das Symbol bstr undefiniert ist, wird "BSTR" an "110" angekettet, so daß sich str = "110BSTR" ergibt. Offenbar sollte aber der Inhalt von str an die Binärfolge "110"b angehängt werden. Um dieses Ziel zu erreichen, muß "110"b durch den Verkettungsoperator II mit str verbunden werden: str = "110"x II str.	5

Beispiele	Erläuterung	Regeln
"00000000 ...11111111"b	"0000 0000...1111 1111"b steht für eine Binärfolge aus den 256 Binärzahlen von 0-255. Da die Länge einer Binärfolge auf 800 Binärziffern begrenzt ist (etwaige Leerstellen zählen nicht mit), führt diese Definition zu REXX-Fehler 30. **Hinweis:** Auch wenn Regel 7 beachtet wird, etwa indem Zeichenfolgen > 800 Binärziffern durch Verkettung kleinerer Zeichenfolgen gebildet werden, kann es in älteren REXX-Implementierungen zu REXX-Fehler 12 kommen, weil die maximale Länge von 500 Zeichen für eine Klausel überschritten wird. In diesem Fall kann man sich aber behelfen, indem man die Binärfolgen stückweise einer Variablen zuweist, da die maximale Länge für Variableninhalte praktisch unbegrenzt ist.	7

Anwendung

1	Externe kontra interne Darstellung
2	Umwandlung in und von Binärzahlen

1. Externe kontra interne Darstellung

Eine Binärfolge ist keine Binärzahl, sie stellt lediglich eine Folge von Bytes dar, die den binären Wert haben, der durch die Binärfolge ausgedrückt wird.

Die Binärfolge repräsentiert die externe Darstellung einer Binärzahl, wie sie intern von REXX realisiert wird.

```
"01000100 01000101 01010020 00100000 01010110 01000001 01010100
...01000101 01010010"b
```

Die obige Binärfolge wird von REXX intern als eine Folge von binären Bytes dargestellt. Wenn man diese binäre Bytefolge als Zeichenfolge im ASCII-Code interpretiert, bedeutet sie: "DER□VATER".

2. Umwandlung in und von Binärzahlen

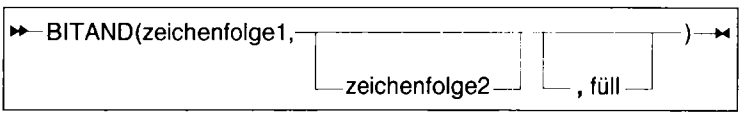
Zur Umwandlung in und von Binärzahlen stehen die beiden eingebauten SAA-Funktionen b2x und x2b zur Verfügung. Damit kann nur in das hexadezimale Format umgewandelt werden und umgekehrt. Die Umwandlung in dezimales Format oder in Zeichenformat kann aber durch die Kombination mit anderen Konvertierfunktionen erfolgen. *Siehe:* Konvertierfunktionen.

```
bin1 = "1000 1110"
hex1 = b2x(bin1)
hex2 = "8e"
bin2 = x2b(hex2)
```

➔ hex1: "8E", bin2: "10001110"

Beachten Sie, daß die beiden Konvertierfunktionen eine Binär- bzw. Hexfolge ohne die Kennung B/b bzw. X/x erzeugen.

bitand



Siehe: logische Funktionen.

Die eingebaute SAA-Funktion bitand (bit by bit AND = bitweise UND) verknüpft die beiden Zeichenfolgen zeichenfolge1 und zeichenfolge2 bitweise durch logisches UND miteinander und liefert das Ergebnis als Zeichenfolge zurück. Die Ergebniszeichenfolge ist so lang wie die längere der beiden Ausgangszeichenfolgen.

zeichenfolge1 und zeichenfolge2 sind beliebige Zeichenfolgen. Wenn zeichenfolge2 fehlt, wird zeichenfolge1 mit der Nullzeichenfolge verknüpft, oder wenn das Füllzeichen füll angegeben ist, mit einer Zeichenfolge, die in der Länge von zeichenfolge1 mit dem Füllzeichen aufgefüllt wird.

Wenn das Füllzeichen füll fehlt, endet die bitweise Verknüpfung, sobald die kürzere der beiden Ausgangszeichenfolgen erschöpft ist, und der nicht verarbeitete Rest der längeren Zeichenfolge wird an das Ergebnis angehängt. Wenn füll angegegen wird, wird vor der logischen Verknüpfung die kürzere der beiden Zeichenfolgen hinten mit dem Füllzeichen aufgefüllt.

Die logische UND-Verknüpfung bewirkt, daß nur die Verknüpfung zweier Einserbits zu einem Einserbit im Ergebnis führt: 1 & 1 ⇒ 1. Die übrigen Verknüpfungen 0 & 1, 1 & 0 und 0 & 0 ergeben ein Nullbit im Ergebnis.

Bits löschen: Zum *Löschen* von Bits in einer Zielzeichenfolge baut man eine Bitmaske auf, die an den entsprechenden Stellen ein Nullbit und an allen anderen Stellen Einserbits enthält, und verknüpft die Bitmaske mit der Zielfolge durch bitand. Eine Maske, die nur aus Nullbits besteht, löscht alle Bits in der Zielfolge.

Bits testen. Um den Wert von Bits in einer Zielzeichenfolge zu testen, baut man eine Maske auf, in der die zu testenden Bits gesetzt sind, und verknüpft die Bitmaske mit der Zielfolge durch bitand. Dann baut man eine zweite Maske auf, in der nur *die* getesteten Bits gesetzt sind, für die wir den Wert 1 erwarten. Wenn der Vergleich dieser Maske mit dem Ergebnis der bitand-Funktion übereinstimmt, haben alle getesteten Bits den erwarteten Wert (0 oder 1).

füll — Zeichen, mit dem die kürzere der beiden Ausgangszeichenfolgen hinten aufgefüllt werden soll. Die Nullzeichenfolge oder eine Zeichenfolge, die aus mehr als einem Zeichen besteht, führt zu REXX-Fehler 40.

zeichenfolge1 — Beliebige Zeichenfolge.

zeichenfolge2 — Beliebige Zeichenfolge.
Standard: Nullzeichenfolge.

Beispiele

REXX-Anweisungen	Erläuterung
b1 = bitand("abc") b2 = bitand("abc", "_")	keine Verknüpfung: Da zeichenfolge2 fehlt, wird die Nullzeichenfolge verwendet, und weil das Füllzeichen fehlt, wird in der Länge der Nullzeichenfolge verknüpft, also gar nicht. An das Ergebnis wird zeichenfolge1 = "abc" angehängt, so daß wir wieder zeichenfolge1 erhalten: "abc". b2 liefert natürlich das gleiche Ergebnis.

REXX-Anweisungen	Erläuterung
b3 = bitand("1010 1100"b, ..."0101 0011"b)	gleichlange Zeichenfolgen: Da bei b3 beide Binärfolgen gleich lang sind und die Bits in jeder Stelle entgegengesetzt sind, werden im Ergebnis alle Bits gelöscht: "00"x = "0000 0000"b.
b4 = bitand("ff 3e"x, "00"x)	ungleichlange Zeichenfolgen, kein Füllzeichen: Die beiden 4 Bytes "ff"x und "00"x werden durch logisches UND verknüpft. Das ergibt "00"x, weil die Bits in allen Stellen entgegengesetzt sind. Da kein Füllzeichen angegeben ist, wird das Ergebnis um das nicht verarbeitete Byte "3e"x verlängert: "00 3e"x.
b5 = bitand("ff 3e"x, "00"x, ..."ff"x)	ungleichlange Zeichenfolgen mit Füllzeichen: Die kürzere Zeichenfolge "00"x wird durch das Füllzeichen "ff"x auf die Länge der längeren Zeichenfolge "ff 3e"x aufgefüllt: "00 ff"x. Die logische Verknüpfung der beiden so erhaltenen Zeichenfolgeen durch UND ergibt "00 3e"x.

Anwendung

1	Bits testen
2	Vorzeichen einer Binärzahl abfragen
3	ASCII-Zeichen in Großbuchstaben umwandeln

1. Bits testen

Aufgabe: In der Zeichenfolge zeichenfolge = "0011 1011"b soll getestet werden, ob das dritte Bit von links gesetzt und das sechste Bit gelöscht ist.

```
zeichenfolge = "0011 1011"b
testmaske = "0010 0100"b
and = bitand(zeichenfolge, testmaske)
vergleichsmaske = "0010 0000"b
vergleich = (and = vergleichsmaske)
```

➔ and: "0010 0000"b; vergleich: 1

Dazu bauen wir die Testmaske testmaske auf, in der die zu testenden Bits gesetzt sind. Die Verknüpfung and = bitand(z, t) ergibt "0010 0000"b. Wir bauen jetzt die Vergleichsmaske vergleichsmaske auf, in der nur die zu testenden Bits gesetzt sind, die den Wert 1 haben sollen. Der Vergleich and = vergleichsmaske ergibt 1 (wahr).

```
zeichenfolge = "0011 1111"b
testmaske = "0010 0100"b
and = bitand(zeichenfolge, testmaske)
vergleichsmaske = "0010 0000"b
vergleich = (and = vergleichsmaske)
```

➔ and: "0010 0100"b; vergleich: 0

Wenn zeichenfolge hingegen den Wert "0011 1111"b hat, ergibt and den Wert "0010 0100"b und and = vergleichsmaske demzufolge den Wert 0 (falsch), da and ungleich vergleichsmaske ist. Das dritte Bit ist zwar erwartungsgemäß gesetzt, das sechste Bit ist aber nicht gelöscht.

Siehe auch: storage, Anwendung.

2. Vorzeichen einer Binärzahl abfragen

Aufgabe: Bei einer Binärzahl ist das Vorzeichen im höchstwertigen Bit gespeichert, d. h. im linkesten Bit des ersten Bytes der Zahl. Wenn das Bit 0 ist, ist die Zahl positiv, andernfalls negativ. Das Vorzeichen einer Binärzahl beliebiger Länge soll abgefragt werden, ohne die Zahl in eine Dezimalzahl umzuwandeln.

```
zahl = "80 ff"x /* -255 */
if pos("80"x, bitand(zahl, "80"x)) = 1 then say "Die Zahl ist negativ"
```

Wenn das erste Bit im ersten Byte von zahl gesetzt ist, ergibt die logische UND-Verknüpfung von zahl und "80"x im ersten Byte wiederum "80"x, wenn also pos("80"x, ...) gleich 1 ist, ist zahl negativ. Natürlich kann man diese Abfrage auch durch if left(zahl, 1) > 127 ersetzen.

3. ASCII-Zeichen in Großbuchstaben umwandeln

Aufgabe: ASCII-Zeichen sollen, sofern es sich um die Kleinbuchstaben a-z handelt, mit der bitand-Funktion in Großbuchstaben umgewandelt werden.

```
name = bitand("Sibylle",, "df"x)
➔ name: "SYBILLE"
```

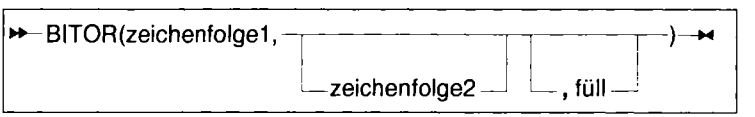
Wir zeigen dieses uralte Standardbeispiel für die bitand-Funktion hier nur deswegen, weil daran deutlich wird, wie man mit Bitmasken gezielt Bits löschen kann, ohne die anderen Bits zu verändern. Wenn Sie eine Zeichenfolge in Großbuchstaben umwandeln wollen, sollten Sie dazu die codeunabhängige Funktion translate verwenden bzw. die upper-Option einiger Anweisungen. Die hier vorgestellte Lösung ist auch deswegen nicht einwandfrei, weil das bedingungslose Löschen des Bits mit der Wertigkeit "20"x dazu führt, daß auch die Zeichen mit den Codes "60"x und "7b"x - "7f"x in andere Zeichen umgewandelt würden.

Da die zweite Zeichenfolge im Aufruf von bitand fehlt, tritt die Nullzeichenfolge an ihre Stelle. Da diese kürzer als die erste Zeichenfolge ist und ein Füllzeichen angegeben wurde, wird die zweite Zeichenfolge mit dem Füllzeichen "df"x aufgefüllt, d. h. in jedem Byte der zweiten Zeichenfolge sind alle Bits außer dem Bit mit der Wertigkeit "20"x gesetzt.

Daher bleiben auch in den Ergebnisbytes alle Bits außer diesem erhalten. Das Bit mit der Wertigkeit "20"x wird in jedem Fall gelöscht, da eine UND-Verknüpfung, in der ein Nullbit enthalten ist, immer zu einem Nullbit im Ergebnis führt.

Da sich im ASCII-Code die Großbuchstaben von den Kleinbuchstaben dadurch unterscheiden, daß bei den Kleinbuchstaben das Bit mit der Wertigkeit "20" gesetzt ist, erreichen wir durch diese logische Verknüpfung, daß "Sibylle" in "SIBYLLE" umgewandelt wird.

bitor



Siehe: logische Funktionen.

Die eingebaute SAA-Funktion bitor (bit by bit OR = bitweise ODER) verknüpft die beiden Zeichenfolgen zeichenfolge1 und zeichenfolge2 bitweise durch logisches ODER miteinander und liefert das Ergebnis als Zeichenfolge zurück. Die Ergebniszeichenfolge ist so lang wie die längere der beiden Ausgangszeichenfolgen.

zeichenfolge1 und zeichenfolge2 sind beliebige Zeichenfolgen. Wenn zeichenfolge2 fehlt, wird zeichenfolge1 mit der Nullzeichenfolge verknüpft, oder wenn das Füllzeichen füll angegeben ist, mit einer Zeichenfolge, die in der Länge von zeichenfolge1 mit dem Füllzeichen aufgefüllt wird.

Wenn das Füllzeichen füll fehlt, endet die bitweise Verknüpfung, sobald die kürzere der beiden Ausgangszeichenfolgen erschöpft ist, und der nicht verarbeitete Rest der längeren Zeichenfolge wird an das Ergebnis angehängt. Wenn füll angegegen wird, wird vor der logischen Verknüpfung die kürzere der beiden Zeichenfolgen hinten mit dem Füllzeichen aufgefüllt.

Die logische ODER-Verknüpfung bewirkt, daß nur die Verknüpfung zweier Nullbits zu einem Nullbit im Ergebnis führt: 0 | 0 ⇒ 0. Die übrigen Verknüpfungen 0 | 1, 1 | 0 und 1 | 1 ergeben ein Einserbit im Ergebnis.

Bits setzen: Zum *Setzen* von Bits in einer Zielzeichenfolge baut man eine Bitmaske auf, die an den entsprechenden Stellen ein Einserbit und an allen anderen Stellen Nullbits enthält, und verknüpft die Bitmaske mit der Zielfolge durch bitor. Eine Maske, die nur aus Einserbits besteht, setzt alle Bits in der Zielfolge.

füll — Zeichen, mit dem die kürzere der beiden Ausgangszeichenfolgen hinten aufgefüllt werden soll. Die Nullzeichenfolge oder eine Zeichenfolge, die aus mehr als einem Zeichen besteht, führt zu REXX-Fehler 40.

zeichenfolge1 — Beliebige Zeichenfolge.

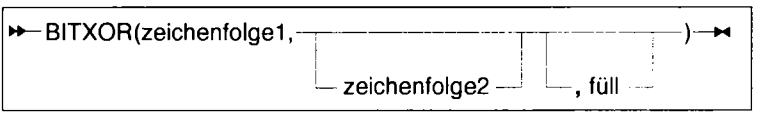
zeichenfolge2 — Beliebige Zeichenfolge.
Standard: Nullzeichenfolge.

Beispiele

REXX-Anweisungen	Erläuterung
b1 = bitor("abc") b2 = bitor("abc", "_")	keine Verknüpfung: Da zeichenfolge2 fehlt, wird die Nullzeichenfolge verwendet, und weil das Füllzeichen fehlt, wird in der Länge der Nullzeichenfolge verknüpft, also gar nicht. An das Ergebnis wird zeichenfolge1 = "abc" angehängt, so daß wir wieder zeichenfolge1 erhalten: "abc". b2 liefert natürlich das gleiche Ergebnis.
b3 = bitor("1010 1100"b, ..."1111 1111"b)	gleichlange Zeichenfolgen: Da bei b3 beide Binärfolgen gleich lang sind und die Bits in einer Zeichenfolge alle gesetzt sind, sind im Ergebnis ebenfalls alle Bits gesetzt: "ff"x = "1111 1111"b.

REXX-Anweisungen	Erläuterung
b4 = bitor("ff 3e"x, "00"x)	ungleichlange Zeichenfolgen, kein Füllzeichen: Die beiden Bytes "ff"x und "00"x werden durch logisches ODER verknüpft. Das ergibt "ff"x, weil die Bits des einen Bytes alle gesetzt sind. Da kein Füllzeichen angegeben ist, wird das Ergebnis um das nicht verarbeitete Byte "3e"x verlängert: "ff 3e"x.
b5 = bitor("ff 3e"x, "00"x, ..."7c"x)	ungleichlange Zeichenfolgen mit Füllzeichen: Die kürzere Zeichenfolge "00"x wird durch das Füllzeichen "7c"x auf die Länge der längeren Zeichenfolge "ff 3e"x aufgefüllt: "00 7c"x. Die logische Verknüpfung der beiden so erhaltenen Zeichenfolgen durch ODER ergibt "ff 7e"x.

bitxor



Siehe: logische Funktionen.

Die eingebaute SAA-Funktion bitxor (bit by bit eXclusive OR = bitweise Exklusiv-ODER) verknüpft die beiden Zeichenfolgen zeichenfolge1 und zeichenfolge2 bitweise durch logisches Exklusiv-ODER miteinander und liefert das Ergebnis als Zeichenfolge zurück. Die Ergebniszeichenfolge ist so lang wie die längere der beiden Ausgangszeichenfolgen.

zeichenfolge1 und zeichenfolge2 sind beliebige Zeichenfolgen. Wenn zeichenfolge2 fehlt, wird zeichenfolge1 mit der Nullzeichenfolge verknüpft, oder wenn das Füllzeichen füll angegeben ist, mit einer Zeichenfolge, die in der Länge von zeichenfolge1 mit dem Füllzeichen aufgefüllt wird.

Wenn das Füllzeichen füll fehlt, endet die bitweise Verknüpfung, sobald die kürzere der beiden Ausgangszeichenfolgen erschöpft ist, und der nicht verarbeitete Rest der längeren Zeichenfolge wird an das Ergebnis angehängt. Wenn füll angegegen wird, wird vor der logischen Verknüpfung die kürzere der beiden Zeichenfolgen hinten mit dem Füllzeichen aufgefüllt.

Die logische Exklusiv-ODER-Verknüpfung bewirkt, daß die Verknüpfung zweier ungleicher Bits zu einem Einserbit im Ergebnis führt: 0 && 1, 1 && 0 ⇒ 1. Zwei gleiche Bits führen zu einem Nullbit im Ergebnis: 0 && 0, 1 && 1 ⇒ 0.

Bits umkehren: Zum *Umkehren* von Bits einer Zielzeichenfolge baut man eine Bitmaske auf, die an den entsprechenden Stellen ein Einserbit und an allen anderen Stellen Nullbits enthält, und verknüpft die Bitmaske mit der Zielfolge durch bitxor. Eine Maske, die nur aus Einserbits besteht, kehrt alle Bits in der Zielfolge um. Dies entspricht der logischen Operation \ (NOT = NICHT).

Alle Bits löschen. Wenn die Bitmaske gleich der Zielfolge ist, werden alle Bits in der Zielfolge gelöscht.

füll — Zeichen, mit dem die kürzere der beiden Ausgangszeichenfolgen hinten aufgefüllt werden soll. Die Nullzeichenfolge oder eine Zeichenfolge, die aus mehr als einem Zeichen besteht, führt zu REXX-Fehler 40.

zeichenfolge1 — Beliebige Zeichenfolge.

zeichenfolge2 — Beliebige Zeichenfolge.
Standard: Nullzeichenfolge.

Beispiele

REXX-Anweisungen	Erläuterung
b1 = bitxor("abc") b2 = bitxor("abc", " ")	keine Verknüpfung: Da zeichenfolge2 fehlt, wird die Nullzeichenfolge verwendet, und weil das Füllzeichen fehlt, wird in der Länge der Nullzeichenfolge verknüpft, also gar nicht. An das Ergebnis wird zeichenfolge1 = "abc" angehängt, so daß wir wieder zeichenfolge1 erhalten: "abc". b2 liefert natürlich das gleiche Ergebnis.
b3 = bitxor("1010 1100"b, ..."1111 1111"b)	gleichlange Zeichenfolgen: Da bei b3 beide Binärfolgen gleich lang sind und die Bits in einer Zeichenfolge alle gesetzt sind, werden im Ergebnis alle Bits umgekehrt: "53"x = "0101 0011"b.
b4 = bitxor("ff 3e"x, "00"x)	ungleichlange Zeichenfolgen, kein Füllzeichen: Die beiden Bytes "ff"x und "00"x werden durch logisches Exklusiv-ODER verknüpft. Das ergibt "ff"x, weil die Bits des einen Bytes alle 1 und die des anderen alle 0 sind. Da kein Füllzeichen angegeben ist, wird das Ergebnis um das nicht verarbeitete Byte "3e"x verlängert: "ff 3e"x.
b5 = bitxor("ff 3e"x, "00"x, ..."f0"x)	ungleichlange Zeichenfolgen mit Füllzeichen: Die kürzere Zeichenfolge "00"x wird durch das Füllzeichen "f0"x auf die Länge der längeren Zeichenfolge "ff 3e"x aufgefüllt: "00 f0"x. Die logische Verknüpfung der beiden so erhaltenen Zeichenfolgeen durch Exklusiv-ODER ergibt "ff ce"x.

Anwendung

1	Unterschiede in zwei Zeichenfolgen ermitteln
2	Unterschiede zweier Zeichenfolgen markieren

1. Unterschiede in zwei Zeichenfolgen ermitteln

Aufgabe: Es soll die Anzahl der unterschiedlichen Zeichen in den beiden Zeichenfolgen text1 und text2 ermittelt werden.

Wir entwickeln zuerst die Problemlösung in Einzelschritten *(Langform)*, damit die verschachtelte endgültige Lösung *(Kurzform)* besser verständlich wird. Mit bitxor (text1, text2) wird die Operation des logischen Exklusiv-Oder auf text1 und text2 angewandt, d. h. zwei gleiche Quellbits führen im Ergebnis zu einem Bit mit dem Wert 0, während zwei ungleiche Quellbits ein Bit mit dem Wert 1 liefern. temp1 liefert also eine Zeichenfolge, in der alle gleichen Zeichen durch "00"x repräsentiert sind.

```
text1 = "Canetti, Elias"
text2 = "Canetti, Veza"

/* Langform: */
temp1 = bitxor (text1, text2)
temp2 = translate (temp1, "00"x|| "□", "□" || "00"x)
temp3 = space (temp2, 0)
diff = length(temp3)
/* Kurzform: */
diff = length(space(translate(bitxor(text1, text2), "00"x || "□", "□" || "00"x), 0))
```

➔ Auf einem ASCII-Rechner:
temp1: "00000000000000000001309130073"x
temp2: "20202020202020202201309132073"x
temp3: "13091373", diff: 4

➔ Auf einem EBCDIC-Rechner:
temp1: "00000000000000000020162000e2"x
temp2: "40404040404040404020162040e2"x
temp3: "20162040e2", diff: 4

translate (temp1, "00"x || "□", "□" || "00"x) enthält eine Eingabetabelle, die aus einer Leerstelle und einem Byte mit dem Wert 0 besteht. Da die Ausgabetabelle umgekehrt aus einem Byte mit dem Wert 0 und einer Leerstelle besteht, werden alle Leerstellen in temp1 in Nullbytes umgesetzt und alle Nullbytes in Leerstellen. temp2 enthält also anstelle gleicher Zeichen Leerstellen.

Die space-Funktion reduziert die Anzahl der Leerstellen auf die Anzahl, die durch den zweiten Parameter angegeben wird. temp3 entsteht aus temp2 dadurch, daß mit space (temp2, 0) alle Leerstellen ausgeblendet werden. temp3 besteht also nur noch aus Bytes, die unterschiedliche Zeichen in den beiden Quellzeichenfolgen repräsentieren. Die Anzahl der Unterschiede zwischen beiden Zeichenfolgen wird daher durch length(temp3) bezeichnet.

Die Langform der Problemlösung wird durch die Verschachtelung der einzelnen Schritte zur Kurzform zusammengefaßt. Dies kann man für normale Programme eigentlich nicht empfehlen, weil die Lesbarkeit dadurch stark eingeschränkt wird. In einer externen Funktion, die für den Anwender eine Blackbox darstellt, ist aber eine solche Schachtelung aus Gründen der Effektivität akzeptabel.

2. Unterschiede zweier Zeichenfolgen markieren

Aufgabe: Es soll eine Zeichenfolge ausgegeben werden, die an *den* Positionen das Markierungszeichen "^" enthält, die in den beiden Zeichenfolgen text1 und text2 unterschriedlich sind.

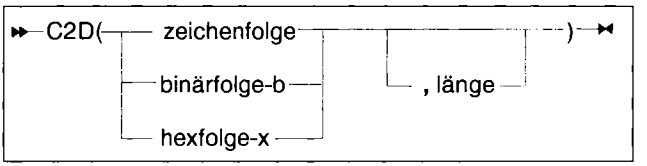
Wie in *Beispiel 1* wird mit bitxor (text1, text2) zuerst die Zeichenfolge temp1 gebildet, die statt gleicher Zeichen in den Quellzeichenfolgen Nullbytes enthält.

```
text1 = "Canetti, Elias"
text2 = "Canetti, Veza"
/* Langform: */
temp1 = bitxor (text1, text2)
markers = translate (temp1, "□",, "^")
/* Kurzform: */
markers = translate(bitxor(text1, text2), "□",, "^")
```

```
➔ temp1: "00000000000000000001309130073"x (ASCII)
  temp1: "00000000000000000020162000e2"x (EBCDIC)
  text1: "Canetti, Elias", text2: "Canetti, Veza", markers: □□□□□□□□□^^^□^
```

Die Anweisung translate (temp1, "□",, "^") enthält keine Eingabetabelle. Dann gilt standardmäßig der ganze Zeichensatz von 0-255 als Eingabetabelle, d. h. alle Zeichen werden übersetzt. Die Ausgabetabelle besteht nur aus einer Leerstelle. Sie wird daher bis zur Länge der Eingabetabelle durch das Füllzeichen "^" aufgefüllt. Das erste Byte der Eingabetabelle, das Nullbyte, wird durch das erste Byte der Ausgabetabelle, die Leerstelle, übersetzt. Da gleiche Zeichen durch Nullbytes repräsentiert werden, werden diese also in temp1 in Leerstellen übersetzt. Alle anderen Zeichen in temp1 repräsentieren unterschiedliche Zeichen und werden in das Markierungszeichen "^" übersetzt. Wenn man diese Zeichenfolge unter den Quellzeichenfolgen ausdruckt, werden die unterschiedlichen Zeichen beider Quellzeichenfolgen markiert.

c2d



Siehe auch: Konvertierfunktionen; Zweierkomplement.

Die eingebaute SAA-Funktion c2d (character t[w]o decimal = Zeichenformat in dezimales Format umwandeln) interpretiert die Eingabezeichenfolge als Binärzahl, wandelt sie in eine Ganzzahl um und liefert diese als Funktionsergebnis zurück. Die Eingabezeichenfolge kann eine Textzeichenfolge (zeichenfolge), eine Binärfolge (binärfolge-b) oder eine Hexfolge (hexfolge-x) sein.

Die Eingabezeichenfolge wird ggf. durch führende Nullen auf volle Bytes ergänzt.

Wenn die Ausgabezahl mehr Stellen als die *maximale Anzahl Ziffernstellen* hat, wird Fehler 40 gemeldet. Die maximale Anzahl Ziffernstellen wird mit der Anweisung numeric digits definiert und hat standardmäßig den Wert 9.

länge bezeichnet die Länge der Eingabezeichenfolge in Bytes:

- Wenn länge fehlt, wird die Eingabezeichenfolge als vorzeichenlose Binärzahl interpretiert.
- Wenn länge ungleich Null ist, wird die Eingabezeichenfolge als Binärzahl mit Vorzeichen und mit länge Bytes angesehen. Je nach dem Wert von länge wird die Eingabezeichenfolge ggf. mit führenden Nullbytes ("00"x) aufgefüllt (das Vorzeichenbit wird *nicht* expandiert!) oder links abgeschnitten. Wenn das linkeste Bit 0 ist, wird eine positive Ganzzahl zurückgegeben. Andernfalls wird die Eingabezeichenfolge als Binärzahl im Zweierkomplement betrachtet, und es wird eine negative Ganzzahl zurückgeliefert.
- Wenn länge 0 ist, wird als Funktionsergebnis 0 zurückgegeben.

Wenn die Eingabezeichenfolge aus der Nullzeichenfolge besteht, wird als Funktionsergebnis 0 zurückgeliefert.

binärfolge-b	Binärfolge von beliebiger Länge *mit* der Kennung B/b.
hexfolge-x	Hexfolge von beliebiger Länge *mit* der Kennung X/x.
länge	Nichtnegative Ganzzahl. Anzahl der Bytes der Eingabezeichenfolge, die berücksichtigt werden sollen. Die Eingabezeichenfolge wird ggf. durch Auffüllen mit führenden Nullbytes ("00"x) oder durch Abschneiden führender Bytes angepaßt. Das linkeste Bit der Eingabezeichenfolge wird als Vorzeichen angesehen (0 = positiv, 1 = negativ). **Standard:** Die Eingabezeichenfolge wird als vorzeichenlose Binärzahl interpretiert.
zeichenfolge	Zeichenfolge von beliebiger Länge.

Implementierungslimit: Die Eingabezeichenfolge darf aus höchstens 250 Bytes bestehen, die zur Bildung des Funktionsergebnisses herangezogen werden. Führende Vorzeichenbytes ("00"x und "ff"x) bleiben dabei unberücksichtigt. Bei Überschreitung des Implementierungslimits wird Fehler 40 gemeldet.

Wenn länge fehlt, darf also die Eingabezeichenfolge nicht länger als 250 Bytes sein. Ist länge angegeben, darf dieser Wert 250 nicht überschreiten.

Beispiele

REXX-Anweisungen	Erläuterung
zahl1 = c2d("c"x) zahl2 = c2d("0c"x) zahl3 = c2d("80"x) zahl4 = c2d("1100"b) zahl5 = c2d("a")	vorzeichenlose Binärzahl: Da die Länge fehlt, werden die Argumente als vorzeichenlose Binärzahlen interpretiert: zahl1, zahl2: 12, zahl3: 128, zahl4: 12 zahl5: 97 [ASCII]
zahl1 = c2d("c"x, 1) zahl2 = c2d("cc"x, 1) zahl3 = c2d("80"x, 1)	Binärzahl mit Vorzeichen, Länge = Argumentlänge (in Bytes): zahl1: "0c"x = "00001100"b. Linkestes Bit = 0, daher positiv. Ergebnis: 12. zahl2: "cc"x = "1100 1100"b. Linkestes Bit = 1, daher negativ. Zweierkomplement: "100"x - "cc"x = 256 - 204 = 52. Ergebnis: -52. zahl3: "80"x = "1000 0000"b. Linkestes Bit = 1, daher negativ. Zweierkomplement: "100"x - "80"x = 256 - 128 = 128. Ergebnis: -128.
zahl1 = c2d("cc"x, 2) zahl2 = c2d("80"x, 2)	Binärzahl mit Vorzeichen, Länge > Argumentlänge (in Bytes): zahl1: "00 cc"x = "0000 0000 1100 1100"b. Linkestes Bit = 0, daher positiv. Ergebnis: 204. zahl2: "00 80"x = "0000 0000 1000 0000"b. Linkestes Bit = 0, daher positiv. Ergebnis: 128.

REXX-Anweisungen	Erläuterung
zahl1 = c2d("c"x, 0) zahl2 = c2d("8 0c"x, 1) zahl3 = c2d("00 84"x, 1)	Binärzahl mit Vorzeichen, Länge < Argumentlänge (in Bytes): zahl1: " "x = " "b. Ergebnis: 0. zahl2: "0c"x = "0000 1100"b. Linkestes Bit = 0, daher positiv. Ergebnis: 12. zahl3: "84"x = "1000 0100"b. Linkestes Bit = 1, daher negativ. Zweierkomplement: "100"x - "84"x = 256 - 132 = 124. Ergebnis: -124.
zahl1 = c2d(" ") zahl2 = c2d(" "b) zahl3 = c2d(" "x)	Nullzeichenfolge umwandeln: zahl1, zahl2, zahl3: 0
zahl = c2d("f80"x, 0)	Länge = 0: zahl: 0
numeric digits 2 zahl = c2d("80"x)	Ausgabelänge > maximale Anzahl Ziffernstellen: Die Stellenzahl von zahl (= 128) überschreitet die maximale Anzahl Ziffernstellen, die mit numeric digits auf 2 festgelegt wurde. REXX meldet Fehler 40.
numeric digits 650 z = copies("7f"x, 250) zahl1 = c2d(z) zahl2 = c2d("f1"x || z) zahl3 = c2d("f1" || z, 250) zahl4 = c2d("f1" || z, 251)	Eingabelänge > Implementierungslimit: Die maximale Anzahl Ziffernstellen wird mit numeric digits auf 650 gesetzt, da eine Binärzahl von 250 Bytes in eine Ganzzahl mit mehr als 600 Stellen umgewandelt wird. z wird mit copies eine 250 Bytes lange Zeichenfolge zugewiesen. zahl1: Die Eingabezeichenfolge ist zulässig. zahl2: Führt zu Fehler 40, da die Eingabezeichenfolge 251 Bytes lang ist und damit das Implementierungslimit überschreitet. zahl3: Die Eingabezeichenfolge ist zulässig. Sie ist zwar 251 Bytes lang und liegt damit über dem Implementierungslimit, dank der Längenangabe von 250 werden aber nur 250 Bytes ausgewertet. zahl4: Führt zu Fehler 40, da die Länge von 251 Bytes das Implementierungslimit überschreitet.

Häufige Fehler

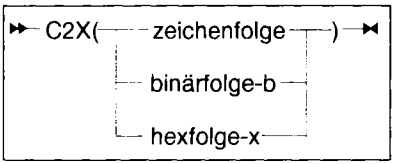
Länge fehlt. Wenn der Längenparameter fehlt, wird die Zeichenfolge als vorzeichenlose Binärzahl interpretiert:

```
c2d("80"x) → 128
```

Wenn dies nicht erwünscht ist, muß die Länge mit angegeben werden:

```
c2d("80"x, 1) → -128
```

c2x



Siehe auch: Konvertierfunktionen.

Die eingebaute SAA-Funktion c2x (character t[w]o hexadecimal = Zeichenformat in hexadezimales Format umwandeln) wandelt die Eingabezeichenfolge in eine Hexfolge um und liefert diese als Funktionsergebnis zurück. Die Eingabezeichenfolge kann eine Textzeichenfolge (zeichenfolge), eine Binärfolge (binärfolge-b) oder eine Hexfolge (hexfolge-x) sein.

Die Eingabezeichenfolge wird ggf. durch eine führende Null auf volle Bytes ergänzt. Die Ausgabezeichenfolge besteht aus doppelt so vielen Bytes wie die Eingabezeichenfolge.

Die Ausgabezeichenfolge wird *ohne* die Kennung X/x zurückgeliefert und enthält keine Leerstellen. Die Hexadezimalziffern A-F werden immer in Großbuchstaben ausgegeben.

Wenn die Eingabezeichenfolge aus der Nullzeichenfolge besteht, wird eine Nullzeichenfolge zurückgeliefert.

binärfolge-b	Binärfolge von beliebiger Länge *mit* der Kennung B/b.
hexfolge-x	Hexfolge von beliebiger Länge *mit* der Kennung X/x.
zeichenfolge	Zeichenfolge von beliebiger Länge. Die zurückgelieferte Hexfolge ist von dem verwendeten Zeichensatz abhängig (ASCII oder EBCDIC).

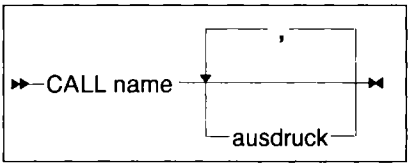
Beispiele

REXX-Anweisungen	Erläuterung
text2 = "11"b text3 = "44552"x hex1 = c2x("Säge") hex2 = c2x(text2) hex3 = c2x(text3)	Zeichenfolge umwandeln: hex1: "53846765" [ASCII] hex2: "03" hex3: "044552"
hex1 = c2x(""b) hex2 = c2x(""x) hex3 = c2x("")	Nullzeichenfolge umwandeln: hex1, hex2, hex3: ""

Anwendung

Siehe: storage, Anwendung.

call

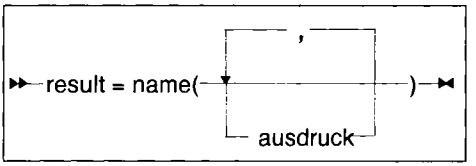


Siehe auch: call off/on; Funktion; Unterprogramm.

Die Anweisung call (aufrufen) ruft das Unterprogramm oder die Funktion name auf. name ist der Name eines internen oder externen Unterprogramms, einer internen oder externen Funktion oder einer eingebauten Funktion. Die Ausdrücke ausdruck,... sind beliebige Ausdrücke, die nach der Auswertung als Argumentfolgen von dem aufgerufenen Unterprogramm eingelesen werden können.

Unterprogramme und Funktionen werden unter den Stichwörtern *Funktion* und *Unterprogramm* ausführlich beschrieben.

Wenn das aufgerufene Unterprogramm mit return oder exit ein Ergebnis zurückgibt, wird dieses nach der Rückkehr aus dem Unterprogramm der speziellen Variablen result zugewiesen. Insofern ist ein call-Aufruf funktional identisch mit einer Zuweisung, in der der Variablen result das Ergebnis einer Funktion zugewiesen wird:



Wenn das aufgerufene Programm kein Ergebnis zurückgibt, ist die spezielle Variable result nach einem call-Aufruf undefiniert, während ein Funktionsaufruf in diesem Fall zu einem REXX-Fehler führt.

Hinweise:

- Als Verzweigungsziel sind die Markennamen ON und OFF nicht zulässig, da diese als Unterschlüsselwörter zum Ein- bzw. Ausschalten von Fehlerbedingungen reserviert sind.
- In einer interpret-Anweisung darf sich eine call-Anweisung nicht auf eine Marke in der zu interpretierenden Zeichenfolge beziehen, da Marken innerhalb einer interpret-Zeichenfolge nicht erlaubt sind. *Siehe:* interpret.

ausdruck	Beliebiger Ausdruck.
name	Symbol, das als Konstante angesehen wird und einen Markennamen darstellt, oder konstante Zeichenfolge in Großbuchstaben, die einen Markennamen darstellt.