Entwurf

Open Object Rexx 5.0
Kurzreferenz für Klassiker

Jochem Peelen

Stand: 02.02.2022

Zielsetzung

- Benutzern der Sprache soll eine kompakte Möglichkeit zum Nachschlagen oft benötigter Informationen geboten werden.
- Wer ooRexx nicht kennt, kann sich einen Überblick verschaffen.

Das Handbuch zu *Open Object Rexx 5.0* (Datei *rexxref.pdf*) geht auf 800 Seiten zu. Schon als ich noch beruflich mit dem klassischen Rexx arbeitete, schuf ich mir die Möglichkeit, die Syntax der am häufigsten genutzten, nicht-trivialen Sprachelemente rasch nachzuschlagen. Diese Hilfe habe ich mir 2019 für Version 5.0 ganz neu geschrieben.
Die Syntaxdiagramme sind hoffentlich einfach zu verstehen. Im Zweifelsfall kann auf Seite 52 nachgesehen werden. Das Inhaltsverzeichnis befindet sich schnell erreichbar am Ende.
Trotz des dicken Handbuches ist das Programmverhalten von Rexx darin nicht immer eindeutig beschrieben. Die Darstellung in dieser Kurzreferenz beruht auf meinen eigenen Tests *unter Windows* und erfolgt deshalb ohne Gewähr.

Warum „Klassiker"?

Beim klassischen Programmieren kommt man mit wenigen einfachen Regeln aus, vergleichbar den 19 Grundrechtsartikeln des Grundgesetzes. Die objektorientierte Arbeitsweise kommt mir demgegenüber wie das Bürgerliche Gesetzbuch mit seinen 2385 Paragrafen vor. *Welche der mindestens 11 Methoden mit dem Namen **CompareTo** habe ich denn gerade vor mir?* Eine „**Kurz**referenz" für objektorientiertes Programmieren ist undenkbar.
Hier wird deshalb auf objektorientierte Sprachteile verzichtet. Eine Anzahl klassisch benutzbarer Methodenaufrufe, die funktionale Lücken schließen, sind allerdings beschrieben. Eine weitere Ausnahme gilt für das hinzugekommene *stabile* Sortieren (Seite 37), das zum besseren Verständnis etwas eingehender dargestellt ist.
Auf meinem Notebook habe ich über 28000 selbst geschriebene Rexx-Zeilen im Einsatz. Ich weiß *sehr* zu schätzen, dass klassische Programme in *Open Object Rexx* unverändert weiter funktionieren.

Entwurf

Open Object Rexx 5.0 befindet sich leider immer noch in der Beta-Phase. Seit 2016 setze ich den jeweils aktuellen Stand unter Windows 8.1 und 10 ohne Probleme produktiv ein. Die *Rexx Language Association* (Webseite **rexxla.org**) arbeitet weiter an der endgültigen Version. Die zahlreichen *nix-Derivate (plus MacOS), die selbst für Trivialfunktionen wie die Abfrage der eigenen IP-Adresse mit überraschenden Unterschieden aufwarten, machen die Arbeit offensichtlich mühsam. Rund 50 *nix-Installationspakete existieren für ooRexx 4.2.0. Ganze zwei Pakete (32 und 64 Bit) decken dagegen Windows ab.
Die vorliegende Fassung der Kurzreferenz berücksichtigt die funktionalen Änderungen vom Februar 2021 an den auf Seite 21 beschriebenen Schleifenoptionen. Alles andere betrifft Schreibfehler, eindeutigere Formulierungen oder ist dem in pdfLaTeX 2021 etwas veränderten Seitenumbruch geschuldet. Neu sind die Abschnitte auf den Seiten 35 und ab Seite 42.

1 Zeichenketten-Funktionen

1.1 Informationen über Zeichenketten

Länge

```
                              n  =  length( — zkette — )


                            -- length('ooRexx')   ⇒    6
                            -- length('')         ⇒    0
```

Liefert die Länge von *zkette*, die bei einer leeren Zeichenkette 0 ist.

Datentyp prüfen

```
 flag = datatype( — zkette — , ┌─┬─ 'A' ─┬─┐ )   -- Alphanumeric    a-z, A-Z, 0-9
                               ├─┼─ 'M' ─┼─┤     -- Mixed case      a-z, A-Z
  0     falsch                 ├─┼─ 'L' ─┼─┤     -- Lowercase       a-z
  1     wahr                   ├─┼─ 'U' ─┼─┤     -- Uppercase       A-Z
                               ├─┼─ 'X' ─┼─┤     -- heXadecimal     0-9, a-f, A-F, ''
                               ├─┼─ 'B' ─┼─┤     -- Binary          0,   1,   ' ', ''
                               ├─┼─ 'O' ─┼─┤     -- lOgical         == 0 | == 1
                               ├─┼─ 'N' ─┼─┤     -- Numeric         Zahl
                               ├─┼─ 'W' ─┼─┤     -- Whole number    ganze Zahl z.B. 12, -2.0, 3E4
                               │       │ │     -- "nur Ziffern"   VERIFY() benutzen
                               │       │ │     --
                               ├─┼─ '9' ─┼─┤     -- 9digits         ganze Zahl <= 999999999 (9stellig)
                               ├─┼─ 'I' ─┼─┤     -- Internal        digits <= 9 (32bit)| <= 18 (64bit)
                               ├─┼─ 'S' ─┼─┤     -- Symbol          zulässiger Name oder Konstante
                               └─┴─ 'V' ─┴─┘     -- Variable        zulässiger Name
Alternativformat:

 datatype( ── zkette ── )    ⇒    NUM wenn Zahl │ CHAR für alles andere
```

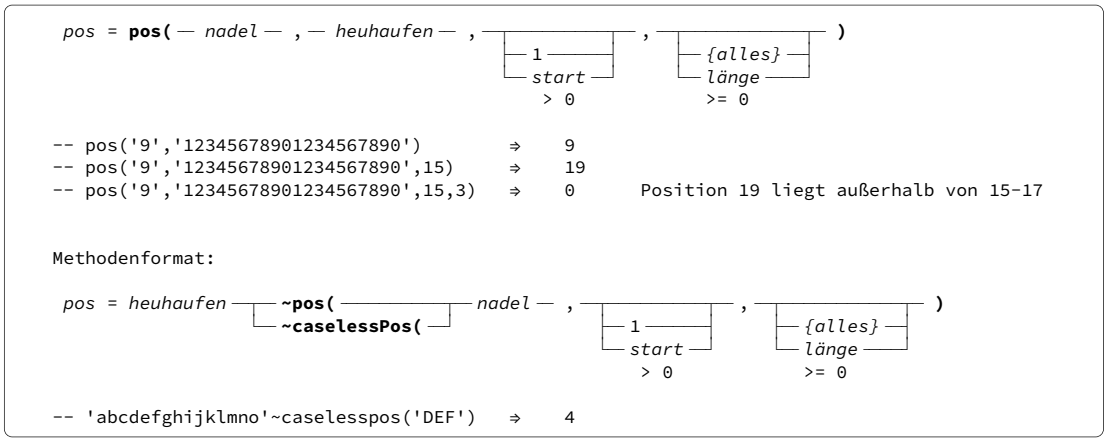
Liefert 1, falls *zkette* dem gewünschten Datentyp entspricht, sonst 0. Beim Alternativformat ohne Typangabe wird NUM zurückgegeben, falls *zkette* numerisch ist, sonst CHAR.

```
  pos = verify( — heuhaufen — , — byteliste — , ┌─────────┐ , ┌─────────┐ , ┌─────────┐ )
                                                ├─ 'N' ─┤   ├─ 1 ─────┤   ├─ {alle} ─┤
                                                └─ 'M' ─┘   └─ start ─┘   └─ länge ──┘
                                                              > 0           >= 0

  -- verify(4711,'0123456789')              ⇒    0     Zeichenkette besteht nur aus Ziffern
  -- verify(3.14,'0123456789')              ⇒    2     Position des ersten Fremdzeichens
  --
  -- verify('Marder','0123456789','M')      ⇒    0     Keine Ziffern enthalten
  -- verify('Leopard2A7','0123456789','M')  ⇒    8     Position der ersten Ziffer
  -- verify('Satzende. ','?!.;:','M')       ⇒    9     dto. erstes Interpunktionszeichen
```

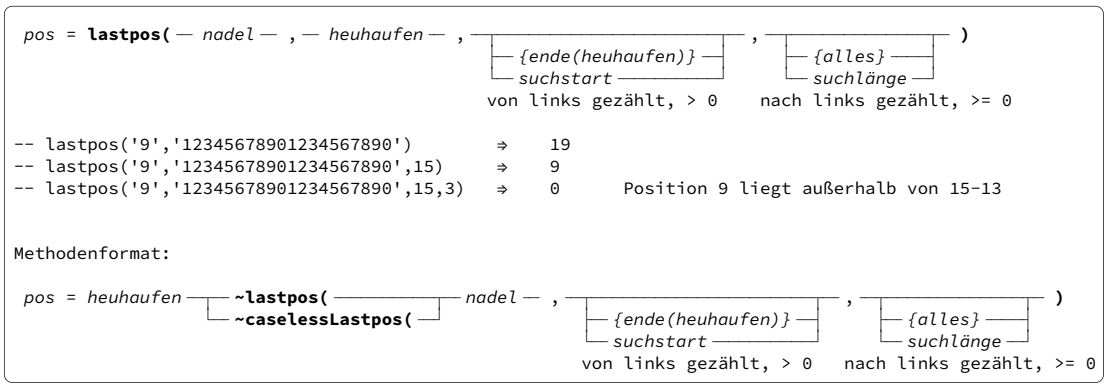
Im Modus **N**[omatch] wird 0 geliefert, falls in *heuhaufen* **nur** die Zeichen der *byteliste* vorkommen ODER wenn **heuhaufen** die leere Zeichenkette ist. Sonst wird die Position der ersten falschen Zeichens in *heuhaufen* geliefert.
Im Modus **M**[atch] wird 0 geliefert, wenn *heuhaufen* **kein** Zeichen der *byteliste* enthält. Sonst wird die Position des ersten Treffers in *heuhaufen* geliefert.

Mit *start* kann die Suche bei einer anderen Position als 1 beginnen. Die Anzahl der zu prüfenden Zeichen ist mit *länge* bestimmbar. Länge 0 liefert immer Ergebnis 0.

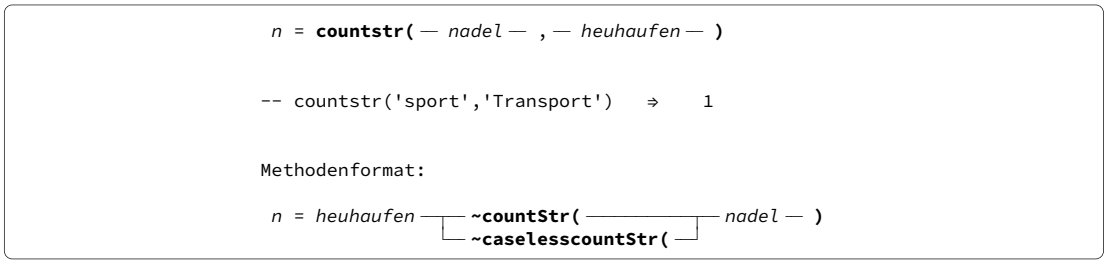
In Zeichenketten suchen



Liefert die Position des ersten Zeichens von *nadel* in *heuhaufen* oder 0 wenn *nadel* nicht in *heuhaufen* enthalten ist. Als Methode ist eine Version aufrufbar, die Groß- und Kleinbuchstaben[1] als identisch (caseless) betrachtet.



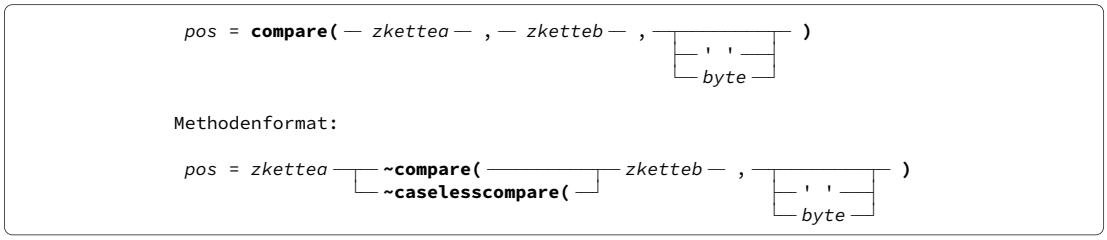
Arbeitet wie **pos()**, sucht jedoch am Ende von *heuhaufen* beginnend in Richtung Anfang. Die gelieferte Position ist aber vom Anfang aus –also links beginnend– gezählt.



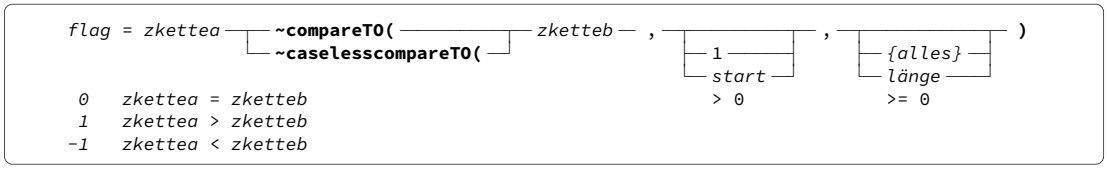
Zählt wie oft *nadel* vollständig in *heuhaufen* enthalten ist.

[1] Für ooRexx als im englischen Sprachraum entstandene Programmiersprache sind die deutschen Umlaute oder entsprechende Zeichen anderer Sprachen **keine** Buchstaben und erfordern besondere Maßnahmen. Das betrifft auch alle Sortierfunktionen.

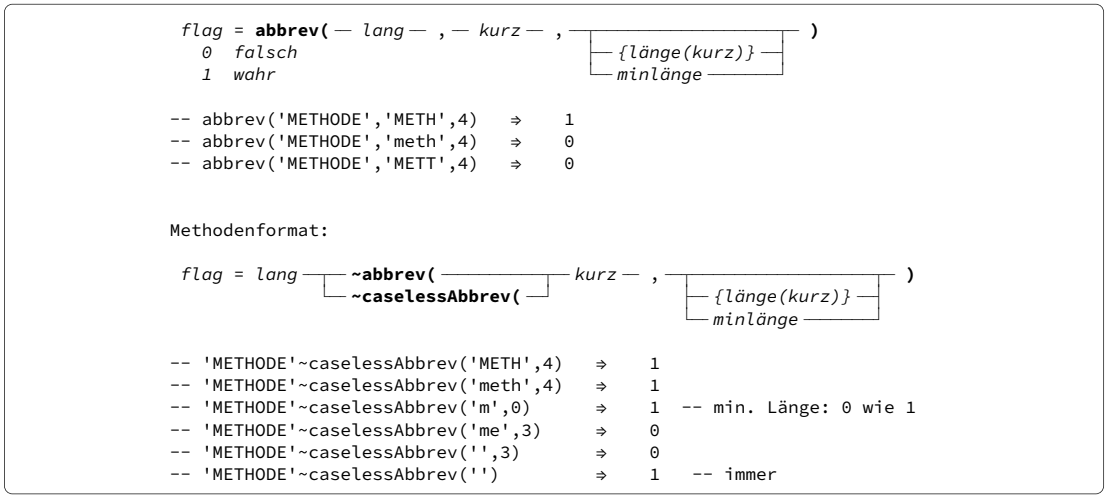
Zeichenketten vergleichen



Liefert 0 wenn beide Zeichenketten gleich sind, sonst die Position des ersten unterschiedlichen Zeichens. Ist eine Zeichenkette kürzer, wird sie vor dem Vergleich mit Zeichen *byte* (Vorgabe: Leerstelle) aufgefüllt.



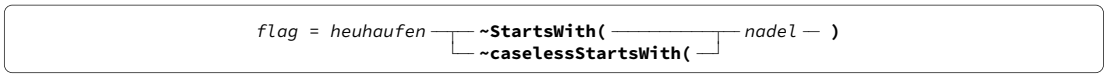
Durch 0, 1 oder -1 wird signalisiert, ob ein Unterschied zwischen den Zeichenketten besteht. Es erfolgt keine Längenanpassung. Kleiner oder größer ergibt sich aus der fiktiven Sortierfolge der Zeichenketten.
Durch Angabe von *start* kann eine andere Startposition als der Anfang für den Vergleich gewählt werden. Mit *länge* ist die Anzahl der zu vergleichenden Zeichen steuerbar. Bei Länge 0 wird immer Ergebnis 0 zurückgegeben.

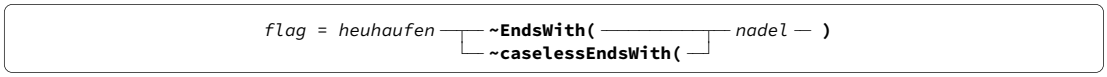


Liefert 1 wenn die Zeichen in *kurz* mit dem Anfang von *lang* übereinstimmen, sonst 0. Bei Angabe von *minlänge* wird 0 zurückgegeben, falls *kurz* kürzer ist.

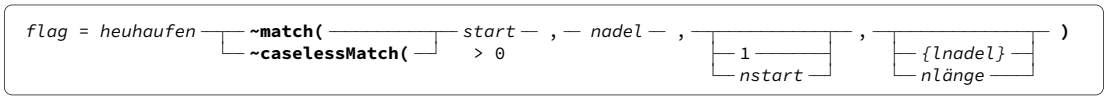
Zeichenketten-Teilstücke vergleichen

Laut Handbuch sind die sechs Methoden dieses Abschnitts besonders effizient, weil sie ohne Erzeugung neuer Zeichenketten-Objekte auskommen.



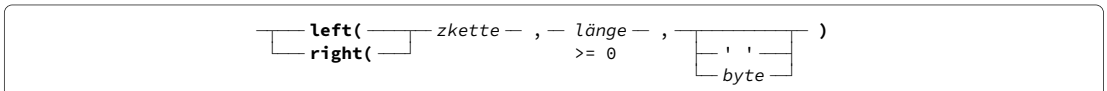


Beide liefern 1, wenn am Anfang/Ende von *heuhaufen* die Zeichenkette *nadel* steht, sonst 0.



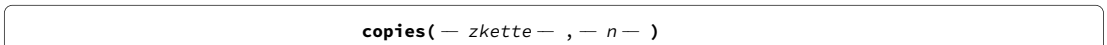
Liefert 1, wenn Zeichenkette *nadel* in *heuhaufen* vorkommt, sonst 0. Der Vergleich beginnt ab Spalte *start* von *heuhaufen*. Mit *nstart* und *nlänge* kann erreicht werden, dass nur ein Teil von *nadel* für den Vergleich benutzt wird.

1.2 Zeichenketten abschneiden oder verlängern



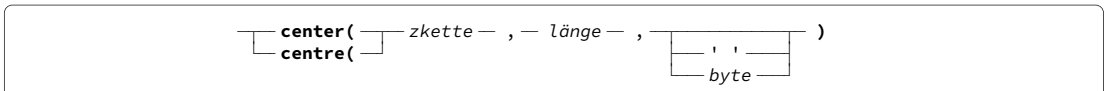

```
-- left('ooRexx',2)         ⇒    'oo'
-- left('ooRexx',12)        ⇒    'ooRexx      '
-- left('ooRexx',12,'_')    ⇒    'ooRexx______'
--
-- right('ooRexx',2)        ⇒    'xx'
-- right('ooRexx',12)       ⇒    '      ooRexx'
-- right('ooRexx',12,'_')   ⇒    '______ooRexx'
```

Liefert eine Zeichenkette der gewünschten *länge*, ausgehend vom linken oder rechten Ende von *zkette*. Es wird abgeschnitten oder bei Bedarf mit Zeichen *byte* aufgefüllt. **Hinweis:** Die bei Wortketten auf Seite 11 beschriebene Funktion **strip()** zum Löschen von führenden und nachfolgenden Zeichen kann zum Abschneiden ebenfalls in Frage kommen.




```
-- copies('-*-',3)   ⇒    '-*--*--*-'
-- copies('-*-',0)   ⇒    ''
-- copies('',5)      ⇒    ''
```

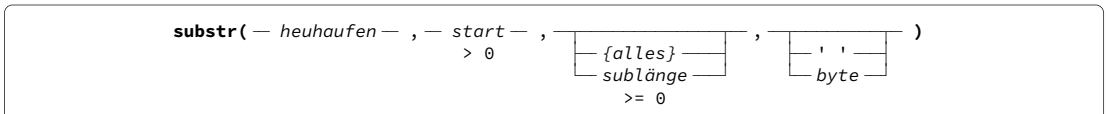
Liefert *n* Kopien von *zkette*.




```
-- center('abc',23)         ⇒    '          abc          '
-- center('abc',23,'-')     ⇒    '----------abc----------'
-- center(' abc ',23,'-')   ⇒    '--------- abc ---------'
-- center('abc',0)          ⇒    ''
```

Liefert eine innerhalb von *länge* zentrierte *zkette*, die abgeschnitten oder mit Zeichen *byte* aufgefüllt ist. Muss eine ungerade Zahl von Zeichen abgeschnitten/hinzugefügt werden, wird rechts ein Zeichen mehr abgeschnitten oder hinzugefügt.

1.3 Daten aus Zeichenketten extrahieren




```
-- substr('abcdefgh',4)      ⇒    'defgh'
-- substr('abcdefgh',4,3)    ⇒    'def'
-- substr('abcdefgh',12)     ⇒    ''        start > länge(heuhaufen)
-- substr('abcdefgh',12,3)   ⇒    '   '     bei expliziter sublänge
```

Liefert einen Teil von *heuhaufen*, der an Position *start* beginnt und *sublänge* Zeichen lang ist. Wird eine *sublänge* angegeben, die über das Ende von *heuhaufen* hinausgeht, wird mit Zeichen *byte* aufgefüllt.

Alternative: Schlüsselwort PARSE

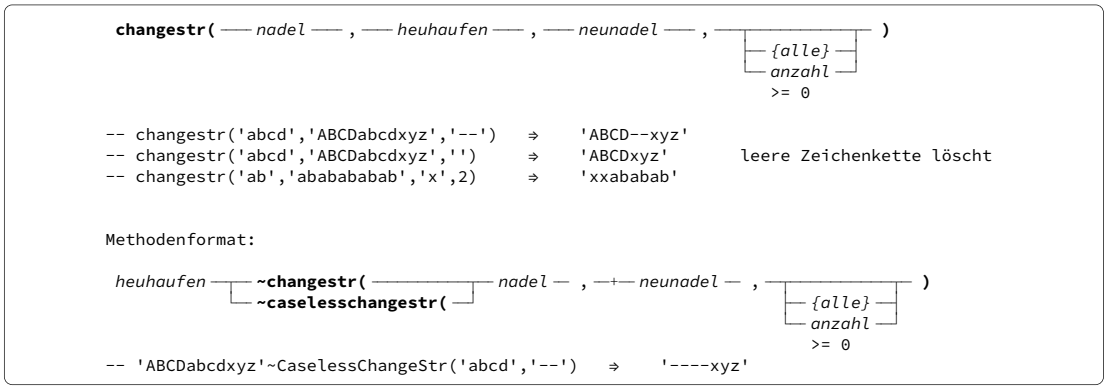
Das nachfolgende Schema kann nur eine kurze Gedächtnisstütze sein. Die vielfältigen Möglichkeiten von **parse**, darunter das –auch überlappende– Zerlegen in mehrere Zeichenketten in einem Schritt, sind in Kapitel 9 der *ooRexx Reference* beschrieben.

```
-- Format parse var  ... für Daten in Variablen:

--          ....+....1....+....2..
beispiel = '   eins zwei    drei '                  -- zu zerlegende Variable

 parse var  beispiel 4 name1 13 19 name2 21          -- positionsweise

say '>'name1'< >'name2'<'    ⇒      >eins zwei< >ei<

 parse var  beispiel . wort2 wort3 .                 -- wortweise (Punkt am Ende beachten)

say '>'wort2'< >'wort3'<'    ⇒      >zwei< >drei<

 parse var  beispiel 9 name1 . 17 name2              -- positions- und wortweise gemischt

say '>'name1'< >'name2'<'    ⇒      >zwei< >drei<

 parse var  beispiel 'ns ' name2 ' '                 -- anhand von Zeichenketten

say '>'name2'<'              ⇒      >zwei<

trenn1 = 'ns '
trenn2 = ' '
 parse var  beispiel (trenn1) name2 (trenn2)         -- dto. Zeichenketten in Variablen

say '>'name2'<'              ⇒      >zwei<

sp1 = 4
sp2 = 13
 parse var  beispiel =(sp1) name3 =(sp2)             -- dto. Positionen in Variablen

say '>'name3'<'              ⇒      >eins zwei<
```

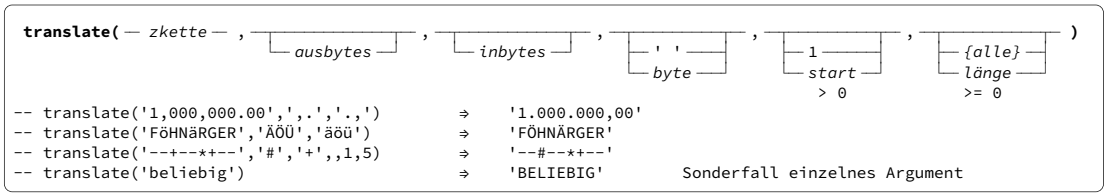
```
-- Format parse value  ...  with  ... für Zeichenketten und Funktionswerte:

 parse value — ' eins zwei drei ' — with — zerlegeschema wie oben
               — funktion(arg1,...,argn) —
```

1.4 Daten in Zeichenketten ändern

Umwandeln

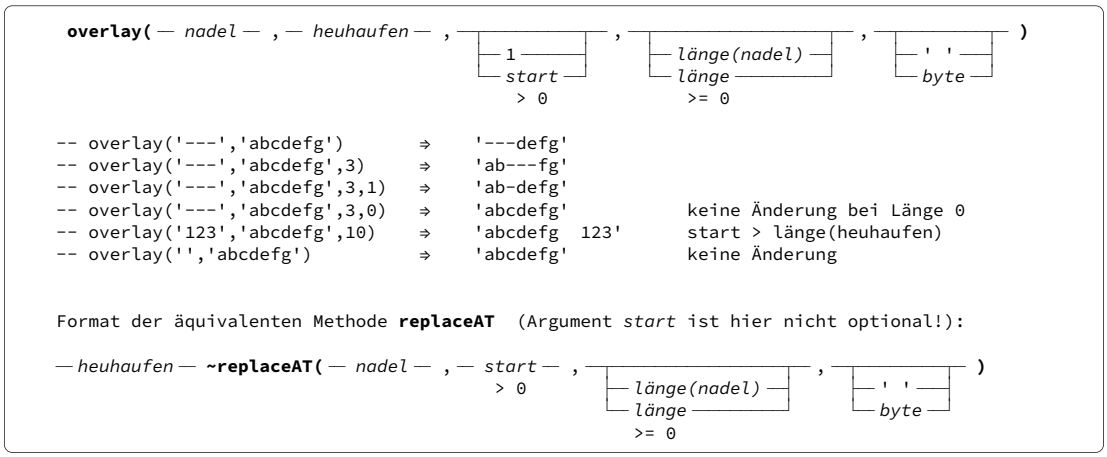


Liefert *heuhaufen*, in dem alle Vorkommen von *nadel* geändert sind in die Zeichenkette *neunadel*, die auch länger oder kürzer sein kann. Mit *anzahl* kann die Änderung auf die ersten Vorkommen von *nadel* beschränkt werden.

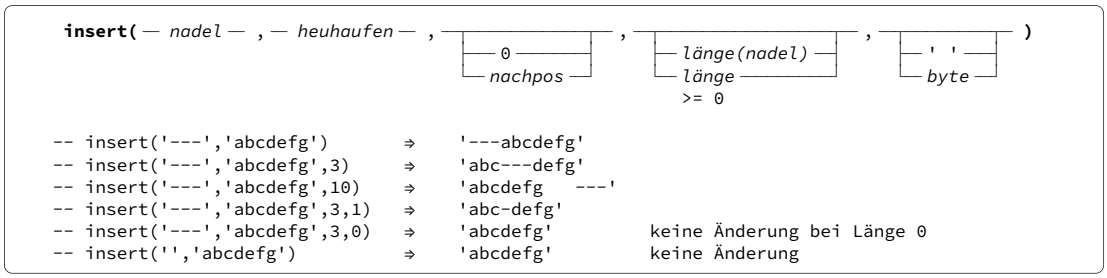


Liefert *zkette*, in der die Zeichen der Liste *inbytes* durch das jeweils in Liste *ausbytes* an derselben Position stehende Zeichen ersetzt sind.
Mit *start* und *länge* kann der bearbeitete Bereich von *zkette* begrenzt werden. Ist Liste *ausbytes* kürzer als *inbytes*, wird sie mit Zeichen *byte* aufgefüllt.

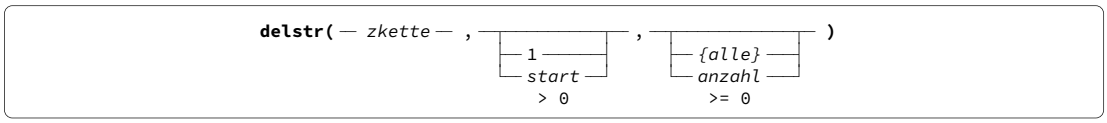
Überschreiben, Einfügen, Löschen



Liefert *heuhaufen*, der ab Position *start* überschrieben ist mit Zeichenkette *nadel*. Ist *länge* größer als *nadel*, wird mit Zeichen *byte* aufgefüllt, ebenso wenn *start* die Länge von *heuhaufen* übertrifft.

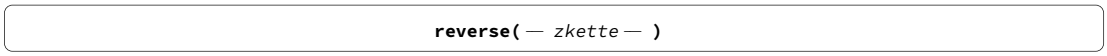


Liefert *heuhaufen*, in den hinter Position *nachpos* die Zeichenkette *nadel* eingefügt ist. Die rechts von *nachpos* stehenden Zeichen werden entsprechend nach rechts verschoben.

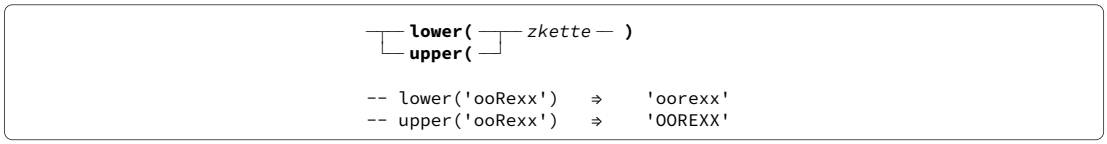


Liefert *zkette*, aus der *anzahl* Zeichen ab Position *start* gelöscht sind. Eventuell rechts vom Löschbereich verbliebene Zeichen werden nach links verschoben.

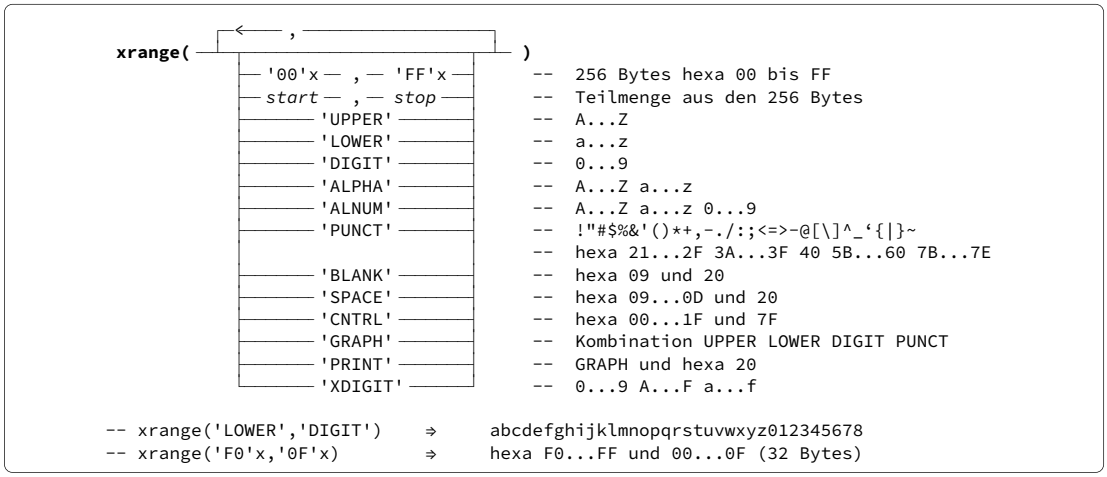
Sonstiges



Liefert die Zeichen von *zkette* in umgekehrter Reihenfolge *ettekz*.



Auch hier ist zu bedenken, dass nur die gewöhnlichen 26 Buchstaben bearbeitet werden.

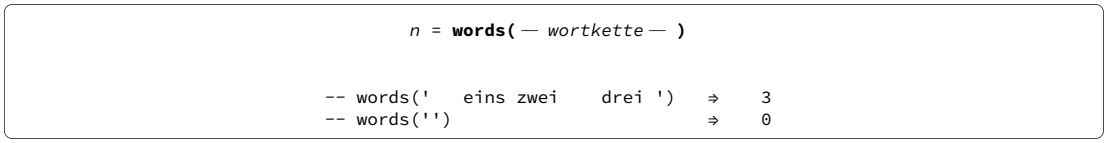


Liefert eine Zeichenkette mit den Bytes der gewünschten Klassen. Ist bei der Angabe von zwei einzelnen Bytes das Zeichen *start* größer als *stop*, besteht das Ergebnis aus den Bytes von *start* bis hexa `FF`, denen ohne Zwischenraum die Bytes von hexa `00` bis *stop* folgen.

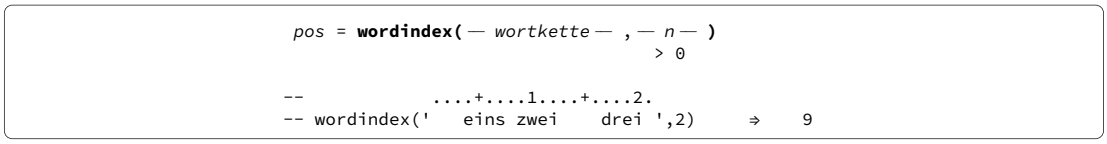
2 Wortketten-Funktionen

Wortketten enthalten **Leerstellen**[1] als Trennzeichen zwischen den „Wörtern“. Diese müssen jedoch nicht aus Buchstaben bestehen, sondern können alle Zeichen außer Leerstellen enthalten.

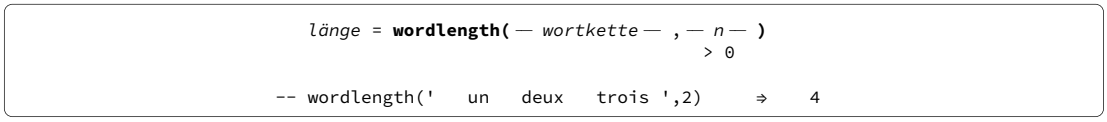
2.1 Informationen über Wortketten



Liefert die Anzahl der Wörter in *wortkette* oder 0 wenn es eine leere Zeichenkette ist.

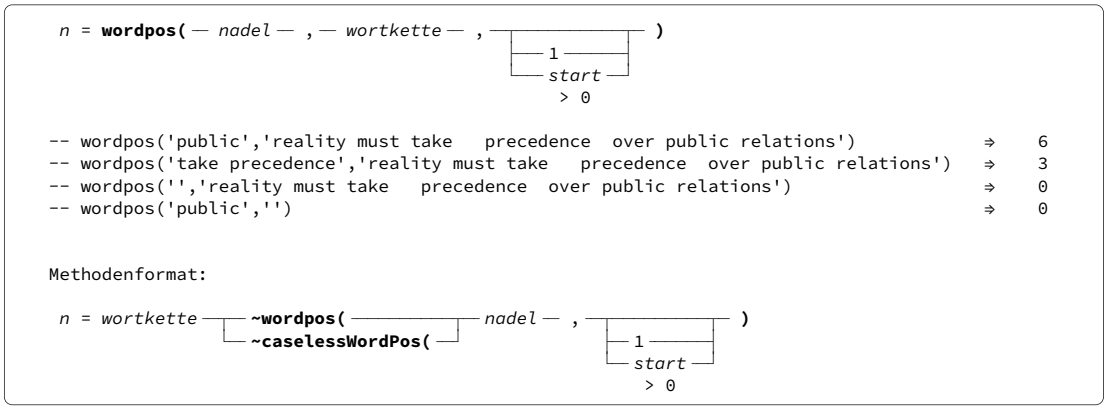


Liefert die Position in *wortkette* an der das *n*-te Wort beginnt.



Liefert die Länge des *n*-ten Wortes in *wortkette*.

2.2 In Wortketten suchen

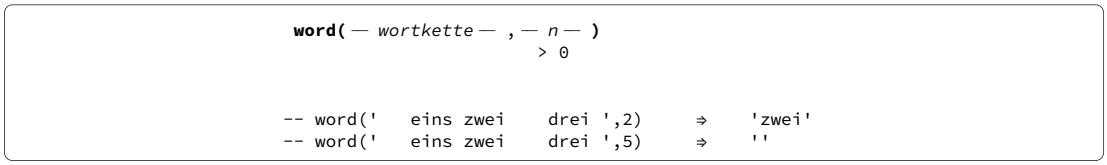


Liefert 0 falls *nadel* nicht in *wortkette* vorkommt, sonst die Nummer *n* des Wortes in *wortkette* an dem *nadel* beginnt. *nadel* kann ein einzelnes Wort oder eine Wortkette sein. Mit *start* kann die Suche bei einem anderen als dem ersten Wort von *wortkette* beginnen.

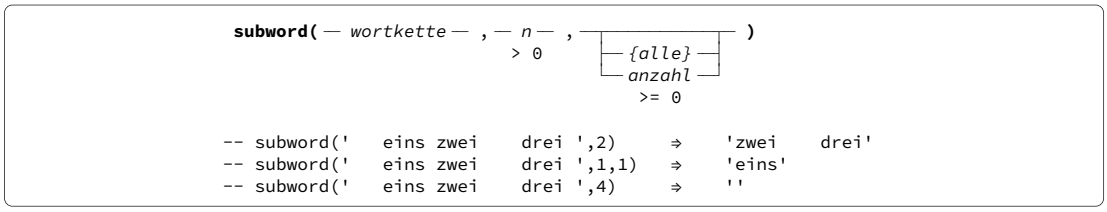
[1] Auch das Tabulatorzeichen (hexa 09) gilt in diesem Zusammenhang als Leerstelle, jedoch nicht die übrigen Zeichen der Klasse SPACE von Funktion **xrange** auf Seite 9.

0 wird immer geliefert, wenn ein oder beide Wortargumente leere Zeichenketten sind oder *start* größer als die Wortzahl in *wortkette* ist.

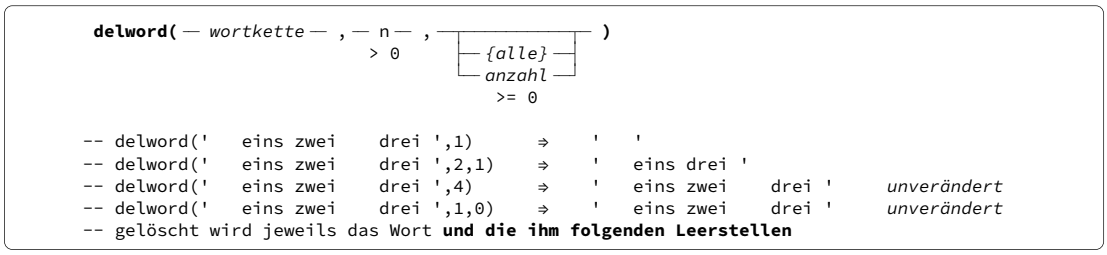
2.3 Wortketten-Daten lesen oder löschen



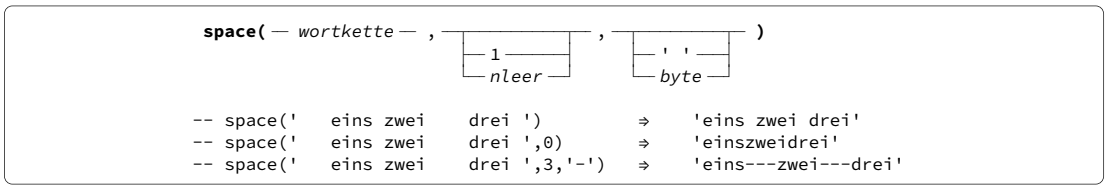
Liefert das *n*-te Wort aus *Wortkette*.



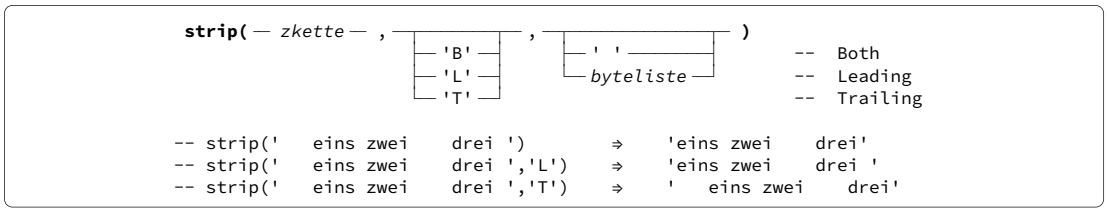
Liefert *anzahl* Wörter aus *wortkette*, beginnend mit dem *n*-ten Wort.



Liefert den Rest von *wortkette*, nachdem daraus *anzahl* Wörter gelöscht sind, beginnend mit dem *n*-ten Wort.



Liefert *wortkette* aufgefüllt mit je *nleer* Leerstellen in den Wortzwischenräumen. Anstelle der Leerstellen kann auch ein beliebiges Zeichen *byte* verwendet werden. **space()** bearbeitet die Leerstellen **innerhalb** einer Wort- oder Zeichenkette; vergleiche **strip()**.



Liefert *zkette* mit gelöschten Leerstellen am Anfang (Leading), Ende (Trailing) oder Anfang und Ende (Both). Es kann auch eine *byteliste* von zu löschenden Zeichen vorgegeben werden. Ist *byteliste* eine leere Zeichenkette, wird nichts gelöscht. **strip()** löscht die Zeichen **an den Enden** einer Wort- oder Zeichenkette; vergleiche **space()**.

3 Datum und Uhrzeit

In Rexx ist für jeden Tag seit Montag, den 01. Januar 0001 dessen laufende Nummer (Format *Basedate*) verfügbar, so dass Fristen zwischen Daten einfach feststellbar sind. Es wird der Gregorianische Kalender verwendet, so als ob er damals schon gegolten hätte. Allerdings werden die seit 1972 eingeschobenen bisher 27 Schaltsekunden (zuletzt 2017) ignoriert.

```
-- Format für aktuelles Datum (aufgerufen am 1. Mai 2019)

 date( ─┬──────────┬┬───────────────────────────┬─ )
        │          │└─ , ── , ── , ─ trenn ─┘       Standard─
        │          │                                Trennzeichen
        ├─ 'N' ─┤      ⇛    1 May 2019                ' '
        ├─ 'I' ─┤      ⇛    2019-05-01                '-'
        ├─ 'S' ─┤      ⇛    20190501                  ''
        ├─ 'E' ─┤      ⇛    01/05/19                  '/'
        ├─ 'O' ─┤      ⇛    19/05/01                  '/'
        ├─ 'U' ─┤      ⇛    05/01/19                  '/'
        ├─ 'M' ─┤      ⇛    May
        ├─ 'W' ─┤      ⇛    Wednesday
        ├─ 'D' ─┤      ⇛    121                  (1...366 Tage)
        ├─ 'B' ─┤      ⇛    737179               (Basedate: Tage seit 01.01.0001)
        │       │
        │       │      --   Um 00:00 Uhr des     1. Mai 2019   waren abgelaufen:
        ├─ 'F' ─┤      ⇛    63692265600000000    Mikrosekunden seit 01.01.0001 00:00
        └─ 'T' ─┘      ⇛    1556668800           Sekunden      seit 01.01.1970 00:00

-- Beispiel für anderes Trennzeichen:
--      date('E',,,'.')   ⇛    01.05.19
```

Zeitformat **Full** sind *Mikro*sekunden seit dem 01. Januar 0001 und **Ticks** die Sekunden seit dem 01. Januar 1970. Für Rechnungen damit muss die Einstellung NUMERIC DIGITS auf mindestens 18 bzw. 12 erhöht werden. Siehe dazu Seite 22.

Datumsformat umwandeln

```
-- Formatumwandlung eines Datums zwischen Montag, den 01.01.0001 und Freitag, den 31.12.9999

 date( ─┬─────────┬─ , ─ indatum ─ , ─┬─ 'N' ──┬─ , ─┬────────────┬─ , ─┬──────────────┬─ )
        ├─ 'N' ─┤                     ├─ 'I' ─┤      └─ trenn ─┘        └─ itrenn ─┘
        ├─ 'I' ─┤                     ├─ 'S' ─┤      in Ausgabe         Trennzeichen in
        ├─ 'S' ─┤                     ├─ 'E' ─┤                         indatum falls
        ├─ 'E' ─┤                     ├─ 'O' ─┤                         nicht Standard
        ├─ 'O' ─┤                     ├─ 'U' ─┤
        ├─ 'U' ─┤                     ├─ 'B' ─┤
        ├─ 'M' ─┤                     ├─ 'F' ─┤
        ├─ 'W' ─┤                     └─ 'T' ─┘
        ├─ 'B' ─┤                  Format von indatum
        ├─ 'D' ─┤
        ├─ 'F' ─┤
        └─ 'T' ─┘
      Ausgabeformat

--  Umwandlung Basistag in Kalenderdatum:
--  date('I',737179,'B')            ⇛    2019-05-01
--  date('I',737179,'B','.')        ⇛    2019.05.01
--  date('I',737179,'B','')         ⇛    20190501

--  Wochentag als Ziffer: 0 = Montag ... 6 = Sonntag
--  date('B','2019-05-01','I') // 7  ⇛    2  also Mittwoch
```

Liefert das Datum *indatum* im gewünschten Ausgabeformat. Dabei kann mit *trenn* ein vom Standardformat abweichendes Trennzeichen in der Ausgabe verwendet werden. *trenn* muss genau ein Zeichen lang sein, ohne Buchstabe oder Ziffer zu sein. Auch die leere Zeichenkette ist zulässig, um Trennzeichen im Ausgabeformat zu unterdrücken.
Die Angabe von *itrenn* ist nur erforderlich, wenn *indatum* Trennzeichen enthält, die vom Standard für das jeweilige Format abweichen.

Uhrzeit

```
-- Format für aktuelle Zeit der Computeruhr:

time( ──┬──────────┬── )
        │          │        ⇒  zum Beispiel:
        ├── 'N' ───┤        ⇒  09:46:37
        ├── 'L' ───┤        ⇒  09:46:37.397000
        ├── 'C' ───┤        ⇒  9:46 a.m
        ├── 'H' ───┤        ⇒  9                  0...23        Stunden seit 00:00
        ├── 'M' ───┤        ⇒  586                0...1439      dto. Minuten
        ├── 'S' ───┤        ⇒  35197              0...86399     dto. Sekunden
        │          │
        ├── 'F' ───┤        ⇒  636923007970000000  Mikrosekunden seit 01.01.0001 00:00
        ├── 'O' ───┤        ⇒  7200000000          Mikrosekunden lokale Differenz zu UTC
        ├── 'T' ───┤        ⇒  1556668800          Sekunden      seit 01.01.1970 00:00
        │          │
        ├── 'E' ───┤        ⇒  152.114000          Elapsed: aktuelle Stopuhrsekunden
        └── 'R' ───┘        ⇒  152.114000          Reset: startet Stopuhr neu bei 0.000000
```

Liefert die aktuelle Uhrzeit in dem gewünschten Format

Laufzeiten messen

Beim ersten Aufruf in einem Programm liefern **time(E)** oder **time(R)** immer 0 und starten eine interne Stopuhr. Später liefert **time(E)** deren aktuellen Stand. **time(R)** tut dasselbe, startet die Zeit aber jedesmal neu bei 0. In Windows (getestet: XP, 7, 8.1 und 10) beträgt die Auflösung der Zeitangaben eine tausendstel Sekunde. Die gelieferten Nachkommastellen 4 bis 6 sind also immer 0.

Uhrzeitformat umwandeln

```
                    -- Formatkonvertierung einer Uhrzeit von
                    -- 00:00:00.000000 bis 23:59:59:999999

                     time( ──┬──────────┬── , ─ inzeit ─ , ──┬── 'N' ──┬── )
                             ├── 'N' ───┤                    ├── 'L' ──┤
                             ├── 'L' ───┤                    ├── 'C' ──┤
                             ├── 'C' ───┤                    ├── 'H' ──┤
                             ├── 'H' ───┤                    ├── 'M' ──┤
                             ├── 'M' ───┤                    ├── 'S' ──┤
                             ├── 'S' ───┤                    ├── 'F' ──┤
                             ├── 'F' ───┤                    └── 'T' ──┘
                             └── 'T' ───┘                Format von  inzeit
                          Ausgabeformat

                    -- time('N','9:46am',C)              ⇒    09:46:00
                    -- time('N',63692300797000000,'F')   ⇒    09:46:37
```

Liefert die Uhrzeit *inzeit* im gewünschten Ausgabeformat.

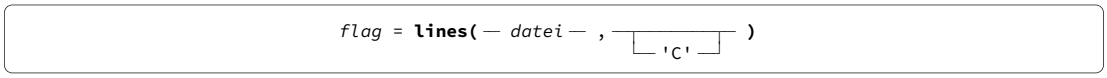
Auf die Objektklassen **DateTime** und **TimeSpan** mit zusammen rund 110 Methoden für die Arbeit mit Daten, Uhrzeiten und Zeitspannen sei hingewiesen.

4 Dateien und Verzeichnisse

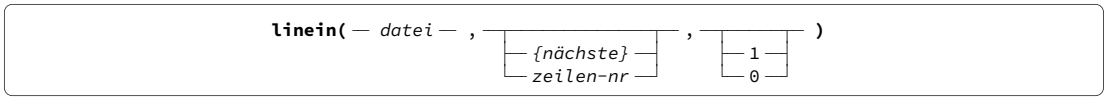
Als Erbe aus der Lochstreifenzeit sind (*nix- und) Windows-Dateien ein linearer Zeichenstrom ohne Zeilenstruktur. Logische Zeilen in den Daten werden mit dem Zeilenende-Zeichenpaar hexa 0D0A (für Wagenrücklauf und Zeilenvorschub des Fernschreibers) angezeigt. Zum Zählen der Zeilen muss das Betriebssystem die ganze Datei lesen.

4.1 Dateien zeilenweise lesen und schreiben

Das Zeilenende-Zeichenpaar (hexa 0D0A) wird hierbei vom Dateisystem abgestreift oder angehängt, ist also für das Programm nicht existent.[1]



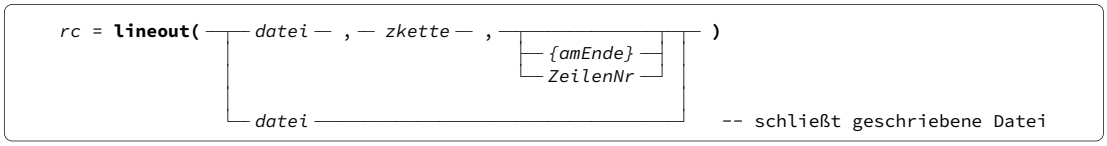
Liefert 1, falls Daten aus *datei* gelesen werden können, in allen anderen Fällen 0. Argument **C** bewirkt einen Scan der Datei und liefert die Anzahl der noch ungelesenen Zeilen.



Liefert, beginnend mit Zeile 1 von *datei*, die jeweils nächste Zeile. Nur je 1 Zeile pro Aufruf ist möglich. Konzeptionell sind die Zeilen fortlaufend ab 1 nummeriert. Es kann direkt Zeile *zeilen-nr* gelesen werden. Die nächste *zeilen-nr* darf dann auch Richtung Dateianfang liegen. Ist das dritte Argument 0, wird ohne zu lesen nur die Leseposition geändert. Und zwar so, dass der darauf **folgende** Aufruf von **linein(***datei***)** den Inhalt von *zeilen-nr* liefert.

```
-- Beispiel einer Schleife zum vollständigen Lesen einer Datei

datei = 'C:\demo\muster.txt'
do i=1 while lines(datei) = 1     -- oder auch:   do i=1 for lines(datei,'C')
     ...
     zeile = linein(datei)
     ...
end i
```

Aufruf von **linein()** nach dem Lesen der letzten vorhandenen Dateizeile führt zum endlosen Warten. Daher ist die Benutzung von **lines()** wichtig.

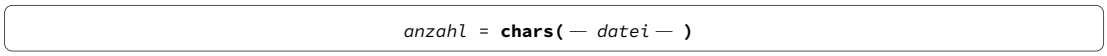


Hängt *zkette* als neue Zeile an das Ende von *datei* an. Wird *ZeilenNr* angegeben, überschreibt **lineout()** ab dem Anfang dieser Zeile die existierenden Daten, ohne Rücksicht ob alte und neue Länge übereinstimmen. *ZeilenNr* muss innerhalb der existierenden Daten liegen oder die erste neue Zeile am Dateiende sein. Im Fehlerfall ist *rc* 1, sonst 0.

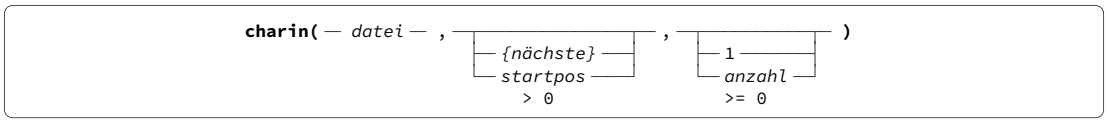
[1] Der von Rexx am Mainframe bekannte Befehl **EXECIO** steht in vereinfachter Form ebenfalls zur Verfügung. Er ist im Kapitel HOSTEMU der Datei *rexxextensions.pdf* beschrieben.

4.2 Dateien block- oder zeichenweise lesen und schreiben

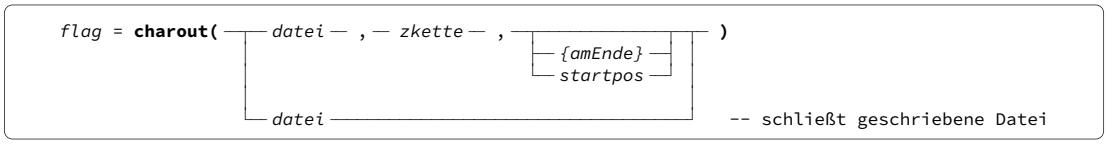
Hier hat das Zeilenende-Zeichenpaar (hexa 0D0A) keine Steuerfunktion und wird wie alle anderen Bytes auch behandelt. Ein Block kann minimal 1 Zeichen lang sein. Zeilenweise Struktur der Daten ist hier auch möglich, sofern die Daten *Zeilen identischer Länge* bilden, zum Beispiel eine Tabelle mit 244 Zeichen pro Zeile. Alle gelesenen und geschrieben Blocks müssen dann 244 Zeichen lang sein oder ein ganzes Vielfaches davon.



Liefert die *anzahl* der verfügbaren –noch nicht gelesenen– Zeichen in *datei* oder 0.

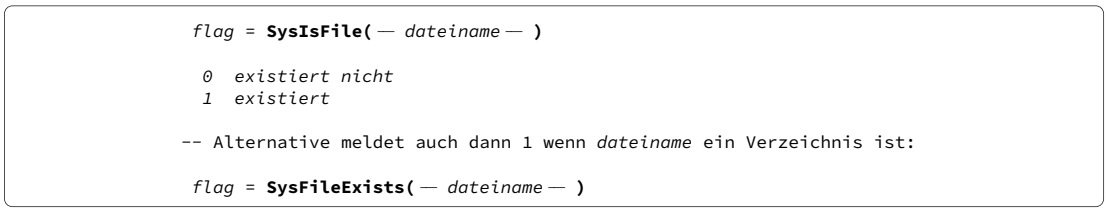


Liefert *anzahl* Zeichen aus *datei*. Sind weniger Zeichen vorhanden, wird eine entsprechend kürzere Zeichenkette geliefert. Mit *startpos* kann als Lesebeginn ein anderes als das erste ungelesene Zeichen der Datei bestimmt werden. Wird zugleich *anzahl* gleich 0 gesetzt, erfolgt nur die Änderung der Leseposition und statt Daten wird eine leere Zeichenkette geliefert.



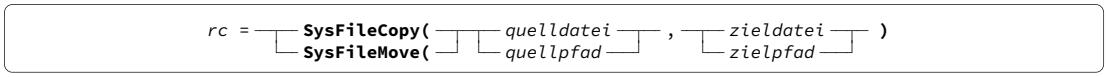
Hängt *zkette* an das Ende von *datei* an oder überschreibt ab *startpos*, falls angegeben.

4.3 Mit vorhandenen Dateien arbeiten



Liefert 1, falls die Datei existiert, sonst 0. Die ältere Funktion **SysFileExists()** hat dieselbe Funktion, unterscheidet aber nicht zwischen Dateien und Verzeichnissen. Beide akzeptieren keine Platzhalter wie * oder ?.

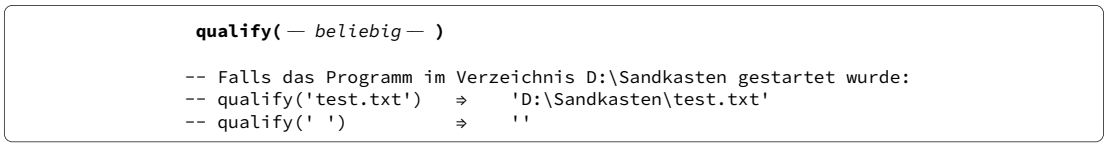
Eignet sich zum Löschen ohne Fehlermeldung, gleichgültig ob die Datei existiert oder nicht. Ausser 0 und 2 sind auch andere Windows Returncodes möglich (siehe Seite 50).



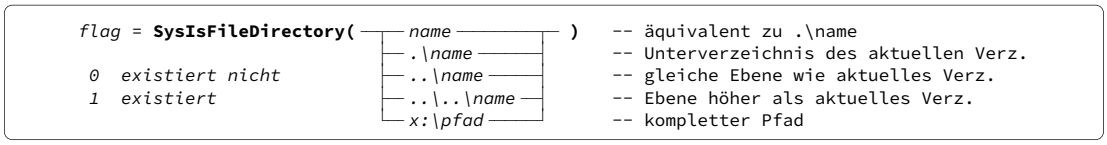
Liefert 0 bei erfolgreichem Kopieren (Copy) oder Umsetzen (Move) der Datei. Im Fehlerfall wird der entsprechende Windows Returncode geliefert. **Umbenennen** einer Datei erfolgt durch „Umsetzen“ (Move) innerhalb desselben Verzeichnisses.

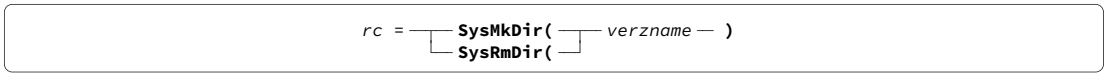
4.4 Mit Verzeichnissen arbeiten

Ohne Pfadangaben werden Dateien im „aktuellen“ Verzeichnis (Ordner) gesucht und angelegt. Dies ist das Verzeichnis, in dem der Befehl zum Start des ooRexx-Programms eingegeben wurde. Beim Start über ein Icon ist es das Verzeichnis, das im Feld „Ausführen in:“ der Icon-Eigenschaften eingetragen wurde.



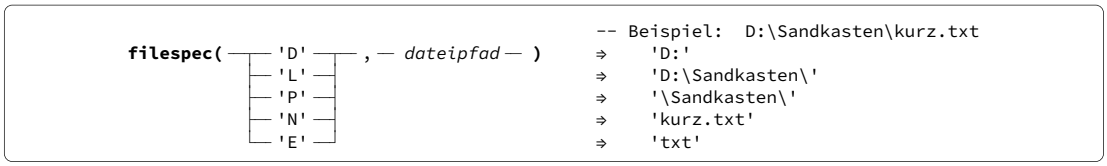
Liefert den kompletten Pfad des aktuellen Verzeichnisses. Zeichenkette *beliebig* wird dabei ungeprüft wie ein Dateiname angehängt, außer sie besteht nur aus einer Leerstelle oder leeren Zeichenkette.





Liefert 0 bei Erfolg, anderenfalls einen Windows Returncode (S. 50).

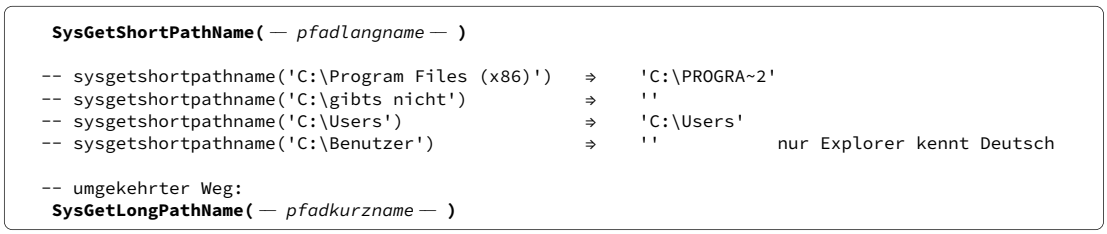
Elemente des Pfadnamens lesen



Ist das gewünschte Element im *dateipfad* nicht enthalten, wird die leere Zeichenkette geliefert.

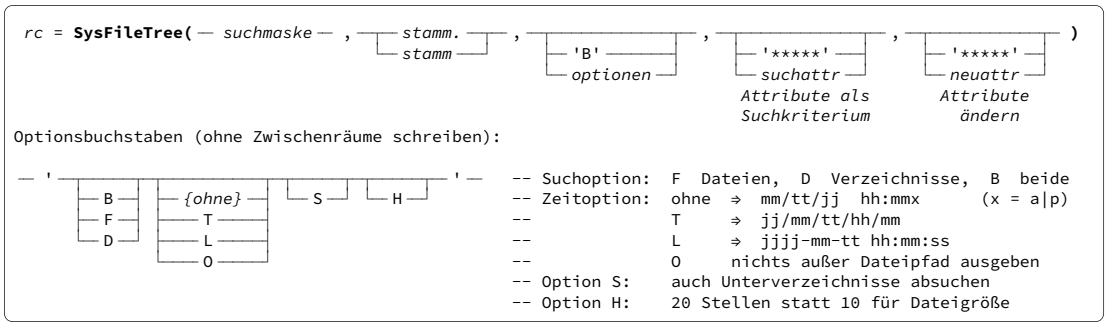
Kurze Pfadnamen

Leerstellen in Pfadnamen lassen einige ältere Programme ins Stolpern kommen. Dies kann durch Verwendung der Windows-internen kurzen Pfadnamen umgangen werden.



In Verzeichnissen nach Dateien suchen

Beim Suchen in Verzeichnissen können die Namen der vorhandenen Dateien und Unterverzeichnisse gelesen werden, sowie jeweils der Zeitstempel, die Größe in Bytes und die fünf Attribute (Archive, Directory, Hidden, Read-Only, System, kurz ADHRS).



Enthält *suchmaske* keinen Pfad, zum Beispiel nur `*.exe`, wird im aktuellen Verzeichnis gesucht. Soll die Suche auf Dateien mit bestimmten Attributen eingeschränkt werden, sind diese als eine 5 Bytes lange Zeichenkette *suchattr* anzugeben, wobei * für beliebig, **+** für gesetzt und **-** für nicht gesetzt (gelöscht) steht.
Die Optionsbuchstaben für die Ausgabe des Resultats sind ohne Zwischenräume anzugeben, zum Beispiel 'FLS'. Das Resultat wird als Stammvariable[2] *stamm.* zurückgeliefert, wobei *stamm.0* die Anzahl der gefundenen Treffer enthält. Ist diese Zahl größer als 0, enthalten die Elemente *stamm.1 ...* die Daten zu jedem Treffer. Auch wenn der Name ohne Punkt angegeben wird, erhält der Stammname einen Punkt, da dieser zur Syntax gehört. Folgende Ausgabeformate sind möglich:

```
Auswirkung der Zeitoption auf die Position von Bytezahl, Attributen, Dateipfad in der Ausgabe:

....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
ohne:              |<19         |<31   |<38
10/05/19  11:17p      509440  A----  C:\program files (x86)\oorexx\ooDialog.exe

Option T:         |<17         |<29   |<36
19/10/05/23/17      509440  A----  C:\program files (x86)\oorexx\ooDialog.exe

Option L:              |<22         |<34   |<41
2019-10-05 23:17:48       509440  A----  C:\program files (x86)\oorexx\ooDialog.exe
2015-05-09 16:25:06            0  -D---  C:\program files (x86)\THE\doc     -- Directory-Beispiel
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8

Option O:
C:\program files (x86)\oorexx\ooDialog.exe
```

Bei Option H beginnen Attribute und Pfadnamen wegen der zusätzlichen Breite des Größenfeldes um 10 Positionen weiter rechts.
Mit der 5 Bytes langen Zeichenkette *neuattr* können die vorhandenen Attribute geändert werden, wobei * für unverändert lassen, **+** für setzen und **-** für löschen steht. Das von der Funktion zurückgelieferte Resultat enthält dann schon die geänderten Attribute.

Dateien öffnen und schließen

Beim ersten Zugriff wird eine Datei automatisch geöffet; es ist kein besonderer Befehl nötig. Alle geöffneten Dateien werden erst am Ende des Programms geschlossen. Explizit kann jede Datei mit folgendem Aufruf geschlossen werden:

```
rc = stream( datei , 'C' , 'CLOSE' )
```

Bei Erfolg wird die Zeichenkette `READY:` geliefert oder die leere Zeichenkette, falls *datei* nicht offen war. Im Fehlerfall enthält *rc* den Windows Returncode; siehe Seite 50. Zum Schreiben geöffnete Dateien können auch mit **lineout()** und **charout()** geschlossen werden, wie oben beschrieben.

[2] Stammvariable sind ab Seite 30 beschrieben.

5 Programmschleifen

Schlüsselwort **do** wird in Rexx sowohl für einfache **do ... end** Blocks als auch Schleifen benutzt, indem in der Zeile mit **do** entsprechende Steueranweisungen folgen. Seit ooRexx 3.2 können Schleifen auch mit **loop ... end** definiert werden.

Einfache Schleifen

```
-- "endlose" Schleifen können mit leave verlassen werden, siehe auch while und until

 do forever                 |    äquivalent:    loop
 ...                        |                   ...
 if ... then leave          |                   if ... then leave
 ...                        |                   ...
 end                        |                   end
```

Die Instruktionen innerhalb der Schleife werden unbegrenzt durchlaufen. Das Programm muss mittels einer der weiter unten besprochenen Möglichkeiten **leave**, **while** oder **until** die Schleife verlassen. Damit können alle ooRexx Schleifenarten verlassen werden. In allen folgenden Beispielen könnte anstelle von **do** auch **loop** stehen.

```
do  ganzzahl
...     >= 0
...
end

-- do 25   ⇒    Schleife wird 25 mal durchlaufen
-- do 0    ⇒    Schleife wird übersprungen
```

Hier steht Vor Beginn die Anzahl der Durchläufe fest. Dabei muss *ganzzahl* eine positive ganze Zahl sein. Ist sie 0, wird die Schleife ignoriert.

```
do for  ganzzahl
...       >= 0
...
end
```

Vorstehende Schreibweise wird vom Interpreter akzeptiert und ausgeführt, ist aber syntaktisch nicht richtig, da **for** eigentlich nur rechts von anderen Bedingungen vorgesehen ist, wie in den nächsten Abschnitten dargestellt.

Iterationsvariable

Beim Schleifenstart wird der Iterationsvariable *iter* ein Startwert zugeordnet. Dieser muss zwar numerisch sein, aber weder unbedingt ganzzahlig noch positiv. Nach jedem Durchlauf wird zu *iter* die **by**-Zahl *inkr* addiert. *inkr* darf auch negativ sein.
iter ist während eines Schleifendurchlaufs änderbar und damit die Schleife steuerbar. *iter* bleibt auch nach dem Schleifenkontext erhalten und kann wie jede andere Variable benutzt werden.

```
-- to und for werden vor Beginn jedes Schleifendurchlaufs getestet
-- Nach Ende jedes Schleifendurchlaufs wird zu iter die Zahl inkr addiert

 do ... iter = start ─┬──────────────────┬─┬─────────────────────┬─┬──────────────────┬─...
                      └─ to ─ endwert ───┘ └─ by ──┬── 1 ────┬──┘ └─ for ─ anzahl ───┘
                                                   └─ inkr ──┘

-- do i=1 to 5           ⇒    Schleife wird mit i=1 ... 5 durchlaufen
-- do i=5 to 1 by -1     ⇒    Schleife wird mit i=5 ... 1 durchlaufen
-- do i=3 to 5 by 0.25   ⇒    Schleife wird mit i=3, 3.25, 3.50 ... 5.00 durchlaufen
-- do i=1 to 5 for 3     ⇒    Schleife wird mit i=1 ... 3 durchlaufen (for stoppt)
-- do i=1 to 3 for 5     ⇒    Schleife wird mit i=1 ... 3 durchlaufen (to stoppt)
-- do i=1                ⇒    "endlose" Schleife mit i=1, 2, 3 ...
-- do i=2 by 2           ⇒    "endlose" Schleife mit i=2, 4, 6 ...
```

to und **by** gehören zusammen. Wird **by** nicht angegeben, hat *inkr* den Wert 1. **by** ohne **to** ergibt eine endlose Schleife, die zum Beispiel mit **leave** beendet werden muss.
for ist unabhängig vom Inhalt der Iterationsvariable und legt die Höchstzahl der Durchläufe fest. Änderungen der Variable *anzahl* nach dem Schleifenstart haben keinen Einfluss. Bei der Programmentwicklung kann **for** mit einer angemessen großen *anzahl* zum Beispiel als Vorkehrung gegen eine endlose Schleife benutzt werden.

```
-- Die Iterationsvariable kann das zugehörige end markieren
-- (optional, aber sehr empfehlenswert)

zhl = 8
 do  x=1 to zhl
 ...
      do  y=0  for zhl
      ...
           do  z=1 to 3
           ...
           end z
      ...
      end y
 ...
 end x
 say x z  ⇒    9 4   -- Iterationsvariable bleiben außerhalb der Schleifen benutzbar
```

Der Name der Iterationsvariable kann an das zugehörige **end**-Schlüsselwort angehängt werden, um das Schleifenende zu markieren. Werden Schleifen ineinander geschachtelt, verbessert das die Übersichtlichkeit sehr. Der Name kann auch an **leave** und **iterate** (siehe weiter unten) angehängt werden. Der Rexx Interpreter benötigt diese Markierung nicht.

Schleifensteuerung durch logische Vergleiche

Mit *entweder* **while** *oder* **until** am Ende der **do**-Zeile kann jede Schleife beendet werden. Ist **while** zu Beginn eines Durchlaufs erfüllt, läuft die Schleife weiter. Ist **until** am Ende eines Durchlaufs erfüllt, endet die Schleife.
Es gelten dieselben, auf Seite 47 dargestellten Vergleichsoperatoren wie für **if** und **when**. Bei logischen Verknüpfungen ist die korrekte Programmierung des beabsichtigten Verhaltens der Schleife nicht immer ganz einfach:

```
-- while wird vor Beginn jedes Schleifendurchlaufs getestet

                 ┌←  , ─────────┐
 do  ...  while ─┴─ bedingung ──┴──

-- do ... while anz <= 10              ⇒    Stop wenn Variable anz > 10 wird
-- do ... while anz <= 10 , flag \= 1  ⇒    Stop wie vor oder wenn flag gleich 1 wird
```

Solange die **while**-Bedingung erfüllt ist, wird die Verarbeitung fortgesetzt. Bei mehreren Bedingungen müssen alle erfüllt sein. Die erste nicht erfüllte Bedingung stoppt die Schleife.

```
-- until wird am Ende jedes Schleifendurchlaufs getestet

                 ┌←  , ──────┐
 do  ...  until ─┴─ bedingung ─┴──

-- do ... until anz >  10              ⇒    Stop wenn Variable anz > 10 wird
-- do ... until anz >  10 , flag = 1   ⇒    Stop wenn anz > 10 und  flag = 1
```

Die Schleife stoppt, wenn die **until**-Bedingung erfüllt ist. Bei mehreren Bedingungen müssen alle erfüllt sein, um die Schleife zu stoppen.

Abfolge der Prüfung auf Schleifenende

```
Vor Beginn jedes Schleifendurchlaufs:

1. Stop wenn die Iterationsvariable iter größer als der to  endwert ist.
2. Stop wenn der Zähler ganzzahl überschritten ist.
3. Stop wenn der Zähler for  anzahl überschritten ist.
4. Stop wenn die while Bedingung nicht mehr erfüllt ist.

5. Falls counter angegeben, Variable zähler um 1 erhöhen.
6. Instruktionen der Schleife durchlaufen.

Am Ende jedes Schleifendurchlaufs und sofort nach iterate :

7. Stop wenn die until Bedingung erfüllt ist.
8. Iterationsvariable iter entsprechend by  inkr erhöhen.
9. weiter mit 1.
```

In Schritt 8 wird die Iterationsvariable *iter* am Schleifenende immer hochgezählt, auch wenn die folgenden Schritte 1 bis 3 vor Beginn des nächsten Durchlaufs die Schleife beenden. Daher hat *iter* am Schluss meist den Wert, der einem zusätzlichen Durchlauf entspricht (zum Beipiel 9 bei **i=1 to 8**). Nur wenn die Schleife durch eine **until**-Bedingung oder mit **leave** verlassen wird, ist *iter* gleich der Anzahl Durchläufe (im Beispiel 8).

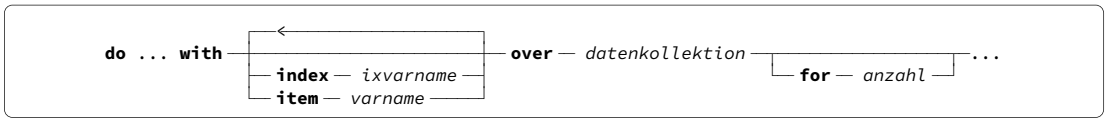
5.1 Neue Schleifenarten für Datenkollektionen

In ooRexx 5.0 sind 13 Datenkollektionen[1] definiert. Dafür gibt es jetzt 2 Schleifenarten, die hier am Kollektionstyp **Array** erklärt werden. Es **muss nicht bekannt sein, wieviele Elemente** die Kollektion enthält. **Lücken in der Indexfolge** (unbesetzte Elemente) sind zulässig.
Beim Start dieser Schleifen wird der Status der Datenkollektion festgehalten. Änderungen sind möglich, bleiben aber innerhalb der noch laufenden Schleife unsichtbar.
Nach dem Ende existieren die Schleifenvariablen *varname* und *ixvarname* weiter und haben den Inhalt des letzten Durchlaufs.

```
do ... — varname — over — datenkollektion ─┬──────────────────┬─...
...                                        └─ for — anzahl ─┘
...
end  varname          ── Anhängen von varname an das zugehörige end ist optional
```

[1] Eine heißt **Stem**, verhält sich aber nicht völlig identisch zur klassischen Stammvariable von Seite 30.

Die Schleife wird einmal für jedes in *datenkollektion* vorhandene, besetzte Element durchlaufen. Dabei erhält Variable *varname* bei jedem Durchlauf den Inhalt des aktuellen Elementes. Nur bei Kollektionen vom Typ *Ordered* (zum Beispiel Array) ist diese Reihenfolge streng nach aufsteigendem Index. Bei anderen Kollektionstypen ist sie nicht vorhersehbar.
Auch hier kann *varname* an das zugehörige **end**-Schlüsselwort angehängt werden, um das Schleifenende zu markieren. Schleifenbeispiele sind auf den Seiten 38, 41 und 45 gezeigt.



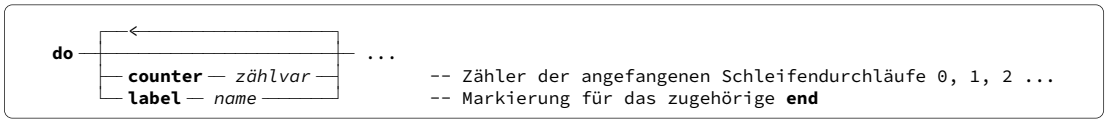
Diese Schleifenart wird ebenfalls einmal für jedes in *datenkollektion* vorhandene, besetzte Element durchlaufen. Falls verwendet, ist die Reihenfolge von **index** und **item** beliebig. Sofern für den jeweiligen Kollektionstyp ein Index existiert –beim Array ist es die jeweilige Zählnummer– wird der aktuelle Wert in Variable *ixvarname* geladen.
Der Inhalt des aktuellen Elementes wird, wenn angefordert, in *varname* geladen. Das folgende Beispiel für Kollektion `feld` ist äquivalent zum vorigen Schleifentyp:

```
do with item beispiel over feld
    say beispiel
end
```

Bei dieser Schleifenart kann ein Name hinter `end` nur bei Benutzung des im nächsten Abschnitt beschriebenen Schlüsselworts **label** verwendet werden.

5.2 Zusätzliche Anweisungen zur Steuerung

Diese gelten für alle Schleifenarten.



Wenn benutzt, müssen diese Optionen unmittelbar hinter **do** in beliebiger Reihenfolge stehen.
counter bewirkt, dass Variable *zählvar* mit 0 initialisiert wird. Jeder begonnene Schleifendurchlauf, auch wenn er mit **leave** oder **iterate** abgebrochen wird, erhöht Variable *zählvar* um genau 1. Das unterscheidet sie von der Iterationsvariable, die beliebige Schrittweiten haben kann und auch innerhalb der laufenden Schleife änderbar ist. Variable *zählvar* kann vom Programmm gelesen werden. Änderungen überschreibt Rexx aber vor Beginn des nächsten Durchlaufs.
Mit **label** wird ein *name* definiert, der auch beim zugehörigen **end**-Schlüsselwort verwendbar ist. So können auch bei Schleifen ohne Iterationsvariable die zusammengehörenden **do** und **end** kenntlich gemacht werden.

Schlüsselwörter leave und iterate

Mit **leave** wird die gerade aktive Schleife sofort beendet. Die Verarbeitung geht dann mit der Programmzeile hinter dem zugehörigen **end**-Schlüsselwort weiter.
Mit **iterate** wird der aktuelle Schleifendurchlauf abgebrochen und sofort der nächste begonnen. Eine vorhandene **until**-Bedingung zum Stop der Schleife wird jedoch ausgeführt.
Sind Schleifen ineinander geschachtelt und die **end**-Schlüsselwörter mit Variablennamen (Iterationsvariable, Label) versehen, können auch aus einer inneren Schleife heraus äußere Schleifen gesteuert werden. Zum Beispiel geht nach **leave x** die Verarbeitung mit der Programmzeile hinter **end x** weiter.

6 Rechnen

Auf einem gewöhnlichen 2.2 GHz Notebook braucht Rexx 1.1 Sekunden für die Berechnung einer 1000 m langen Flugbahn mit einem Runge-Kutta-Nyström Verfahren vierter Ordnung bei 0.001 Sekunden Schrittweite. Das ist schneller als die 2.85 Sekunden Flugzeit in der Realität. Für eine interpretierte Sprache rechnet Rexx sehr zügig.
Rexx bietet bei Bedarf Rechnen mit nahezu beliebiger Stellenzahl, begrenzt nur durch den Speicherplatz (RAM). Als Beispiel diene die Anzahl der mit 256 Bit möglichen Kombinationen. Bei den voreingestellten **9** signifikanten Ziffern wird 2^{256} ausgegeben als:

```
say 2**256   ⇒   1.15792089E+77
```

Um die Präzision auf 80 Ziffern zu erhöhen, genügt der Befehl:

```
numeric digits 80
```

und die gesuchte Zahl wird vollständig angezeigt:

```
say 2**256   ⇒   115792089237316195423570985008687907853269984665640564039457584007913129639936
```

Das dürften nur wenige Programmiersprachen in dieser Einfachheit bieten. Hinzu kommt, dass neben der mitgelieferten mathematischen Bibliothek **rxmath** (16 Stellen) auch eine externe Funktionsbibliothek[1] existiert, die Logarithmus, Exponential- und Winkelfunktionen bei Bedarf mit ebenfalls **nahezu beliebiger Genauigkeit** liefert. Für Rechenversuche zur Auswirkung erhöhter Genauigkeit ist Rexx deshalb sehr gut geeignet.

6.1 Wie Rexx mit Zahlen umgeht

Die Gleitkomma-Arithmetik von IEEE 754, die übrigens auf das klassische Rexx zurückgeht, wird hier mit der Voreinstellung von 9 signifikanten Stellen benutzt. Das bedeutet konkret:

- Jede im Programmcode stehende oder aus Daten gelesene Zahl ist für Rexx erst einmal eine Zeichenkette.
- Soll gerechnet werden, wird diese Zeichenkette intern in eine Zahl nach IEEE 754 umgewandelt und auf 9 signifikante Stellen gerundet, zum Beispiel:

  ```
  1.1234512345     ⇒   1.12345123
  3210.1234512345  ⇒   3210.12345
  ```

- Das Ergebnis jeder Rechnung wird ebenfalls auf 9 Stellen gerundet und dann wieder als Zeichenkette[2] ausgegeben.
- Falls sich vor dem Komma mehr als 9 Stellen ergeben, erfolgt die Ausgabe in Exponentialdarstellung. Mit der Funktion **format()** kann allerdings für jede Zahl die Exponentialdarstellung erzeugt werden.

Zu jedem Programmzeitpunkt kann per Schlüsselwort `numeric digits` anstelle von 9 jede andere Genauigkeit eingestellt werden.

[1] **rxm.cls** ist auf `rosettacode.org` frei verfügbar; siehe Seite 25.
[2] Soweit möglich, speichert Rexx intern beide Darstellungsformen um Konvertierungen einzusparen.

6.2 Operationen und Funktionen

Für die Beispiele gilt wieder `numeric digits 9`, falls nicht anders angegeben.

```
5 + 3          ⇒     8                   -- Addition
5 - 3          ⇒     2                   -- Subtraktion
5 * 3          ⇒     15                  -- Multiplikation
5 / 3          ⇒     1.66666667          -- Division
5 / 3          ⇒     1.666666666666667   -- Division bei numeric digits 16

5 // 3         ⇒     2                   -- Divisionsrest, siehe aber  modulo
12.8 // 2.5    ⇒     0.3
5 %  3         ⇒     1                   -- ganzzahliger Quotient
12.8 % 2.5     ⇒     5
5 ** 3         ⇒     125                 -- Exponent ganzzahlig oder Null
5 ** -3        ⇒     0.008

5**20          ⇒     9.53674316E+13      -- Exponentialdarstellung
5**-20         ⇒     1.048576E-14
5E2 + 0        ⇒     500                 -- entspricht 5 * 10**2
-5E2 + 0       ⇒     -500
```

```
rest = dividend — ~modulo( — divisor — )

  5~modulo(3)      ⇒    2
  (-5)~modulo(3)   ⇒    1
  -5~modulo(3)     ⇒   -2   --  falsch da als -(5~modulo(3)) ausgewertet
  zahl = -5
  zahl~modulo(3)   ⇒    1
```

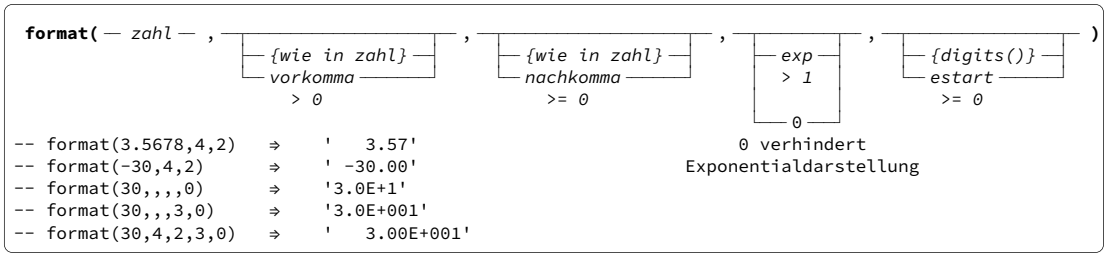
Die Tilde in einer Programmzeile bindet enger als das Minuszeichen. Deshalb müssen negative Zahlen in Klammern gesetzt werden, um die richtige Auswertung des Ausdrucks zu erhalten. Dies ist nicht notwendig, wenn die Zahl in einer Variable steht, weil dann kein Minuszeichen in der Programmzeile auftaucht.

```
zahl ─┬─ ~ceiling ─┬─        -- nächste ganze Zahl in Richtung +unendlich
      └─ ~floor   ─┘          -- nächste ganze Zahl in Richtung -unendlich

  (2.12)~ceiling     ⇒    3
  (3)~ceiling        ⇒    3
  (-2.12)~ceiling    ⇒   -2
  -2.12~ceiling      ⇒   -3    -- falsch da als -(2.12~ceiling) ausgewertet
  x = -2.12
  x~ceiling          ⇒   -2

  (2.12)~floor       ⇒    2
  (3)~floor          ⇒    3
  (-2.12)~floor      ⇒   -3
```

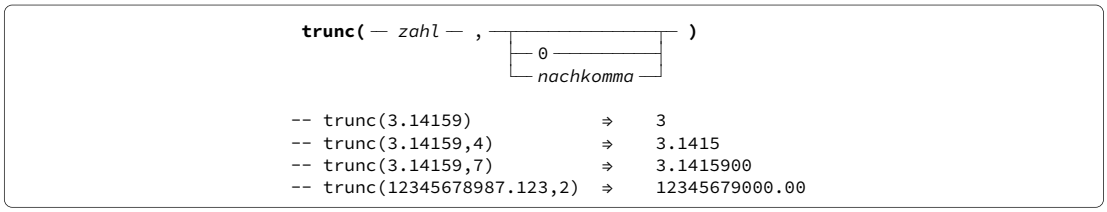
```
digits( ── )                  --  liefert aktuellen numeric digits Wert

abs( — zahl — )               --  Absolutwert von zahl

sign( — zahl — )              --  1 oder 0 oder -1

                              --  liefert aus der Liste von Zahlen
┌─ max( ─┬─ ← , ─┬─ zahl ─┬─ ) --  die größte
└─ min( ─┘       └────────┘    --  die kleinste
```

Die Anzahl der Zahlen im Funktionsaufruf, deren größte oder kleinste gesucht werden soll, ist nur durch den Speicherplatz begrenzt.

Zahlen formatieren



Liefert *zahl* gerundet auf *nachkomma* Kommastellen. Vor dem Komma ist Platz für *vorkomma* Ziffern, einschließlich einem eventuellen Minuszeichen. Reicht der Platz vor dem Komma nicht, gibt es einen Programmfehler.
Mit *estart* gleich Null kann Exponentialdarstellung unabhängig vom Zahlenwert erzwungen werden. Ansonsten bestimmt die Stellenzahl des ganzzahligen Teils von *zahl* in Verbindung mit **numeric digits** den Punkt der Umschaltung.
Die Anzahl der Ziffern im Exponenten richtet sich nach *exp*. Reicht diese Breite nicht, gibt es einen Programmfehler. Null an dieser Stelle verhindert vollständig die Benutzung der Exponentialdarstellung, auch wenn *estart* gleich Null ist. Fehlt *exp*, stehen im Exponenten nur soviele Ziffern wie nötig.



Liefert *zahl* abgeschnitten auf *nachkomma* Ziffern hinter dem Komma. Sind weniger vorhanden, wird mit Nullen aufgefüllt. Das letzte gezeigte Beispiel kommt zustande, weil vor der Verarbeitung die Zahl auf (hier) 9 signifikante Stellen gerundet wird und das Ergebnis nie Exponentialdarstellung hat.

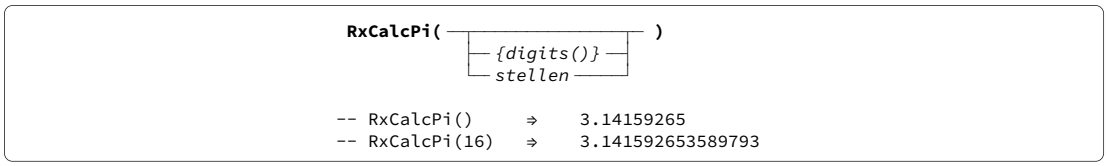
6.3 Mitgelieferte Bibliothek RXMATH

Mit Rexx wird eine mathematische Bibliothek auf der Basis von C-Bibliotheksfunktionen installiert (Datei `rxmath.dll`). Um sie in einem Rexx-Programm zu nutzen, ist **hinter** der letzten Programmzeile folgende Direktive einzufügen:

```
::REQUIRES rxmath LIBRARY

-- ersetzt die vor Version 4.0 notwendige Aktivierungsform:
-- call RxFuncAdd 'MathloadFuncs','rxmath','MathLoadFuncs'
-- call MathloadFuncs
```

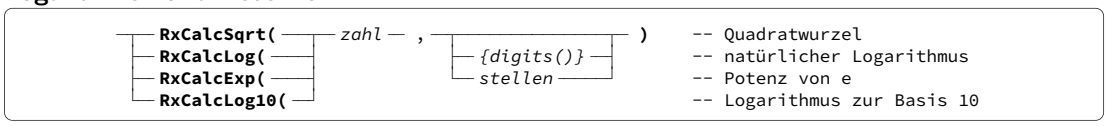
Damit erhält Rexx die Information, in der Datei `rxmath.dll` nach dem mathematischen Funktionen zu suchen.
Wie *Walter Pachl*[3] bei einem Test der Bibliothek mit 2726 Werten festgestellt hat, ist in etwa 30 Prozent der Fälle damit zu rechnen, dass die 16. Stelle um 1 nach oben oder unten vom korrekten Resultat abweicht. Das dürfte für gewöhnliche Rechnungen nicht ins Gewicht fallen, sei aber erwähnt.
RXMATH stellt die folgenden mathematischen Funktionen zur Verfügung:

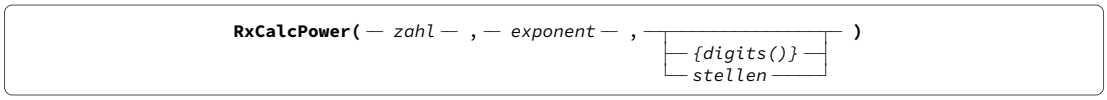
[3] *What's Wrong with Rexx ?* Vortrag, IBM Sindelfingen 2004



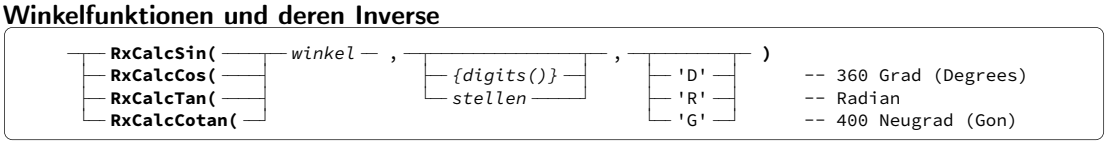
Liefert die Zahl Pi. Für diese und alle anderen RXMATH-Funktionen gilt: Die Rechengenauigkeit des aufrufenden Programms, wie von **digits()** geliefert, gilt auch innerhalb der Funktion. Der gewünschte Wert kann auch als Zahl *stellen* oder als Variable übergeben werden, die eine ganze Zahl zwischen 1 und 16 sein muss. Größere Zahlen als 16 werden wie 16 (*double precision* der benutzten C-Bibliothek) behandelt.

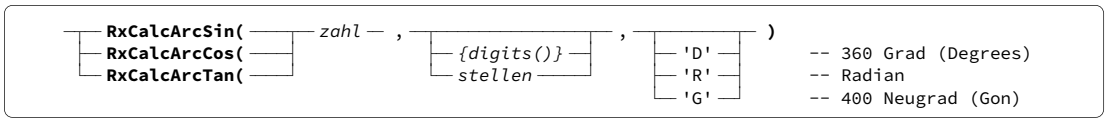
Logarithmen und Potenzen



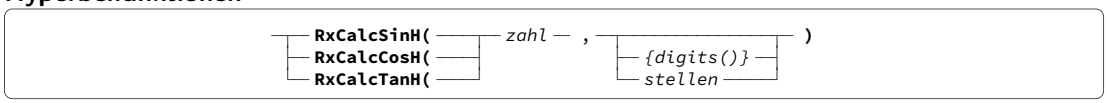


Winkelfunktionen und deren Inverse





Hyperbelfunktionen



6.4 Externe Bibliothek RXM

Dieselben mathematischen Funktionen wie RXMATH, jedoch mit praktisch beliebig wählbarer Genauigkeit, stellt die auf *Walter Pachl* zurückgehende Rexx-Klassendatei `rxm.cls` dar. Sie ist vollständig in Rexx geschrieben und deshalb langsamer als die kompilierte C-Bibliothek RXMATH. Das auf Seite 22 erwähnte Flugbahnprogramm benötigt mit RXM 3.7 Sekunden statt 1.1; bei verdoppelter Genauigkeit (von 16 auf 32 Stellen) sind es 8.0 Sekunden. RXM rechnet intern jeweils mit 10 Stellen mehr als angefordert.
Um RXM benutzen zu können, ist **hinter** der letzten Programmzeile einzufügen:

```
::REQUIRES rxm.cls
```

Die Erweiterung `.cls` für Klassendateien ist an sich eine unverbindliche Konvention. Seit Februar 2020 kann in `::REQUIRES` die Erweiterung `.cls` wegfallen, da seither zuerst nach `.cls`-Dateien gesucht wird.

REQUIRES und der Rexx-Prolog

Wenn –wie hier– das Ziel von REQUIRES eine in Rexx geschriebene Datei ist, wird ein eventuell darin enthaltener „Prolog“ ausgeführt. Als Prolog gelten alle Programmzeilen der Zieldatei, die **vor** der ersten Direktive –jede mit 2 Doppelpunkten eingeleitete Anweisung– stehen. Der Prolog von `rxm.cls` enthält nur die folgende Codezeile:

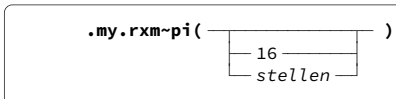
```
.local~my.rxm = .rxm~new(16,"D")
```

Sie erzeugt ein neues Objekt mit dem Namen `.my.rxm`, das die Attribute und Methoden der Klasse RXM enthält. Zugleich wird die Genauigkeit auf 16 Stellen und das Winkelmaß auf 360 Grad (Degrees) voreingestellt. Die Methoden können von da an durch Anhängen an `.my.rxm~` aufgerufen werden. Umgebung `.local` ist der Normalfall und kann beim Aufruf entfallen.

```
.my.rxm~precision=32
```

Dieser Aufruf ändert die Voreinstellung der Genauigkeit von 16 auf hier 32. Alternativ kann bei jedem Methodenaufruf die gewünschte Genauigkeit angegeben werden. RXM vermag leider die Genauigkeit des aufrufenden Rexx-Programmes nicht automatisch zu übernehmen.

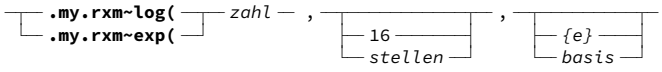
Methoden in RXM



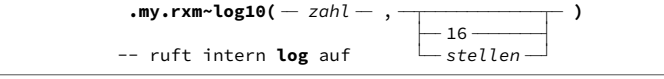

```
-- .my.rxm~pi      ⇒    3.141592653589793
-- .my.rxm~pi(64)  ⇒    3.14159265358979323846264338327950288419716939937510582097494459 2
```

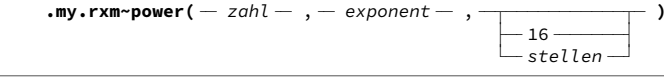
Die Klammern () können entfallen, wenn kein Argument übergeben wird.

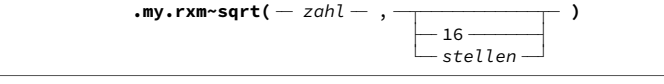
Logarithmen und Potenzen



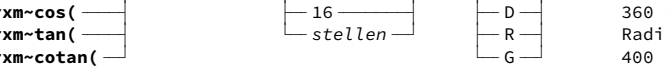

```
-- .my.rxm~log(2)       ⇒    0.6931471805599453      log Basis e
-- .my.rxm~log(2,,10)   ⇒    0.3010299956639812      log Basis 10
-- .my.rxm~log(2,,2)    ⇒    1                       log Basis 2
-- .my.rxm~exp(-0.5)    ⇒    0.6065306597126334      e hoch -0.5
```

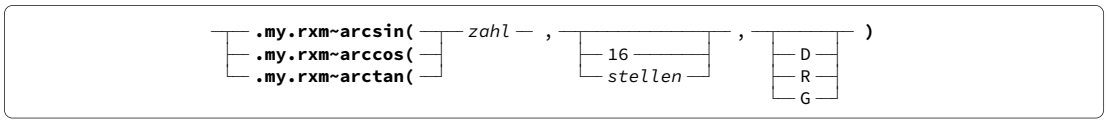




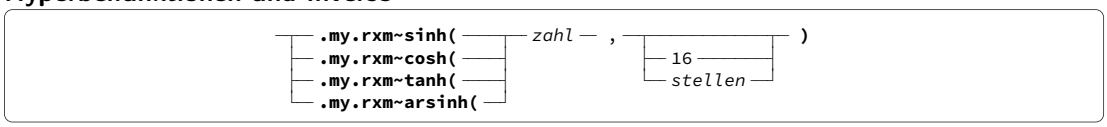


Winkelfunktionen und deren Inverse





Hyperbelfunktionen und Inverse



Gegenüber Bibliothek RXMATH ist Methode **arsinh** hinzugekommen.

Aufruf auch als Funktion

RXM ist mit ROUTINE-Direktiven so programmiert, dass anstelle der obigen Methodenaufrufe auch die Funktionssyntax benutzt werden kann, wie von RXMATH her gewohnt. Es ist dabei anstelle von **RxCalc...** das Präfix **rxm...** zu benutzen. Der hintere Namensteil ist bei beiden identisch, zum Beispiel:

statt **RxCalc**`Sin(...)` *verwende* **rxm**`Sin(...)`

Die Reihenfolge der Argumente ist identisch. Nach der Syntaxprüfung wird intern die zugehörige Methode aufgerufen, in diesem Beispiel also `.my.rxm~Sin(...)`.
Bei den Funktionen **rxmLog()** und **rxmExp()** kann als drittes Argument eine *basis* angegeben werden, wie bei den zugehörigen Methoden. Funktion **rxmArsinh()** ist zusätzlich vorhanden.
Der Vorteil aus Bedienersicht ist die zusätzliche Syntaxprüfung mit erklärenden Fehlermeldungen, die in die Funktionsaufrufe programmiert wurden. Andererseits werden rechenintensive Programme beim direkten Methodenaufruf wohl etwas schneller sein. Das zu dem auf Seite 25 erwähnten Laufzeitvergleich dienende Flugbahnprogramm benutzt Methodenaufrufe.

Herunterladen von RXM.CLS

Die Bibliothek ist auf `rosettacode.org` über das Menü **Explore**, Auswahl **Tasks** und in der dann angezeigten alphabetischen Liste unter **Trigonometric Functions** erreichbar. Dort muss zur Sprache **ooRexx** vorgeblättert werden. Alternativ lautet die direkte URL dafür:

`rosettacode.org/wiki/Trigonometric_functions#ooRexx`

Aus der angezeigten HTML-Datei können die benötigten Daten mit „Cut and Paste“ in Textdateien kopiert werden.

- Der erste Textblock mit der Überschrift **rxm.cls** ist ein Hilfetext und kann zum Beispiel als Datei `rxm.txt` gespeichert werden.
- Danach folgt ein Rexx-Programm für Demonstration und Funktionstest. Es kann zum Beispiel als Datei `rxmdemo.rex` gespeichert werden.
- Der folgende, mit **Output** überschriebene Block listet die Ausgabe des Demoprogramms und könnte als `rxdemo.txt` gespeichert werden.
- Erst der letzte, als **Package rxm** markierte, sehr große Textblock ist die gesuchte Programmbibliothek, die unter dem Namen `rxm.cls` gespeichert werden muss.

Diese Datei `rxm.cls` sollte in das ooRexx-Installationverzeichnis kopiert werden. Dazu sind Administratorrechte erforderlich. Danach können alle Programme diese Bibliothek verwenden.

7 Bits und Bytes

Kleinste speicherbare Einheit ist das Zeichen (Byte), wobei ein Byte 8 Bit belegt. Es bestehen drei Möglichkeiten, ein Byte im Programm zu schreiben, die hier am Beispiel **Z** gezeigt werden:

- Die **Character** Darstellung (kurz c) benutzt das am Bildschirm oder auf der Tastatur dargestellte grafische Zeichen (Glyph), hier das **Z**. Nur ein Teil der 256 möglichen Bytes sind auf diese Weise darstellbar.
- In **binärer** Darstellung werden die 8 Bit des Z als `'01011010'b` geschrieben. Die binäre[1] Schreibweise der 256 Zeichen erstreckt sich also über:

  `'00000000'b ... '11111111'b`

- Die **hexadezimale** (kurz hexa) Darstellung des Z ist `'5A'x`. Dies ist leichter lesbar als eine Bitkette. Jeweils 4 Bit haben 16 mögliche Werte. Die ersten 10 Werte werden durch die Ziffern 0 bis 9 dargestellt. Für den 11. bis 16. Wert benutzt man die „Ziffern" A bis F. Die 8 Bit jedes Zeichens kommen also mit 2 hexadezimalen Ziffern aus:

  `'00'x ... 'FF'x`

Es gibt noch eine vierte Darstellungsmöglichkeit, die jedoch nicht direkt in das Programm geschrieben werden kann. Sie ist aber über Konvertierfunktionen nutzbar, zum Beispiel um hexadezimale Speicheradressen in dezimale Längen umzurechnen.

- Die **dezimale** Darstellung ist die laufende Nummerierung der 256 Zeichen durch die Zahlen 0 bis 255. Buchstabe Z hat die Nummer 90.

Binäre `'11010110'b`, hexadezimale `'D6'x` und dezimale (`214`) Darstellung stehen unveränderlich für dasselbe Byte. Für die Zeichen auf Bildschirm und Tastatur gilt das nur eingeschränkt. Unterschiedliche Zuordnungstabellen (*Codepages*) von Bytes zu Glyphen existieren, um sprachliche Besonderheiten wie die deutschen Umlaute zu berücksichtigen. Außerhalb der Zeichen `'20'x ... '7E'x` (Ziffern, Buchstaben und Interpunktion) muss man mit Unterschieden rechnen. So kann für Zeichen `'9A'x` statt **Ü** auch **š** erscheinen oder umgekehrt.
In der binären und der hexa Schreibweise sind Leerstellen zur besseren Lesbarkeit erlaubt und werden ignoriert. Die Länge ist beliebig. Bei Bedarf wird links mit Null auf das nächste vollständige Byte aufgefüllt. Folgende Funktionen zur Umwandlung sind verfügbar (die 2 sollte als „to" gelesen werden):

von: nach:	Binär	Char	Dezimal	Hexa
Binär		—	—	b2x
Char (Glyph)	—		c2d	c2x
Dezimal	—	d2c		d2x
Hexadezimal	x2b	x2c	x2d	

Umwandlung von und nach binär ist nur über die hexadezimale Darstellung möglich. Binäre und hexadezimale Funktionen erwarten als Argumente einfache Zeichenketten **ohne** nachgestelltes `b` oder `x`.

[1] Die Erfahrung hat gezeigt, dass `b` und `x` nicht als Namen für Variable verwendet werden sollten, da sonst der Interpreter gelegentlich eine binäre oder hexadezimale Zeichenkette vermutet, wo keine ist.

Umwandlungsbeispiele

```
c2x('Z')          ⇒   5A
x2c('5A')         ⇒   Z
x2b('5A')         ⇒   01011010
b2x('0101 1010')  ⇒   5A
```

Vorzeichenlose ganze Zahlen

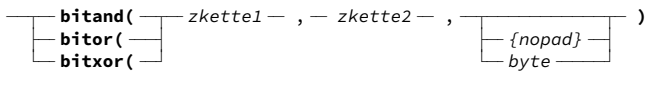
```
c2d('Z')      ⇒   90
x2d('FFFF')   ⇒   65535
d2x(123456)   ⇒   1E240     -- beachte ungerade Stellenzahl
```

Ganze Zahlen mit Vorzeichen

```
x2d('FFFF',4)   ⇒   -1         --
x2d('FFFF',8)   ⇒   65535      --  0000FFFF  =  65535
x2d('00FF',2)   ⇒   -1         --  FF  =  -1
d2x(-1,8)       ⇒   FFFFFFFF   --
```

Um negative Zahlen darzustellen, ist immer eine Längenangabe notwendig. Dann wird das erste Bit als Vorzeichen interpretiert.

Bitweise logische Operationen




```
-- Beispiel:          'e'   =   '65'x   =   '0110 0101'b
--                    'Y'   =   '59'x   =   '0101 1001'b
                                           ---------------
bitand('e','Y')   ⇒   'A'   =   '41'x   =   '0100 0001'b
bitor('e','Y')    ⇒   '<'   =   '7D'x   =   '0111 1101'b
bitxor('e','Y')   ⇒   '}'   =   '3C'x   =   '0011 1100'b
```

Diese Funktionen liefern jeweils eine Zeichenkette in der die Bits von *zkette1* und *zkette2* per AND, OR oder XOR logisch miteinander verknüpft sind: Sind die Zeichenketten unterschiedlich lang, erfolgt der Vergleich nur bis zum Ende der kürzeren Zeichenkette. Die restlichen Bytes stehen unverändert im Ergebnis. Nur falls *byte* angegeben ist, wird die kürzere Zeichenkette damit rechts aufgefüllt.

8 Die Multiwerkzeuge Stammvariable und USE ARG

Die einfachste, schon im klassischen Rexx existierende Datenkollektion ist die Stamm- oder zusammengesetzte Variable (engl. *stem variable, compound variable*), deren Name aus mindestens zwei Teilen besteht, die durch einen Punkt getrennt sind:

```
stamm.ast
stamm.ast.zweig
stamm.ast.zweig.blatt
beliebig verlängerbar ...
```

Ein einfaches Beispiel mit dem Stamm `hanse.` und natürlichen Zahlen als Ast wäre:

```
hanse.1  = 'Bremen'
hanse.2  = 'Hamburg'
hanse.3  = 'Lübeck'
hanse.4  = 'Wismar'
hanse.5  = 'Rostock'
hanse.6  = 'Stralsund'
hanse.7  = 'Greifswald'
```

Diese –mit natürlichen Zahlen indexierte– *Grundform* der Stammvariable spielt im Austausch von Daten zwischen Programmen eine zentrale Rolle. Durch das in Rexx 3.0 eingeführte Schlüsselwort **use arg** (Seite 34) hat ihre Bedeutung noch stark zugenommen, weil seitdem die gemeinsame Nutzung großer Datenstrukturen *ohne* interne Kopiervorgänge möglich ist.

8.1 Durchgezählte Stammvariable

Dies ist die klassische Form für Datenübergabe und Rücklieferung von Daten. *Durchgezählt* erweitert die eben beschriebene Grundform um folgende Bedingungen:

- Das erste Element hat die Nummer 1.
- Die folgenden Elemente haben *lückenlos* jeweils eine um 1 höhere Nummer.
- Die Nummer des letzten Elements muss in Element 0 gespeichert werden. Für das obige Beispiel der Hansestädte also: `hanse.0 = 7`.

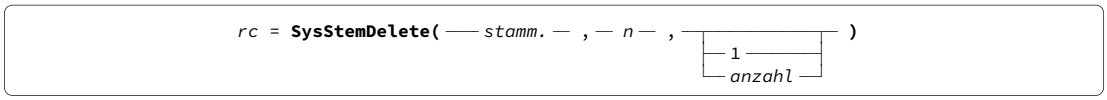
Mit natürlichen Zahlen indexierte Stammvariable, welche diese drei Bedingungen erfüllen, heißen in dieser Kurzreferenz *durchgezählte Stammvariable*. Rexx ist intern **nicht** auf diese Eigenschaften angewiesen. Sie gelten jedoch für die klassische Schnittstelle, zum Beispiel zu Funktionen wie **SysFileTree** (siehe Seite 17) und die hier auf der nächsten Seite beschriebenen.

```
i = 4
say hanse.[i-1]     ⇒     Lübeck
say hanse.i         ⇒     Wismar
say hanse.[i+3]     ⇒     Rostock
```

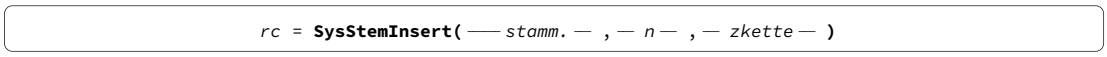
Die Schreibweise des Index mit [] erleichtert insbesondere innerhalb von Schleifen den Zugiff auf benachbarte Elemente.

Funktionen für durchgezählte Stammvariable

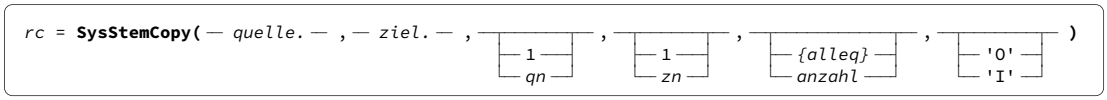
In allen Funktionsaufrufen kann *stamm.* auch ohne Punkt angegeben werden. Die Funktionen ändern Element *stamm.0* je nach Zugang/Abgang von Elementen.



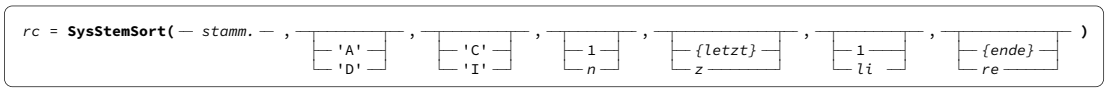
Löscht aus Stammvariable *stamm.* genau *anzahl* Elemente, beginnend mit Nummer *n*. Nachfolgende Elemente besetzen die dadurch freigewordenen Plätze. Wird Element 3 gelöscht, erhält das bisheringe Element 4 die Nummer 3 und Element *stamm.0* wird um 1 verringert.



Fügt den Inhalt *zkette* als neues Element mit Nummer *n* ein. Das existierende Element *n* und alle nachfolgenden erhalten eine um 1 höhere Nummer. Ist *n* um 1 größer als *stamm.0*, wird das neue Element am Ende angehängt.



Kopiert aus *quelle.* genau *anzahl* Elemente, beginnend mit Nummer *qn*, nach *ziel.*. In *ziel.* vorhandene Elemente, beginnend mit Nummer *zn* werden dabei überschrieben.[1] Falls jedoch als sechstes Argument ein **I** (Insert) angegeben ist, werden die neuen Elemente ab Nummer *zn* eingefügt und die vorhandenen erhalten entsprechend höhere Nummern. Ist *zn* um 1 größer als *ziel.0*, werden die neuen Daten an *ziel.* angehängt.
Falls *ziel.* nicht existiert, wird unter diesem Namen eine Kopie von *quelle.* erzeugt, sofern keine weiteren Argumente angegeben werden. Angabe von I ist dann wirkungslos.
Besonderheiten im **Insert**-Modus, sofern *ziel.* existiert, sind:
Falls keine anderen Argumente angegeben werden, stehen im Ergebnis zuerst die Elemente aus *quelle.* und anschließend die aus *ziel.*.
Der Kopiervorgang erfolgt Element für Element. Ist *zn* gleich 2 oder größer, stehen die Elemente von *quelle.* deswegen in umgekehrter Reihenfolge im Ergebnis. Das letzte kopierte Element aus *quelle* erhält dabei Nummer *zn*.



Sortiert die Elemente in *stamm.* nach deren Bytes in den Positionen *li* bis *re*. Voreingestellt ist die aufsteigende Sortierung (A) mit Berücksichtigung von Groß- und Kleinschreibung (C für Case). Argument D (Descending) bewirkt *absteigende* Sortierung. Argument I führt zum Ignorieren der Groß- und Kleinschreibung. Der Sortierlauf kann auf die Elemente Nummer *n* bis *z* beschränkt werden.
Der benutzte Sortieralgorithmus ist „nicht stabil“. Elemente mit *identischem* Sortierwert behalten nicht dieselbe Reihenfolge wie vor dem Sortierlauf. Seite 37 beschreibt eine Alternative.

8.2 Allgemeine Stammvariable

Wie oben erwähnt, ist Rexx nicht darauf angewiesen, dass die im Ast verwendeten Zahlen mit 1 beginnen. Es können beliebige Lücken vorhanden sein. So ließe sich das Beispiel der Hansestädte auch mit den Telefonvorwahlen indexieren:

[1] Im Gegensatz zum Diagramm im Handbuch ist **O**verlay der Vorgabewert, nicht **I**nsert.

```
hanse.0421    = 'Bremen'
hanse.040     = 'Hamburg'
hanse.0451    = 'Lübeck'
hanse.03841   = 'Wismar'
hanse.0381    = 'Rostock'
hanse.03831   = 'Stralsund'
hanse.03834   = 'Greifswald'
```

Hierbei ist zu beachten, dass die Indexe als *Zeichenketten* interpretiert werden. Der Index 040 ist *nicht* derselbe wie 40.

```
gesucht = 03831
say 'Hansestadt' hanse.gesucht    ⇒    Hansestadt Stralsund
```

Trifft Rexx auf eine Stammvariable –hier `hanse.gesucht` – dann wird der Ast (auch Zweige und Blätter, falls vorhanden) geprüft, ob er eine Variable ist. Hier trifft das zu: `gesucht` ist eine Variable mit dem Wert `03831`.
Im nächsten Schritt wird geprüft, ob der resultierenden Stammvariable `hanse.03831` ein Wert zugeordnet ist. Hier lautet der Wert: `Stralsund`, den Rexx an die Stelle der Stammvariable setzt. Damit wird der oben gezeigte Text am Bildschirm ausgegeben.
Was bei einer undefinierten Vorwahlnummer passiert, zeigt folgendes Beispiel:

```
gesucht = 0815
say 'Hansestadt' hanse.gesucht    ⇒    Hansestadt HANSE.0815
```

Hat die resultierende Stammvariable wie hier `hanse.0815` keinen zugewiesenen Wert, wird sie wie jede unbekannte Variable (siehe Seite 46) als „sie selbst“ in Großbuchstaben behandelt: `HANSE.0815`.

Initialisieren

Um eine Ausgabe wie `HANSE.0815` zu vermeiden, kann **vor** der ersten Definition von Variablen eines Stammes –in diesem Beispiel `hanse.` – über die Anweisung:

```
hanse.    = 'unbekannte Vorwahl'
```

bewirkt werden, dass für alle unbesetzten Elemente der Text *unbekannte Vorwahl* als Inhalt erscheint. Auch die leere Zeichenkette ist als Wert zulässig. Diese Anweisung löscht alle eventuell vorher definierten Elemente des Stamms.

Indexierung mit anderen Zeichen

Grundsätzlich können als Ast, Zweig, Blatt usw. benutzte Variable nicht nur Ziffern im Index enthalten, sondern alle Zeichen. Das Beispiel der Hansestädte könnte also auch mit Autokennzeichen indexiert werden:

```
hanse.     = 'unbekannt'
hanse.HB   = 'Bremen'
hanse.HH   = 'Hamburg'
hanse.HL   = 'Lübeck'
hanse.HWI  = 'Wismar'
hanse.HRO  = 'Rostock'
hanse.HST  = 'Stralsund'
hanse.HGW  = 'Greifswald'
```

Als Ast werden hier die Ortszeichen der Autokennzeichen verwendet. **Warnung:** Dies ist hier zur leichteren Verständlichkeit so geschrieben. In praktischen Programmen sollte nicht so gearbeitet werden. Dazu mehr im übernächsten Absatz.

```
gesucht = 'HST'
say 'Kennzeichen' hanse.gesucht    ⇒    Kennzeichen Stralsund

gesucht = 'EMM'
say 'Kennzeichen' hanse.gesucht    ⇒    Kennzeichen unbekannt
```

Das Prinzip ist dasselbe wie oben bei den Vorwahlnummern.

```
kennz = 'HB'
hanse.kennz = 'Bremen'
kennz = 'HH'
hanse.kennz = 'Hamburg'
 usw.
```

Die hier gezeigte Definitionsweise über Zeichenketten garantiert korrekte Indexwerte. Das vorige, einfach verständliche Kennzeichenbeispiel setzt nämlich stillschweigend voraus, dass es Variable mit dem Namen BB ... HGW noch nicht gibt. Solche Annahmen werden bei späteren Änderungen leicht übersehen und das Programm reagiert danach anders als gewollt.
Üblicherweise ist die Zuordnung weniger umständlich als es jetzt scheinen mag, weil in der Praxis die betreffenden Daten ohnehin per Programmschleife aus Dateien gelesen werden.

Mehrdimensionale Stammvariable

```
-- NL für Niederlande (Provinzen)
staat = 'NL'
prov  = 'GE'
land.staat.prov = 'Gelderland'
prov  = 'GR'
land.staat.prov = 'Groningen'

-- DE für Deutschland (Bundesländer)
staat = 'DE'
bland = 'MV'
land.staat.bland = 'Mecklenburg–Vorpommern'
bland = 'NI'
land.staat.bland = 'Niedersachsen'
bland = 'SN'
land.staat.bland = 'Sachsen'

-- FR für Frankreich (Departements)
staat = 'FR'
dept  =  72
land.staat.dept = 'Sarthe'
dept  =  62
land.staat.dept = 'Pas-de-Calais'

-- Anwendungsbeispiel:
ix = 'FR'
iy =  72
say land.ix.iy      ⇒   Sarthe
```

Vorstehendes Beispiel illustriert eine zweidimensionale Stammvariable (engl. *compound variable*) aus Abkürzungen für Staaten und deren Untergliederungen. Es zeigt auch, dass Buchstaben und Ziffern gemischt als Index verwendbar sind. Bei den Ziffern sind Hochkommas entbehrlich, weil Zahlen nicht mit Variablen verwechselt werden können.
Bei mehr als einer Dimension ist das Initialisieren eines Vorgabewerts für unbesetzte Elemente *nicht* möglich.

8.3 Vereinfachte Datenübergabe mit USE ARG

Im Normalfall werden eine oder mehrere Zeichenketten an ein Unterprogramm[2] übergeben. Es kann aber nur eine einzige Zeichenkette als Ergebnis zurückgeliefert werden, auch wenn **RETURN** viele Megabytes handhaben kann. Datensatz-Strukturen auf diese Weise auszutauschen erfordert viel Programmierarbeit in Haupt- und Unterprogramm.
Mit dem Schlüsselwort **USE ARG** anstelle von **ARG** im Unterprogramm können Stammvariable zum Datenaustausch zwischen Haupt- und Unterprogramm benutzt werden. Deren Name genügt als Beschreibung. Rexx stellt dem Unterprogramm sämtliche internen Verwaltungsinformationen über die Elemente der Stammvariable zur Verfügung. Im Gegensatz zur durchgezählten Stammvariable (Seite 30) können also auch unbesetzte Elemente existieren. Element 0 hat keine Sonderstellung und wird nicht als Zähler benötigt.
Statt für 1000 Berechnungen ein Unterprogramm 1000 mal aufzurufen, genügt eventuell ein einziger. Dabei übergibt man eine Stammvariable mit den 1000 Werten und das Unterprogramm schreibt die Ergebnisse in dieselbe oder in eine andere Stammvariable.

```
-- Datei hauptprog.rex
testvar  = 'Ostsee'
hanse.1  = 'Bremen'
hanse.2  = 'Hamburg'
hanse.3  = 'Lübeck'
hanse.4  = 'Wismar'
hanse.5  = 'Rostock'
hanse.6  = 'Stralsund'
hanse.7  = 'Greifswald'

call unterprog hanse. , testvar

-- Die im Unterprogramm unterprog gemachten Änderungen der
-- Stammvariable hanse. sind jetzt auch hier wirksam.

say hanse.77.88     ⇒    Demonstration     neues Element
say testvar         ⇒    Ostsee            Änderung unwirksam
say result          ⇒    Rückgabe          durch CALL gesetzt
```

Dieses Hauptprogramm stellt seine Stammvariable `hanse.` dem Unterprogramm zur Verfügung. Ebenso eine hier `testvar` genannte gewöhnliche Variable.

```
-- Datei unterprog.rex
 use arg demo. , einfach

-- Die von hauptprog.rex  als erstes Argument übergebene Stamm-
-- variable hanse. heißt hier demo.  Es handelt sich aber
-- um dieselben Daten.
say demo.1                          ⇒   Bremen
demo.77.88 = 'Demonstration'

-- Dagegen sind Änderungen an der gewöhnlichen Variable
-- für das Hauptprogramm nicht sichtbar.
say einfach                         ⇒  Ostsee
einfach = 'abcd'

return 'Rückgabe'           -- konventionelle Datenrückgabe
```

Das Unterprogramm kennt die vom Aufrufer benutzten Namen nicht. Die als erstes Argument übergebene Stammvariable wird intern `demo.` genannt. Wie man am Beispiel sieht, können ihr neue Elemente hinzugefügt werden. Dabei darf die Dimension erhöht werden. Alle Änderungen, Löschungen, Erweiterungen findet das Hauptprogramm anschließend in seiner Stammvariable `hanse.` vor. Denn alle Manipulationen des Unterprogramms an `demo.` wurden in Wirklichkeit an `hanse.` ausgeführt.

[2] Beschreibung der Aufrufmöglichkeiten auf Seite 48.

Änderungen des Unterprogramms an gewöhnlichen Variablen wie `testvar` sieht das Hauptprogramm dagegen nicht.
Ein Unterschied zu **ARG** ist die Behandlung fehlender Argumente durch **USE ARG**. Hätte das Hauptprogramm kein zweites Argument übergeben, wäre von **ARG** die Variable `einfach` des Unterprogramms als leere Zeichenkette initialisiert worden. Bei **USE ARG** ist Variable `einfach` dagegen in diesem Fall *undefiniert* und hat nach den Rexx-Regeln ihren Namen `EINFACH` als Inhalt.
Ein weiterer Unterschied zu **ARG** ist, dass **USE ARG** ausschließlich *durch Komma getrennte* Variablennamen akzeptiert. Steht aber hinter **USE ARG** kein Name zwischen den Kommas, ignoriert **USE ARG** stillschweigend das an dieser Position übergebene Argument. Dadurch können **ARG** und **USE ARG** sich problemlos dieselbe übergebene Argumentenkette teilen, *wenn die Argumente logisch durch Kommas voneinander getrennt sind.*
Initialisieren von Stammvariablen (Seite 32), auf die über **USE ARG** zugegriffen wird, ist im Unterprogramm nicht sinnvoll. Dabei erzeugt es unter dem alten Namen (im Beipiel `demo.`) ein *neues* Programmobjekt, mit dem normal gearbeitet werden kann. Jedoch kappt diese Aktion die logische Verbindung zum Hauptprogramm. In `hanse.` wird weder die Initialisierung wirksam, noch spätere Änderungen.

8.4 USE ARG Verwendung mit Objekttyp Array

Laut Handbuch können mit **USE ARG** alle Arten von Datenkollektionen zwischen Haupt- und Unterprogrammen kommuniziert werden. Also auch Objekte vom Typ **Array**, deren praktischer Einsatz ab Seite 43 im Abschnitt über externe Sortierung mittels Funktion **sort2** gezeigt wird. Die Anwendung ist analog zur Stammvariable.
Unterschiedlich ist nur der Programmcode für die Arrays selbst, also das Befüllen von Array-Elementen mit Daten oder das Auslesen der Daten.

```
-- Datei hauptprog.rex

 telefonbuch    = .array~new    -- Array telefonbuch anlegen
 telefonbuch[5] = 'beliebige Zeichenkette'

call unterprog telefonbuch

say telefonbuch[5]              ⇒  'geänderte Zeichenkette'
```

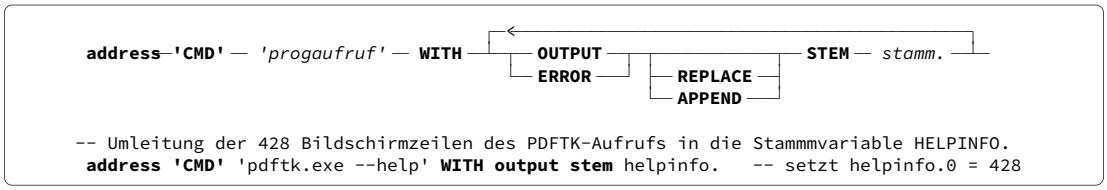
Dieses Skelett eines Hauptprogramms zeigt das Anlegen des Arrays und das Befüllen seines Elementes 5 mit Daten. Beim Aufruf des Unterprogramms wird der Name des Arrays ganz normal als Argument übergeben.

```
-- Datei unterprog.rex
use arg verzeichnis         -- lokaler Name des Arrays

say verzeichnis[5]          ⇒  'beliebige Zeichenkette'

verzeichnis[5] = 'geänderte Zeichenkette'
return 0
```

Im Unterprogrammm wird das übergebene Argument unter dem Namen `verzeichnis` in Zugriff genommen. Dabei erkennt ooRexx ohne weiteres Zutun, dass es sich um ein Array-Objekt handelt. Folglich können arraytypische Anweisungen darin verwendet werden. Hier sind es die Ausgabe des Elements 5 am Bildschirm und seine anschließende Änderung.
Alle Änderungen des Unterprogramms am Array `verzeichnis` sieht das Hauptprogramm in „seinem“ Array `telefonbuch`. Denn es handelt sich letztlich um dasselbe Objekt.

8.5 Umleitung der Ausgaben fremder Programme

Es kommt vor, dass man die Bildschirmausgaben von fremden Programmen –wie im nachfolgenden Beispiel `pdftk.exe`– „abfangen“ will, um sie weiter zu verarbeiten. Abgesehen von den Möglichkeiten des Windows-Pipelining lässt sich sonst wenig tun, da man kompilierte Fremdprogramme nicht ändern kann.
Mit der in ooRexx 5.0 neuen Erweiterung **WITH** des Schlüsselworts **ADDRESS** besteht jetzt die Möglichkeit, von Fremdprogrammen per STDOUT und STDERR am Bildschirm ausgegebene Zeilen als durchgezählte Stammvariable im aufrufenden Rexx-Programm bereitzustellen. Diese Methode funktioniert zum Beispiel bei EXE-, BAT-, CMD- und auch REX-Programmen.




```
-- Umleitung der 428 Bildschirmzeilen des PDFTK-Aufrufs in die Stammmvariable HELPINFO.
address 'CMD' 'pdftk.exe --help' WITH output stem helpinfo.   -- setzt helpinfo.0 = 428
```

CMD in Hochkommas steht hier wegen der vorgegebenen Syntax von **ADDRESS**. Es bewirkt nur die gewünschte Weitergabe des rechts davon stehenden Programmaufrufs an Windows. Dieser kann sicherheitshalber auch in Hochkommas stehen.
Hinter Schlüsselwort **WITH** folgen ein oder zwei Anweisungsblöcke. Typ **OUTPUT** steuert die Umleitung von STDOUT-Ausgaben des Programmaufrufs. Typ **ERROR** leitet dessen STDERR-Ausgaben um. Werden beide benutzt, folgt der zweite Anweisungsblock rechts vom ersten. Dann müssen die angegebenen Stammvariablen verschiedene Namen haben.
Hinter dem Wort **STEM** folgt der Name der zu füllenden Stammvariable. Nach Ausführung des Programmaufrufs enthält Element *stamm.0* die Anzahl der Zeilen und die Elemente *stamm.1* und folgende den Inhalt jeweils einer Zeile.
Der vorige Absatz gilt, falls keine Option oder falls **REPLACE** vor **STEM** steht. Ist dagegen Option **APPEND** angegeben, muss Element *stamm.0* schon existieren und eine Zahl enthalten. Die neuen Zeilen werden dann als Element *stamm.[zahl+1]* und folgende an die bestehende Stammvariable angehängt.
Mit einem hier nicht dargestellten Anweisungsblock **INPUT** können bei Bedarf auch die mittels STDIN an das Programm übergebenen Informationen gesteuert werden. Die durch **WITH** bereitgestellten Erweiterungen bieten noch mehr, recht komplexe Möglichkeiten. Sie bedeuten aber meiner Meinung nach keinen wirklichen funktionalen Zugewinn für die Programmierpraxis. Daher sind sie hier nicht weiter beschrieben.

Umgang mit Leerstellen in Pfadnamen

Manche älteren Programme kommen mit Leerstellen in Pfadnamen nicht zurecht. Jedes Verzeichnis hat aber auch einen Windows-internen Kurznamen ohne Leerstellen. Auf Seite 16 ist beschrieben, wie man ihn feststellen kann.

9 Stabiles Sortieren

Stabil bedeutet, dass Einträge mit identischem Sortierwert dieselbe Reihenfolge beibehalten wie vorher. Das ist oftmals für die weitere Verarbeitung sehr nützlich. Funktion **SysStemSort()** (S. 31) sortiert leider nicht stabil; der Hessling-Editor auch nicht. Andererseits arbeiten die *ooRexx* Sortier*klassen* stabil. Wer eine schnelle Lösung sucht, sollte auf Seite 42 bei **sort2** weiterlesen. Hier folgen mehr Einzelheiten.
Der Handbuchtext schleppt als Ballast noch die Unterscheidung zwischen `StableSortWith` und `SortWith` mit. *Beide* benutzen jedoch den **stabilen** Mergesort-Algorithmus.

9.1 Methode SORTWITH in einem klassischen Programm

Von den 13 Datenkollektionsklassen in ooRexx wird hier die Klasse **Array** gewählt. Im Gegensatz zur Stammvariable kann sie auch mit den `do ... over` Schleifen einfach genutzt werden. Die Beispieldaten in Datei `hanse.dat` sehen so aus:

```
                                    0421  Bremen
                                    040   Hamburg
                                    0451  Lübeck
                                    03841 Wismar
                                    0381  Rostock
                                    03831 Stralsund
                                    03834 Greifswald
```

Die Vorwahlnummern beginnen in Spalte 1 und sind, wie man sieht, maximal 5 Stellen lang. Die Ortsnamen beginnen in Spalte 7 und der längste hat 10 Buchstaben.

```
arg a1 .                    -- V oder O für Vorwahl- oder Ortssortierung
select case a1
when 'V' then do
     start = 1
     len   = 5
     end
when 'O' then do
     start = 7
     len   = 10
     end
otherwise
     say 'Dieses Programm muss mit V oder O aufgerufen werden.'
     exit 24
     end
```

Beim Programmstart soll der Buchstabe **V**[orwahl] oder **O**[rtsname] mitgegeben werden, um die Sortierspalten entsprechend zu setzen. Diese Programmzeilen sorgen dafür. Es ist kein Problem, dass einige Ortsnamen kürzer sind als die Sortierlänge 10 Zeichen.

```
tabelle = .array~new                -- leeres Array TABELLE anlegen
```

Unter dem Namen `tabelle` wird ein neues Objekt der Klasse `Array` angelegt.

```
indatei = 'hanse.dat'                  -- zu lesende Datei
do i=1 while lines(indatei)
     zeile = linein(indatei)           -- Zeile lesen und ...
     tabelle~append(zeile)             -- als Element an TABELLE anhängen
     end i
```

Methode `append` fügt jede Datenzeile dem Array `tabelle` hinzu.

```
tabelle~sortwith(.ColumnComparator~new(start,len))
```

Hier erfolgt der Sortiervorgang. Es wird ein Objekt der Klasse `ColumnComparator` erzeugt, dem in den Variablen *start* und *len* die Sortierspalten mitgeteilt werden. Dieses Objekt wird dann von Methode `SortWith` zur Steuerung der Sortierung von `tabelle` benutzt.

```
do idx over tabelle
     say idx
     end
```

Um das Ergebnis mit `say` am Bildschirm auszugeben, arbeitet die `do ... over` Schleife alle Elemente des Arrays `Tabelle` ab. Anstelle von `idx` kann ein beliebiger Variablenname verwendet werden.

```
0381  Rostock
03831 Stralsund
03834 Greifswald
03841 Wismar
040   Hamburg
0421  Bremen
0451  Lübeck
```

So sieht die nach Vorwahl sortierte Ausgabe der `do ... over` Schleife aus.

```
-- Alternative: herkömmliche Schleife
do i=1 for tabelle~items
     say tabelle[i]
     end
```

Alternativ kann auf die Array-Elemente auch mit der gezeigten []-Schreibweise (ohne Punkt, da keine Stammvariable sondern ein Array) zugegriffen werden. Methode `items` liefert die Anzahl der Elemente. Da es in dieser Anwendung keine unbesetzten Elemente im Array gibt, ist das zugleich die Anzahl der Schleifendurchläufe.
Damit endet die klassische Lösung stabilen Sortierens.

Andere Sortierfolgen

```
tabelle~sortwith(.CaselessColumnComparator~new(start,len))
```

Soll ohne Rücksicht auf Groß- und Kleinschreibung sortiert werden, verwendet man die `Caseless`-Version der Sortierklasse. Wie mehrfach erwähnt, gelten sprachliche Besonderheiten wie die deutschen Umlaute nicht als gewöhnliche Buchstaben.

```
-- Sortierfolge umkehren mit:
--               .InvertingComparator~new( ── KomparatorKlasse~new(args) ── )

tabelle~sortwith(.InvertingComparator~new(.ColumnComparator~new(start,len)))
```

Klasse `InvertingComparator` kehrt die Sortierfolge von aufsteigend nach absteigend um. Der Name der eigentlichen Sortierklasse wird als Argument an die Methode `new` der Invertierungsklasse übergeben,

Anmerkung: Stabiles Sortieren im Hessling Editor (THE)

Das soeben beschriebene Prinzip kann auch im Hessling Editor verwendet werden, indem die Sortierung nicht im Editor selbst erfolgt, sondern im Rexx-internen Array. Dazu muss ein Makro nur folgende Schritte ausführen:

- Zuerst die Nummern der ersten und letzten Zeile des zu sortierenden Blocks feststellen, um später die alten Zeilen enbloc löschen zu können und danach die sortierten Zeilen an der richtigen Stelle einzufügen.
- Jede Zeile des Blocks wird mit dem Befehl **EXTRACT** gelesen und ins Array `tabelle` kopiert.
- Das Array wird wie oben beschrieben mit `SortWith` sortiert.
- Wegen eines Problems in THE[1] können die alten Zeilen nicht einfach per **REPLACE** und **NEXT** einzeln überschrieben werden.
- Stattdessen mit **DELETE** den gesamten alten Zeilenblock auf einmal löschen.
- Dann mit einer **INPUT**-Schleife die sortierten Zeilen einzeln wieder einfügen, wodurch Aufrufe von **NEXT** unnötig sind.

9.2 Objektorientiertes Programm zum Vergleich

Dieser Abschnitt[2] ist ausdrücklich keine Anleitung. Er soll den im Titel dieser Kurzreferenz angesprochenen *Klassikern* unter den Benutzern einen groben Eindruck vermitteln, wie eine objektorientierte Sortierlösung für `hanse.dat` aussehen könnte.[3] Die Terminologie habe ich im Hinblick auf diese Zielgruppe gewählt. Es wird versucht, durch deutsche Objekt- und Variablennamen (wie `vorwahl`) diese beim Lesen besser von den Rexx-Sprachelementen (wie `expose, compare`) unterscheidbar zu machen.
Für die Datenelemente müssen Objektnamen erdacht werden. Jede Dateizeile enthält 2 Felder (Attribute). Sie werden hier `vorwahl` und `ortsname` genannt. Auch brauchen wir einen Namen für das resultierende Satzformat aus diesen zwei Feldern. Es soll wegen der geplanten Sortieranwendung `hansesort` heißen.

Klassendefinitionen

Dazu wird Datei `oohanse.cls` mit folgendem Inhalt angelegt:

```
::class hansesort public inherit comparable
```

Diese Anweisung definiert die Klasse `hansesort` als von jedem Programm nutzbar (`public`). Da sie zum Sortieren genutzt werden soll, erbt sie die Eigenschaften der vordefinierten Klasse `comparable`.

```
::attribute vorwahl
::attribute ortsname

::method init
  expose         vorwahl ortsname
  use strict arg vorwahl, ortsname
```

[1] Nach meiner Erfahrung mit der ansonsten stabilen 32-Bit Windows-Version 3.3B3 arbeiten die Befehle **NEXT** oder **DOWN** in Makros nicht richtig, sobald sie sehr oft hintereinander ausgeführt werden.
[2] Er ist aus dem mit Rexx gelieferten Beispielprogramm `sortComposite.rex` abgeleitet.
[3] Eine auch Objekte behandelnde Einführung bietet das über `facultas.at\Flatscher` erhältliche Buch FLATSCHER, Rony G.: *Introduction to REXX and ooRexx.* Wien: Facultas 2013

Die Attribut-Direktiven definieren, welche Datenfelder die Struktur `hansesort` enthält. Zugleich wird damit im Hintergrund je eine gleichnamige Methode für den Lese- und Schreibzugriff auf dieses Feld definiert.
Zum Anlegen eines Daten-Objekts muss es eine Methode mit Namen `init` geben. Die Anweisung `expose` bestimmt hier –und bei allen anderen Methoden– welche Felder die Methode lesen und ändern kann. Alle anderen Variablen, die innerhalb der Methode vorkämen, wären für die Außenwelt unsichtbar. Zum Anlegen einer Struktur der Klasse `hansesort` braucht `init` natürlich Zugriff auf alle darin definierten Felder. Hinter Schlüsselwort `expose` steht hier also eine vollständige, *durch Leerstelle* getrennte, Liste der mit `::attribute` deklarierten Felder.
Die Anweisung `use strict arg` liest die beim Anlegen übergebenen Daten und ordnet sie in der Reihenfolge den Feldern zu. Auch hier ist also eine Liste aller definierten Felder notwendig, die aber *durch Komma* voneinander getrennt sein müssen. Die Option `strict` stellt sicher, dass die korrekte Anzahl Argumente vorhanden ist.

```
::method string        -- definiert Format für Ausgabe als Zeichenkette
  expose         vorwahl ortsname
  return '>'vorwahl' -- 'ortsname'<'
```

Das Datensatzformat (die Klasse) `hansesort` ist eine Struktur, keine simple Zeichenkette. Um sie mit Anweisungen wie `say` verarbeiten zu können, benutzt Rexx eine String-Standardmethode zur Ausgabe. Stattdessen kann man eine eigene Methode mit dem vorgegebenen Namen `string` definieren, um die Ausgabe der Daten zu steuern. In diesem Beispiel werden die Feldlängen durch Sonderzeichen sichtbar gemacht.

Sortierklassen

Für jede Sortierung ist eine Unterklasse der Klasse `comparator` zu definieren. Da nach Vorwahl oder Ortsname sortiert werden soll, werden zwei Klassen gebraucht. In jeder Klasse muss eine Methode mit dem vorgegebenen Namen `compare` definiert sein, die die Sortierung steuert.

```
::class VORWsortierung public subclass comparator
::method compare
use strict arg links, rechts
return links~vorwahl~compareto(rechts~vorwahl)
```

Klasse `VORWsortierung` sortiert nach dem Inhalt des Feldes `vorwahl`. Methode `compare` erhält als Argumente zwei aufeinander folgende Inhalte des Feldes `vorwahl` übergeben. Im Beispiel werden sie in die Variablen `links` und `rechts` geladen. Mit Hilfe der eingebauten Vergleichsmethode `compareto` erzeugt der komplizierte Ausdruck hinter `return` den Rückgabewert 1, 0 oder -1. Damit zeigt Methode `compare` dem Rexx-Interpreter an, ob *größer*, *gleich* oder *kleiner* für die Sortierung des Paares `links` und `rechts` gilt.

```
::class ORTSsortierung public subclass comparator
::method compare
use strict arg links, rechts
return links~ortsname~compareto(rechts~ortsname)
```

Klasse `ORTSsortierung` tut dasselbe, jedoch mit Feld `ortsname`. Für jedes weitere Sortierfeld wäre eine entsprechende `comparator` Unterklasse mit zugehöriger Methode `compare` zu definieren.

```
return -links~ortsname~compareto(rechts~ortsname)
```

Diese Zeile ist **nicht** Bestandteil der Datei. Sie soll nur zeigen, wie durch eine winzige Änderung die Umkehrung der Sortierfolge von auf- nach absteigend erreicht werden kann. Es ist das zusätzliche Minuszeichen am Beginn des Ausdrucks rechts von `return`. Damit werden die Ergebnisse -1, 0 und 1 negiert in 1, 0 und -1.
Auf diese Weise ließen sich also bei Bedarf noch zwei weitere Klassen definieren, die jeweils *absteigend* nach Vorwahl oder Ortsname sortieren würden.
Damit ist unsere Klassendatei `oohanse.cls` zur Nutzung durch ein Programm fertig.

Das dazugehörige Rexx-Programm

Die Anwendung der soeben angelegten Klassendatei zeigt Programm `oohanse.rex`:

```
indatei = 'hanse.dat'                  -- zu lesende Datei
tabelle = .array~new                   -- neues Array "TABELLE" anlegen
```

Diese Anweisungen sind dieselben wie im „klassischen“ Programmbeispiel ab Seite 37.

```
do i=1 while lines(indatei)
     zeile = linein(indatei)                    -- Zeile lesen ...
     parse var zeile 1 felda 7 feldb            -- in 2 Felder zerlegen
     tabelle~append(.hansesort~new(felda,feldb)) -- benutzt Klasse HANSESORT
     end i
```

Das Anhängen an das Array erfolgt über die Klasse `hansesort`. Da für diese Klasse 2 Attribute definiert sind, muss jede Dateizeile auch in 2 Felder zerlegt werden. Die Argumentenzahl und -reihenfolge des Aufrufs der Methode `new` muss genau mit der Definition der real benutzten Methode `init` in Klasse `hansesort` übereinstimmen. Statt `felda` und `feldb` könnten hier auch beliebige andere Variablennamen benutzt werden.

```
arg a1 .                -- V oder O für Vorwahl- oder Ortssortierung
select case a1
when 'V' then tabelle~sortwith(.VORWsortierung~new)
when 'O' then tabelle~sortwith(.ORTSsortierung~new)
otherwise
     say 'Dieses Programm muss mit V oder O aufgerufen werden.'
     exit 24
     end
```

Je nach dem beim Programmaufruf übergebenen Buchstaben V oder O wird die zugehörige `comparator` Unterklasse geladen und mit Methode `sortWith` das Array sortiert.

```
do idx over tabelle
     say idx        -- SAY benutzt implizit Methode STRING aus HANSESORT
     end
```

Die Schleife zur Ausgabe des Resultats am Bildschirm ist dieselbe wie im klassischen Beispiel.

```
::REQUIRES oohanse.cls
```

Die Direktive `::REQUIRES` am Schluss des Programms teilt ooRexx mit, in welcher Datei die benötigten Klassendefinitionen zu finden sind. Seit Februar 2020 kann die Erweiterung `.cls` weggelassen werden. Dann sucht Rexx zuerst nach `cls`-Dateien.
Alternativ dazu könnte der Inhalt von Datei `oohanse.cls` an dieser Stelle in das Programm kopiert werden. Dann wären die `public` Optionen in den Klassendefinitionen unnötig. Das würde die Klassen für alle anderen Programme unzugänglich machen.

```
                                    >0381  -- Rostock<
                                    >03831 -- Stralsund<
                                    >03834 -- Greifswald<
                                    >03841 -- Wismar<
                                    >040   -- Hamburg<
                                    >0421  -- Bremen<
                                    >0451  -- Lübeck<
```

Unabhängig davon, ob die Klassendefintionen extern oder intern sind, sieht die nach Vorwahl sortierte Ausgabe aus wie hier gezeigt. Da Array `tabelle` Datensätze im Format der Klasse `hansesort` enthält, verwendet Rexx deren Methode `string` um dem Schlüsselwort `say` diese Daten als Zeichenkette zu übergeben. Die Ausgabe ist somit eingerahmt in die zusätzlichen Zeichen `> ... <`, wie in Methode `string` beispielhaft definiert.
Damit endet Abschnitt 9.2 mit der objektorientierten Lösung stabilen Sortierens. Er ist um 70 Prozent länger als der „klassische“ Abschnitt 9.1 (ohne die Angaben zum Sortieren in THE).

9.3 Externe Sortierfunktion SORT2 als Erweiterung

Die Programmbibliothek **rgf_util2.rex**[4] bietet mit ihrer Funktion **sort2** folgende (stabile) Sortiermöglichkeiten, die **SysStemSort** (Seite 31) und der Hessling-Editor nicht haben:

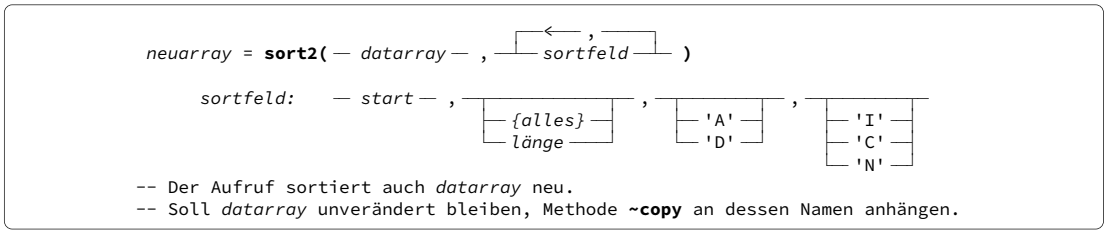
- Sortieren nach mehreren Feldern, wie zum Beispiel Name und Geburtsdatum,
- dabei Mischung von auf- und absteigender Sortierung, sowie
- korrektes Sortieren von Zahlen nach dem numerischen Wert.

```
nicht      <── aufsteigend sortiert ──>
sortiert    zeichenweise    numerisch
 10.1           4.2           -7.2
 -0.6           9.5           -0.6
 -7.2          +8.8            4.2
 +8.8          -0.6           +8.8
 10.2          -7.2            9.5
  9.5          10.1           10.1
  4.2          10.2           10.2
```

Zahlen mit Vorzeichen kommen mit der normalen, zeichenweise erfolgenden Sortierung nicht in die richtige Reihenfolge. **Sort2** kann dies und ist außerdem in der Lage, *nicht stellengerecht ausgerichtete* Zahlen korrekt einzuordnen, solange sie vollständig im angegebenen Sortierfeld stehen. Ebenso kann das Programm die Exponentialdarstellung verarbeiten. Die Schreibweisen `960.2` und `9.602E2` werden dann gleichwertig einsortiert.
Da **sort2** intern objektorientierte Elemente benutzt, müssen die Daten in Objektform vorliegen. „Klassische“ Datenformate lassen sich sehr einfach in die Objektart *Array* umwandeln, und auch wieder zurück. Dies ist auf den folgenden Seiten an einem konkreten Beispiel dargestellt.

Syntax

Dieser Funktionsaufruf liefert nicht wie gewohnt eine Zeichenkette, sondern ein Array-Objekt.



[4] Sie wurde von *Rony G. Flatscher*, Professor an der Wirtschaftsuniversität Wien, schon vor längerer Zeit veröffentlicht. Der hier beschriebene Stand ist 2018-09-30.

Das erste Argument *datarray* zeigt auf die zu sortierenden Daten. Es ist ein Objekt vom Typ Array. Jedes Array-Element entspricht einer Datenzeile. Rechts von *datarray* folgt mindestens eine Argumentengruppe *sortfeld*, die jeweils ein Feld innerhalb der Zeile definiert, nach dessen Inhalt sortiert werden soll.
Als erstes Argument innerhalb *sortfeld* gibt *start* die linke Grenze des Sortierfeldes (Spalte) an. Dem folgt dessen *länge* (Anzahl der Spalten).[5] Voreinstellung ist *bis zum Ende*.
Als nächstes Argument bewirkt ein Buchstabe `A` oder `D` (descending) wie gewohnt auf- oder absteigende Sortierung. Zuletzt kommt das Argument des Sortiermodus. `I` ignoriert Groß- und Kleinschreibung, während `C` (case) Großbuchstaben vor Kleinbuchstaben einsortiert. Modus `N` bewirkt Anwendung der numerischen Sortierung auf dieses Sortierfeld, wie oben beschrieben. Die Voreinstellung für diese Argumente ist `A` und `I`.
sort2 erzeugt ein neues Array-Objekt *neuarray* mit den sortierten Daten.[6] Es muss nicht vorher explizit angelegt werden.
Standardmäßig ist auch das Quellarray *datarray* nach dem Aufruf sortiert, hat also dieselbe Zeilenfolge wie *neuarray*. Sollte das nicht gewünscht sein, hängt man an den Arraynamen die Methode `~copy` an. Dann sieht **sort2** nur eine temporäre Arraykopie.
Dieser `copy`-Schritt erzeugt letztlich ein *drittes* Array. Er sollte daher nur verwendet werden, falls die weitere Programmlogik auf die unsortierte Form von *datarray* angewiesen ist.

Von der Stammvariable zum sortierten Array-Objekt und zurück

Existierende Rexx-Programme, in denen man die neuen Möglichkeiten von **sort2** nutzen möchte, dürften ihre Daten oft als durchgezählte Stammvariable (Seite 30) vorliegen haben. Wir verwenden folgendes Beispiel (Daten nach *MeteoSchweiz*):

```
/*         n Stationsname       Hoehe Luftdruck Temp  */
/*         ....+....1....+....2....+....3....+....4.. */
zeile.1 = '1 Aigle               381   972.0    10.1'
zeile.2 = '2 Col du St-Bernard  2472   751.8    -0.6'
zeile.3 = '3 Jungfraujoch       3580   654.9    -7.2'
zeile.4 = '4 Koppigen            485   960.3    +8.8'
zeile.5 = '5 Neuchatel           485   9.602E2  10.2'
zeile.6 = '6 Waedenswil          485     960.1   9.5'
zeile.7 = '7 Zermatt            1638   834.7     4.2'
zeile.0 = 7
```

Die oben erwähnte Umwandlung solcher Daten ist sehr einfach, da die Objektart *Array* ebenfalls mit den natürlichen Zahlen als Index arbeitet.

```
meteo  = .array~new          -- Array meteo anlegen

do i=1 to zeile.0            -- Elemente 1...7
     meteo[i] = zeile.i      -- mit Daten füllen
     end i
```

Wir wählen für das Array einen beliebigen Namen, hier `meteo` und legen es neu an. Dann werden alle Elemente aus der Stammvariable in dieses Array kopiert. Man beachte die unterschiedliche Index-Syntax. Beim Array folgt die Indexnummer *ohne Punkt* in eckigen Klammern: `meteo[i]` – bei der Stammvariable `zeile.i` trennt dagegen ein Punkt die Indexnummer ab.[7] Die Elemente können mit einer gewöhnlichen Schleife kopiert werden.
Ein Sortierbeispiel mit diesen Daten zeigt die folgende Seite:

[5] Dies gleicht anderen Rexx-Funktionen, ist aber ein Unterschied zu **SysStemSort** und dem Hessling-Editor, wo auch die rechte Grenze des Sortierfeldes als Spalten-Nummer anzugeben ist.
[6] Wird die Funktion über CALL aufgerufen (S. 48), ist die erzeugte Systemvariable RESULT nicht wie üblich eine Zeichenkette, sondern ein Objekt vom Typ Array.
[7] Zur davon unabhängigen Sonderfunktion von [] bei der Ansteuerung von Stammvariablen siehe Seite 30.

```
sortmeteo = sort2(meteo~copy,22,4,'D')

--  erzeugtes Array sortmeteo
--  3 Jungfraujoch       3580   654.9    -7.2
--  2 Col du St-Bernard  2472   751.8    -0.6
--  7 Zermatt            1638   834.7     4.2
--  4 Koppigen            485   960.3    +8.8
--  5 Neuchatel           485   9.602E2  10.2
--  6 Waedenswil          485     960.1   9.5
--  1 Aigle               381   972.0    10.1
```

Beispielaufgabe: Array `meteo` soll absteigend nach der Stationshöhe umsortiert werden. Dieser Wert steht in den Spalten 22 bis 25. Das Sortierfeld beginnt also in Spalte 22 und ist 4 Zeichen lang. Buchstabe D (descending) bewirkt absteigendes Sortieren. Da die Zahlen rechtsbündig ausgerichtet sind und keine Vorzeichen präsent sind, ist der numerische Modus nicht erforderlich. Genau genommen ist hier die Voreinstellung I für den Sortiermodus aktiv, die auf Ziffern keine Wirkung hat.
Das Ergebnis der Sortierung wird in Array `sortmeteo` gespeichert.
Wie oben erwähnt, wird zugleich das Quellarray `meteo` sortiert. Soll es unverändert bleiben, ist seinem Namen der Methodenaufruf **~copy** anzuhängen. Dieser Kunstgriff wird hier gezeigt, ist aber für das Beispiel ohne Belang.

```
drusort = sort2(meteo,22,4,'D' ,,28,9,'A','N' )

--  Array drusort mit 2. Sortierfeld
--  3 Jungfraujoch       3580   654.9    -7.2
--  2 Col du St-Bernard  2472   751.8    -0.6
--  7 Zermatt            1638   834.7     4.2
--  6 Waedenswil          485     960.1   9.5
--  5 Neuchatel           485   9.602E2  10.2
--  4 Koppigen            485   960.3    +8.8
--  1 Aigle               381   972.0    10.1
```

Als Beispiel für den numerischen Modus wird ein weiteres Sortierfeld hinzugefügt: der Luftdruck mit aufsteigender Sortierung. Die Druckangaben sind nicht spaltengleich untereinander angeordnet, sondern verteilen sich auf die Spalten 28 bis 36. Somit muss der numerische Sortiermodus N aktiviert werden, um korrekte Sortierung zu erreichen. Startspalte ist 28, Feldlänge ist 9 Zeichen. Das Ausgabearray soll `drusort` heißen.
Vor dem neuen Startspalten-Argument stehen zwei Kommas als Trenner. Das erste markiert das nicht angegebene ICN-Argument der vorigen Definition, wo die Voreinstellung I wirksam war. Das zweite Komma markiert den Beginn der neuen Felddefinition.
Im Ergebnis ist zu erkennen, dass trotz fehlender Spaltenausrichtung und Benutzung der Exponentialdarstellung die drei Druckwerte der 485 m hoch gelegenen Stationen korrekt aufsteigend wie 960.1, 960.2 und 960.3 sortiert sind. Es bleibt nun noch, das sortierte Array `drusort` in die ursprüngliche Stammvariable zurück zu laden. Dafür gibt es drei funktional gleichwertige Lösungen.

```
-- Zurückkopieren des sortierten Arrays drusort
-- in die durchgezählte Stammvariable zeile.

-- Alternative 1: herkömmliche Schleife

do i=1 to zeile.0
     zeile.i = drusort[i]
     end i
```

Für die herkömmliche Schleife mit Iterationsvariable muss die Anzahl der zu kopierenden Elemente explizit bekannt sein. Diese Zahl `zeile.0` wurde schon am Anfang bei den Beispieldaten definiert.

```
                    -- Alternative 2: DO ... OVER Schleife

                    do counter i elem over drusort
                         zeile.i = elem
                         end elem

                    -- Alternative 3: DO WITH ... OVER Schleife

                    do with index i item wert over drusort
                         zeile.i = wert
                         end
```

Die beiden neuen Schleifenarten (siehe S. 20) verarbeiten automatisch alle im Array vorhandenen Elemente. Es muss aber ein Zähler für die Ansteuerung der Elemente in der Stammvariable mitgeführt werden. Der Zählername ist frei wählbar; alle drei benutzen **i**. Die Variablennamen für das aktuelle Element, hier **elem** und **wert**, sind ebenfalls frei wählbar.

Hinweis

Das vorstehende Beispiel ist dazu gedacht, den Einbau der durch **sort2** gebotenen Sortiermöglichkeiten in existierende Programmen darzustellen, *ohne deren interne Struktur umkrempeln zu müssen*. Wer auf der „grünen Wiese“ anfängt und nicht auf die Eigenschaften von Stammvariablen angewiesen ist, wird seine Daten von vorherein in einem Array speichern, falls fortgeschrittenes Sortieren beabsichtigt ist.
Die Schleifen können beispielsweise unmittelbar auch für das Lesen, Sortieren und Schreiben von Dateien eingesetzt werden. Es ist nur die jeweilige Zeile mit der „=“-Zuweisung durch einen Aufruf von **linein** oder **lineout** zu ersetzen. So lassen sich Daten ohne Umwege mit einem –dank **sort2** komfortabel sortierbaren– Array austauschen.
Die Programmbibliothek enthält übrigens auch eine **stablesort2** genannte Funktion. Aus dem im 2. Absatz auf Seite 37 genannten Grund sortiert **sort2** ebenso stabil.

Herunterladen von rgf_util2.rex

Die jeweils aktuelle Version der Programmbibliothek liegt BSF4ooRexx bei (siehe S. 50). Sie kann über den ersten Link angezeigt und heruntergeladen werden.
Die beiden anderen Links führen zur Wirtschaftsuniversität Wien. Der zweite bringt die Dokumentation, die Prof. Flatscher erstmals auf dem Internationalen Rexx Symposium 2009 präsentiert hat (39 Seiten). Der dritte führt zu einer Liste der implementierten Funktionen (3 Seiten):

```
sourceforge.net/p/bsf4oorexx/code/HEAD/tree/trunk/bsf4oorexx.dev/bin/rgf_util2.rex

wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_RGF_UTIL2-20100806-article.pdf
wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_rgf_util2.pdf
```

Die Datei `rgf_util2.rex` sollte in das ooRexx-Installationverzeichnis kopiert werden. Dazu sind Administratorrechte erforderlich. Um die Funktionen der Bibliothek nutzen zu können, muss das aufrufende Programm die folgende Direktive hinter der letzen Codezeile enthalten:

```
::REQUIRES rgf_util2.rex
```

10 Einige Grundregeln

10.1 Wie Rexx Programmzeilen liest

Alles was nicht in Hochkommas eingeschlossen ist, setzt Rexx in GROSSBUCHSTABEN um, bevor es verarbeitet wird. Schreibweisen wie **wordpos** oder **WordPos** dienen nur der leichteren Lesbarkeit durch den Menschen. Rexx sieht hier **WORDPOS**.
Das folgende Schema zeigt, wie Rexx eine Programmzeile interpretiert:

```
-- Voraussetzung für das Beispiel: Variablenzuweisung vorher im Programm
zeitzone = 'Sommerzeit'

-- im Programm steht die Zeile:
say 'Es ist'   jetzt      time() "Uhr   "     zeitzone

-- 1. Außerhalb von Hochkommas gilt: alles wird in GROSSBUCHSTABEN umgesetzt
--    und alle Leerstellen-Ketten werden auf 1 Leerstelle reduziert:
SAY 'Es ist' JETZT TIME() "Uhr   " ZEITZONE

-- 2. Funktionsaufrufe und Variable werden durch ihre Rückgabedaten/Inhalte ersetzt:
SAY 'Es ist' JETZT 15:19:54 "Uhr   " Sommerzeit

-- 3. Schlüsselwörter ausführen. SAY gibt am Bildschirm aus:
Es ist JETZT 15:19:54 Uhr    Sommerzeit
```

- **SAY** erkennt der Interpreter als Rexx-Schlüsselwort. Es bewirkt abschließend die Ausgabe des Zeileninhalts am Bildschirm.[1]
- Die in Hochkommas (') eingeschlossene Zeichenkette bleibt unangetastet.
- **JETZT** ist weder als Variable noch anderweitig bekannt und bleibt deshalb so stehen.
- **TIME()** ist ein Funktionsname. Die von der Funktion zurückgelieferte Zeichenkette wird an dieser Stelle eingesetzt.
- Die in doppelte Hochkommas (") eingeschlossene Zeichenkette bleibt unangetastet.
- **ZEITZONE** ist als Variable bekannt und wird durch ihren Wert **Sommerzeit** ersetzt.

Zusammenziehungen

```
dateiname = 'liesmich'
erweita   = 'txt'
erweitb   = '.dat'

say dateiname'.abc'        ⇒    liesmich.abc      -- Variable und Zeichenkette
say dateiname'.'erweita    ⇒    liesmich.txt      -- Zeichenkette zwischen Variablen
say dateiname||erweitb     ⇒    liesmich.dat      -- zwei Variable
say 'DAT'random()erweitb   ⇒    DAT152.dat        -- Zeichenkette, Funktion, Variable
```

Sind keine Leerstellen erwünscht, dürfen Zeichenketten und Variable sich unmittelbar berühren. Dasselbe gilt für Zeichenketten und Funktionsaufrufe. Variablen werden miteinander oder mit einem nachfolgenden Funktionsaufruf durch das Zeichenpaar || zusammengezogen.
Folgt auf eine Zeichenkette unmittelbar eine Variable, sollte sie nicht x oder b heißen, damit der Interpreter nicht versucht, sie als Hexa- oder Binärschreibweise zu lesen.

[1] Steht ein unbekanntes Wort am Anfang einer Zeile, wird die Zeile wie eine Eingabe in der Kommandozeile an Windows weitergegeben; siehe Seite 48

Zeilenfortsetzung

```
say 'Dieser Text' ,
    'steht logisch in einer einzigen Zeile'

say 'Dies' ; say 'sind logisch' ;  say 'drei Zeilen.'
```

Das *Komma* als letztes Zeichen einer Zeile fungiert als ihr Fortsetzungszeichen. Es kann an jeder Stelle stehen, an der eine Leerstelle erlaubt ist. In der resultierenden logischen Zeile steht an seiner Stelle ein Leerzeichen.
Umgekehrt kann das *Semikolon* als Zeilenende-Zeichen verwendet werden. Alle folgenden Zeichen stehen für Rexx in einer neuen Programmzeile.

Kommentare

```
/*   Auf Mainframe, OS/2 und DOS musste die 1. Zeile ein Kommentar sein    */
/*
   auch mehrzeilig        /* Kommentar im Kommentar geht, wenn vollständig */
   ist möglich
*/
say 'Heute ist der' date() /* oder mitten in der Zeile */ time()
-- macht diese Zeile zum Kommentar  (eingeführt mit ooRexx 3.0)
say 'Heute ist der' date()       -- macht den Rest der Zeile zum Kommentar
```

Die mitgelieferten Beispielprogramme haben `#!/usr/bin/env rexx` als erste Zeile. Das ist eine Anweisung aus der *nix-Welt, wo sie *shebang* heißt. Sie wird in Windows ignoriert.

Normale und strikte Vergleichsoperationen

Logische Vergleiche[2] liefern entweder 0 für *falsch* oder 1 für *wahr*, wie von den Schlüsselwörtern **if, when, while, until** benötigt.

```
Vergleichsoperatoren         strikte Vergleichsoperatoren
<  <=   =   >=  >  \=           << <<=  == >>=  >> \==
```

Falls auf beiden Seiten des Operators eine gültige Zahl steht, erfolgt der Vergleich numerisch.[3] Anderenfalls erfolgt ein Zeichenkettenvergleich, bei dem führende Leerstellen[4] ignoriert werden. Leere oder kürzere Zeichenketten werden rechts mit Leerstellen aufgefüllt.
Die **strikten** Operatoren vergleichen jedes Zeichen für sich. Unterschiedlich viele Leerstellen sind dann ungleich; 1 und 1.0 ebenso. Das **when** des in ooRexx 5.0 neuen **select case** (Beispiel S. 37) macht immer strikte **==** Vergleiche.
Vergleiche können logisch mit & (UND), | (ODER) sowie && (exklusives ODER) verknüpft sein. Es werden immer *alle* verknüpften Vergleiche einer Anweisung ausgeführt.
Ausnahme: seit ooRexx 3.2 gibt es das **Komma** als bedingtes UND (Beispiel S. 19 unten). Sobald hierbei eine Vergleichsoperation 0 liefert, werden die rechts davon stehenden nicht mehr ausgeführt.

```
if lines(datei) = 1 then ...      -- gleichwertig:   if  lines(datei) then ...
if lines(datei) = 0 then ...      -- gleichwertig:   if \lines(datei) then ...
```

Liefert eine Funktion oder Variable genau 0 oder 1, kann hinter **if, when, while, until** die Vergleichsoperation entfallen. Zeichen \ steht für die Negation.

---

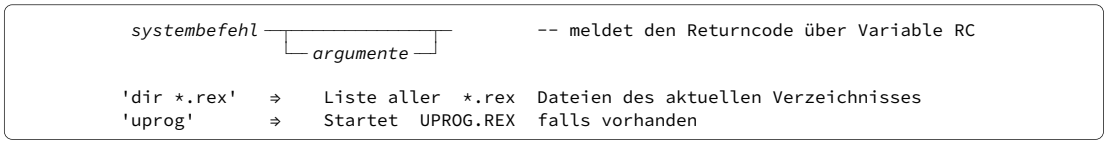
[2] Für die dargestellten 12 Operatoren gibt es weitere 12 alternative Schreibweisen mit gleicher logischer Bedeutung, die hier weggelassen sind.

[3] Vorsicht. Da manche hexadezimalen Werte wie 0E01 von Rexx für numerisch gehalten werden, denn in Exponentialdarstellung ist $0E01 = 0 \cdot 10^1 = 0$, sollten hexadezimale Zeichenketten immer strikt verglichen werden.

[4] Genauer gesagt, alle Zeichen der Klasse SPACE, wie bei **xrange** auf Seite 9 beschrieben.

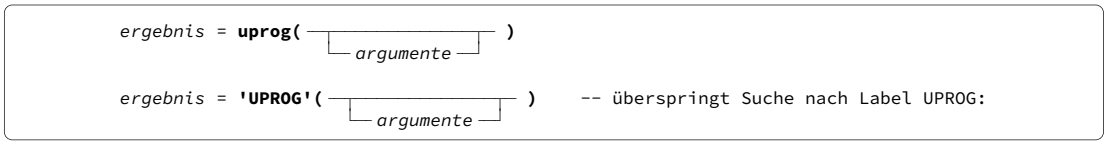
10.2 Aufrufen von Unterprogrammen

Ausführung wie in der Kommandozeile



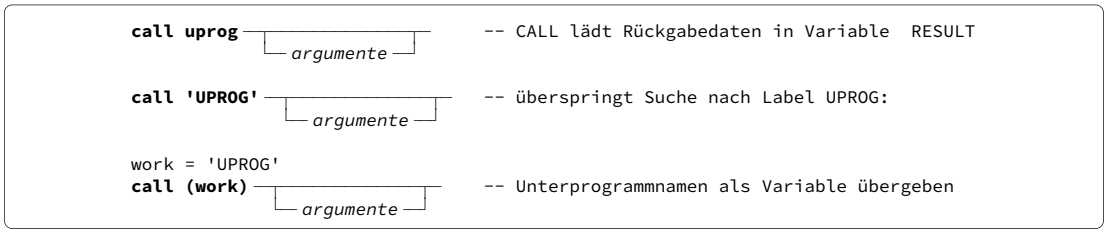
Wenn Rexx nach dem Einsetzen von Variableninhalten und Funktionswerten ein unbekanntes Wort am Zeilenanfang findet, wird diese Zeile als Systembefehl aufgefasst und ans Betriebssystem weitergeleitet. Dort wird sie wie eine Eingabe in der Kommandozeile behandelt. Windows sucht in der Reihenfolge: Systembefehl, EXE-Datei, BAT-Datei, CMD-Datei und REX-Datei. Bei vergeblicher Suche erscheint die Windows-Meldung; *„...falsch geschrieben oder konnte nicht gefunden werden."*
Auf diese Weise lässt sich ein Rexx-Programm starten wie in der Kommandozeile. Es kann über das Schlüsselwort `exit` eine positive oder negative ganze Zahl als Information ans aufrufende Programm zurückgeben. Dieser sogenannte Returncode ist bei normaler Ausführung 0. Ansonsten kann die Art des Fehlers durch eine vereinbarte Zahl mitgeteilt werden, wie zum Beispiel bei den Windows-Returncodes (siehe Seite 50). Der Returncode ist immer als Variable **RC** im aufrufenden Programm verfügbar.

Aufruf als Funktion



Ein Funktionsaufruf ist erkennbar an der Klammer unmittelbar hinter dem Namen. Im Gegensatz zur Kommandozeile[5] können *mehrere*, durch Komma *logisch* geteilte Zeichenketten übergeben werden (*Argumentenkette*). Siehe Syntaxbeispiel S. 52 oben.
Soll ein Unterprogramm als Funktion aufrufbar sein, **muss** es mit dem Schlüsselwort `return` eine –auch leere– Zeichenkette ans aufrufende Programm zurückliefern.
Im aufrufenden Programm **muss** der Funktionsaufruf in einer Codezeile stehen, welche die von der Funktion gelieferten Daten verarbeitet. Ansonsten wird Rexx den Rückgabewert als vermeintlichen Befehl an Windows weiterreichen.

Aufruf mit CALL

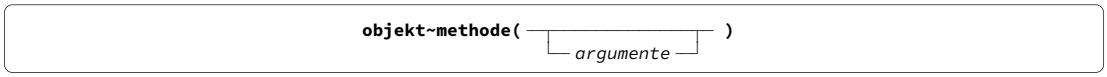


Ein Aufruf mit dem Schlüsselwort `call` unterscheidet sich vom Funktionsaufruf –abgesehen vom Verzicht auf Klammern– dadurch, dass das Zielprogramm keine Daten zurückliefern muss. Wenn doch, tut es das ebenfalls mit `return`. Die Rückgabedaten sind dann im aufrufenden Programm als Variable **RESULT** verfügbar. Das Komma wird wie bei Funktionen benutzt.

[5] Die Kommandozeile sieht ein Komma als Zeichen wie jedes andere, also ohne besondere Steuerfunktion.

Nur der Aufruf über `call` bietet die Möglichkeit eines variablen Programmnamens. Dazu ist die Variable in Klammern zu setzen.

Methodenaufruf



Aus Sicht der Handhabung in einem klassischen Programm stellen Methoden eine Kombination der beiden vorigen Aufrufarten dar:

- Für Methoden ist es im Gegensatz zu Funktionen zulässig, keine Daten zurückzuliefern.
- Liefert die Methode Daten zurück, werden diese wie bei einen Funktionsaufruf an dessen Stelle in die Programmzeile eingefügt.
- Im Gegensatz zum *Funktions*aufruf kann ein *Methoden*aufruf problemlos auch allein in einer Zeile stehen. Zurückgegebene Daten werden dann ignoriert. Sie stehen dem aufrufenden Programm aber in Variable **RESULT** zur Verfügung.

Für die Suchreihenfolge kommt es auf die aktuelle Klassenhierarchie an.

10.3 Suchreihenfolge für Funktionen und CALL

Abgesehen von der Bedingung, dass eine Funktion einen Wert zurückliefern muss, hat man die freie Wahl zwischen der Syntax als Funktionsaufruf oder CALL. Das aufzurufende Programm kann sowohl ein Bestandteil von Rexx sein, als auch eine Eigenentwicklung oder ein Zusatzprodukt. Für beide Aufrufarten

```
uprog()
call uprog
```

ist die Suchreihenfolge identisch.[6] Beim ersten JA auf eine der folgenden Fragen verzweigt die Verarbeitung dorthin:

- Gibt es ein Label **uprog:** in der aktuellen Programmdatei? Dieser Schritt wird übersprungen, wenn der gesuchte Name in einfachen oder doppelten Hochkommas steht.
- Ist **uprog** der Name eines Bestandteils von Rexx? Das sind
  - die unmittelbar zur Sprache gehörenden Funktionen[7] und
  - die in Bibliothek **rexxutil.dll** mitgelieferten **Rx...** und **Sys...** Funktionen.[8]
- Gibt es in der aktuellen Programmdatei eine Direktive **::ROUTINE uprog**?
- Gibt es in einer vom aktuellen Programm per ::REQUIRES einbezogenen Datei eine Direktive **::ROUTINE uprog PUBLIC**?
- Ist im *Rexx Macrospace* unter den mit Suchfolge **Before** geladenen Programmen eines mit dem Namen **uprog**?
- Wurde per **::REQUIRES *name* LIBRARY** ein *externes Funktionspaket* (DLL-Datei) geladen, in dem **uprog** enthalten ist?
- Gibt es eine Datei mit dem Namen **uprog.rex** ...
  - im aktuellen Verzeichnis oder
  - in einem Verzeichnis der PATH Umgebungsvariable – dazu gehört immer auch das Installationsverzeichnis von ooRexx?

[6] Zur besseren Lesbarkeit ist hier die Rexx-interne Umsetzung aller Namen in Großbuchstaben weggelassen.
[7] Kapitel 7.4 *„Built-in Functions“* in Datei *rexxref.pdf*
[8] Kapitel 8 *„Rexx Utilities“* in Datei *rexxref.pdf*

- Ist im *Rexx Macrospace* unter den mit Suchfolge **After** geladenen Programmen eines mit dem Namen **uprog**?
- Das Programm stoppt und meldet: *Error 43: Could not find Routine UPROG*

Was ist ein Rexx Macrospace?

Rexx-Programme können in den Hauptspeicher (RAM) des Betriebssystems geladen werden. Das spart beim Aufruf Plattenzugriffe. Die dazu notwendigen Funktionen (SysAddMacroSpace und andere) sind Teil der Rexx Utilities.
Diese Methode ist vom Mainframe übernommen, wo sie unter der Bezeichnung *Nucleus Extension* sehr wirksam war. ooREXX ist auf modernen Notebooks auch ohne diesen Kunstgriff sehr schnell. Daher habe ich keine praktischen Erfahrungen mit Macrospaces unter Windows.

Was ist ein externes Funktionspaket?

Rexx bietet eine Schnittstelle, um Funktionen mit C oder C++ zu erstellen, die wie jede andere Rexx-Funktion aufgerufen werden können. Sie müssen Teil einer DLL sein, die mit einer Direktive **::REQUIRES** ***name*** **LIBRARY** dem Rexx-Programm bekannt gemacht („geladen“) wird. Ein Anwendungsbeispiel ist `rxmath.dll` (siehe Seite 24).

Hinweis auf BSF4ooRexx

Unter diesem Namen stellt Professor *Rony G. Flatscher* ein Werkzeug bereit, mit dem seit ooRexx 4.1 auf alle in einer installierten **Java** Umgebung verfügbaren Funktionen zugegriffen werden kann. Es ist von **sourceforge.net** herunterladbar.

10.4 Bedeutung von Windows Returncodes

```
SysGetErrorText( — rc — )

-- SysGetErrorText(5)   ⇒    Zugriff verweigert
```

Kann Betriebssystem Windows eine Aktion nicht ausführen, wird der Grund dafür per numerischem Returncode gemeldet. Der zum Returncode gehörende Fehlermeldungstext kann über diese Funktion geliefert werden. Nachfolgend eine Auswahl von Returncodes:

```
     2  Das System kann die angegebene Datei nicht finden.
     3  Das System kann den angegebenen Pfad nicht finden.
     4  Das System kann die Datei nicht öffnen.
     5  Zugriff verweigert
    13  Ungültige Daten
 8, 14  Für den Befehl/Vorgang ist nicht genügend Speicher verfügbar.
    15  Das System kann das angegebene Laufwerk nicht finden.
    16  Das Verzeichnis kann nicht entfernt werden.
    17  Das System kann die Datei nicht auf ein anderes Laufwerk verschieben.
    18  Es sind keine weiteren Dateien vorhanden.
    19  Das Medium ist schreibgeschützt.
    23  Datenfehler (CRC-Prüfung)
    26  Auf das angegebene Laufwerk kann nicht zugegriffen werden.
    32  Der Prozess kann nicht auf die Datei zugreifen, da sie von einem anderen
        Prozess verwendet wird.
    36  Zu viele Dateien zur gemeinsamen Verwendung geöffnet.
    39  Der Datenträger ist voll.
   183  Datei mit diesem Namen existiert schon.
```

Der Rexx-Interpreter benutzt für Programmfehler eigene Returncodes. Deren Bedeutung läßt sich in gleicher Weise über die Funktion **ErrorText()** anzeigen. Anhang C der Handbuchs (Datei *rexxref.pdf*) enthält außerdem eine ausführliche Liste.

10.5 Schema einer ooRexx-Funktionsbibliothek

Erfahrungsgemäß können sich Unterprogramme ansammeln, die von mehreren Programmen benutzt werden. Dadurch entsteht eine große Zahl einzelner Rexx-Dateien. Mit der Direktive **::ROUTINE** lassen sich stattdessen Funktionsbibliotheken bilden, die mehrere Unterprogramme in einer Datei zusammenfassen.

```
-- Datei bibliothek.rex als Schema einer eigenen Funktionsbibliothek

-- Rexx-Programmzeilen vor der ersten ::DIEKTIVE bilden den Prolog
-- In diesem Fall enthält er keine ausführbaren Zeilen

-- Erste ::DIREKTIVE, hier eine NUMERIC DIGITS Voreinstellung für alle Routinen
::OPTIONS digits 16               -- neu seit ooRexx 4.0

::ROUTINE zylinder PUBLIC         -- 1. Unterprogramm zylinder
use arg argumente ...             -- Argumente mit ARG und/oder USE ARG lesen
...                               -- Rexx-Anweisungen zur Berechnung
return volumen oberfläche ...     -- Ergebnis z.B. durch Leerstellen getrennt

::ROUTINE kegel    PUBLIC         -- 2. Unterprogramm kegel
use arg argumente ...
...
zahl = uprog(abc)                 -- Aufruf des privaten Unterprogramms
...
return volumen oberfläche ...     -- Ende von kegel
uprog:                            -- privates Unterprogramm zu kegel
...                               -- das nur für kegel sichtbar ist
return wert                       -- Rückgabe des Ergebnisses

::ROUTINE pyramide PUBLIC         -- 3. Unterprogramm pyramide
use arg argumente ...
...
return volumen oberfläche ...

::ROUTINE name     PUBLIC         -- und so weiter ...
use arg argumente ...
...
return rückgabedaten
```

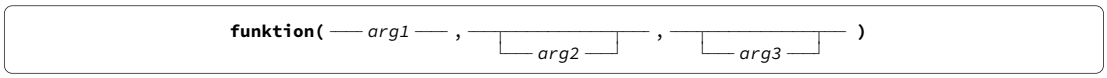
Zum Prolog siehe Seite 26. Jedes **Unterprogramm** beginnt mit einer ::ROUTINE-Direktive und endet vor der nächsten mit **::** beginnenden Direktive oder am Dateiende. Angabe der Option PUBLIC macht es von außen aufrufbar. Alle sind voneinander abgekapselt als stünden sie in separaten Rexx-Dateien. Das gilt auch für Aufrufe innerhalb derselben Bibliothek.
Interne Unterprogramme einer Routine, wie das mit dem Label `uprog:` beginnende, sind unsichtbar für die anderen Routinen. Das Unterprogramm seinerseits sieht nur die Variablen des eigenen Routine-Programms. Für **uprog** sind, wie in einer gewöhnlichen Rexx-Datei, alle Variablen von **kegel** sichtbar. Das kann explizit mit PROCEDURE und EXPOSE eingeschränkt werden. Nur **kegel** kann **uprog** benutzen.

```
-- Beispiel der Benutzung von Unterprogramm PYRAMIDE
-- aus Funktionsbibliothek BIBLIOTHEK.REX
...

call pyramide argument1 , argument2 ...

...

-- am Programmende die Funktionsbibliothek angeben:
::REQUIRES bibliothek.rex
```

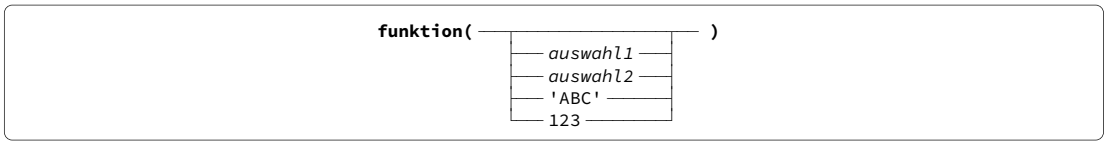
Dieses Beispiel zeigt den Aufruf von Unterprogramm **Pyramide**. Dazu muss am Ende des aufrufenden Programms per **::REQUIRES** bekannt gemacht werden, wo Rexx nach **Pyramide** suchen soll. Die Erweiterung **.rex** kann weggelassen werden. Seit Februar 2020 sucht Rexx 5.0 zuerst eine **.cls**-Datei dieses Namens und danach eine **.rex**-Datei.

10.6 Erklärung der Syntaxdiagramme

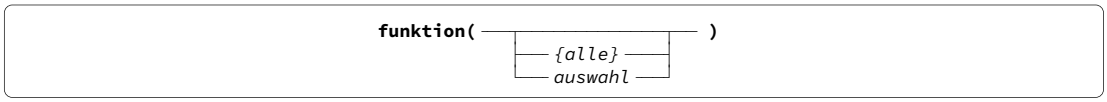
Die in dieser Kurzreferenz verwendeten Syntaxdiagramme sind folgendermaßen aufgebaut:



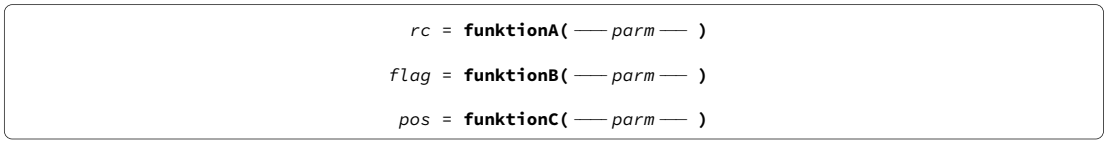
Dieser Funktion können beim Aufruf durch **Komma** getrennte *Argumente* (auch Parameter genannt) als *Argumentenkette* mitgegeben werden. Dabei ist im Beispiel *arg1* zwingend, während die anderen beiden auch weggelassen werden können.
Die trennenden **Kommas** sind der Einfachheit halber immer dargestellt. Es gilt die Regel, dass *rechts vom letzten benutzten Parameter alle Kommas weggelassen werden können.*
Die *Kursivschrift* zeigt dabei an, dass dies ein symbolischer Name ist. Dort steht im Programm in der Regel eine Variable, eine Zeichenkette in Hochkommas (" oder ') oder eine Zahl, die keine Hochkommas braucht.



Eine „Leiter“ zeigt, dass diese Funktion ohne Parameter, oder mit genau einem der Parameter aufgerufen werden kann. Auf der obersten Sprosse ist dabei der Vorgabewert (Default) angegeben, der wirksam ist, wenn kein Parameter mitgegeben wird.
Die Zeichenkette in Hochkomma kann anstelle einer Variablen verwendet werden, wenn dies einfacher oder leichter lesbar ist. Durch die Hochkommas wird sicher verhindert, dass statt des gewünschten Werts ABC der Inhalt einer zufällig existierenden Variablen ABC an die Funktion übermittelt wird. Das ist besonders bei kurzen Variablennamen eine typische Fehlerquelle.



Manchmal kann im Syntaxdiagramm der konkrete Vorgabewert, zum Beispiel die Länge der jeweiligen Zeichenkette, nicht angegeben werden. Dann steht dort in *{ }* eine erklärende Ersatzbezeichnung. Diese, wie im Beispiel *{alle}*, kann nicht beim Funktionsaufruf verwendet werden. Es sei denn, im konkreten Fall ist die Zahl für *{alle}* bekannt, zum Beispiel 351.



Funktionen liefern meist Daten zurück, was im Syntaxdiagramm nicht besonders vermerkt ist. Werden dagegen Statusinformationen über den Erfolg des Funktionsaufrufs zurückgegeben, ist das im Syntaxdiagramm dargestellt.
Der Returncode *rc* ist bei erfolgreicher Ausführung 0. Jede andere Zahl sollte einen Hinweis auf die Art des aufgetretenen Fehlers geben. Zu den in Windows geltenden Bedeutungen siehe Seite 50.
Heisst der Rückgabewert im Diagramm *flag*, hat er den Wert 1 für **wahr** oder 0 für **falsch**. In manchen Fällen ist auch -1 möglich.
Ein Rückgabesymbol *pos* oder *n* zeigt, dass die Funktion eine Längenangabe oder einen Zählwert liefert, also eine positive ganze Zahl oder Null.
Falsche Unterprogrammaufrufe –Syntaxfehler– stoppen das Programm.

Inhaltsverzeichnis