Open Object Rexx 5.1
Kurzreferenz für Klassiker

Jochem Peelen

Stand: 14. April 2025

Die beliebten *REXX Reference Cards* waren für meinen Geschmack immer etwas zu knapp gehalten. Ich schrieb eine Kurzreferenz für mich selbst. Daraus ist schließlich dieses Dokument geworden. Es enthält 99 % der Informationen, die ich für die Arbeit mit ooRexx brauche. Ich hoffe, dass es auch anderen, die klassisch arbeiten, Nutzen bringen kann.

Zielsetzung

- Es gilt für Windows 7 Systeme und neuere, auf denen ooRexx 5.1 installiert ist.
- Der erste Teil ist eine **kompakte** Nachschlagehilfe der Syntax oft benutzter Sprachelemente für:
  - Zeichenketten
  - Wortketten
  - Programmschleifen
  - Uhrzeit und Datum
  - Rechenfunktionen (incl. Rosettacode-Bibliothek **RXM**)
  - Verwaltung von Dateien und Verzeichnissen
  - Lesen und Schreiben von Dateien (auf „neue“ und „alte“ Art)
  - Bits und Bytes (hexadezimal, dezimal)
- Im zweiten Teil ab Seite 35 wird an Beispielen eine Einführung in die Benutzung der folgenden „Multiwerkzeuge“ gegeben:
  - Stammvariable
  - Array (incl. Methode **SORT2** aus Bibliothek **rgf_util2**)
  - **USE ARG** (incl. **::ROUTINE**) für Unterprogramme/Funktionsbibliotheken
- Den Schluss ab Seite 51 bilden Themen, die man seltener nachschlagen muss:
  - „Klassik“ versus „Objekt“ am Beispiel **~sortwith**
  - Wichtige Grundregeln des ooRexx
  - Erklärung der benutzten Syntaxdiagramme
  - Index
  - Inhaltsverzeichnis (auf der letzten Seite: ohne Blättern zugänglich)
- **ooRexx-Neugierige** können sich mit diesem Dokument einen Überblick verschaffen, was sie erwartet. Dazu empfiehlt sich zuerst ein Blick auf die Grundregeln (S. 55) und wie ooRexx mit Zahlen umgeht (S. 16).

Tip: Gibt es Unsicherheiten über das genaue Verhalten einer Funktion oder Methode bei bestimmten Eingaben, bietet das in ooRexx enthaltene interaktive Programm **rexxtry** die Möglichkeit, den Zweifelsfall am Bildschirm durchzuprobieren.

Der Inhalt dieser Schrift beruht wesentlich auf meinen eigenen Wahrnehmungen mit ooRexx 5.0 und 5.1 unter Windows (7, 8.1 und 10) und erfolgt ohne Gewähr für Richtigkeit. Der neueste benutzte Build ist r12924 vom 21. November 2024.

Die für das *36. Internationale Rexx Language Symposium 2025* in Wien erstellte englische Fassung dieser Kurzrefrerenz hat den Titel **Open Object Rexx 5.1 Classic Short Reference**.

© 1985-2025 Jochem Peelen, Greifswald
Gesetzt mit pdfLaTeX in URW Palladio; Diagramme in Adobe SourceCodePro.

1 Zeichenketten-Funktionen

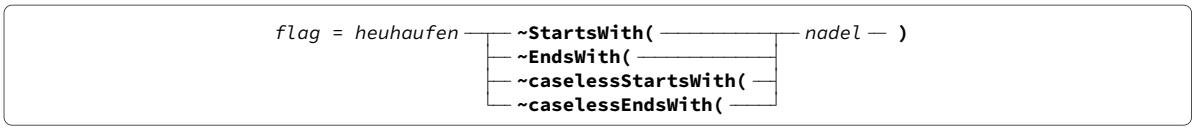
1.1 Informationen über Zeichenketten

Länge

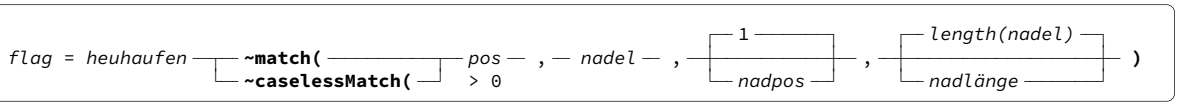
```
n  =  length( — zkette — )

-- length('ooRexx')   ⇒    6
-- length('')         ⇒    0     -- leere Zeichenkette
-- length("")         ⇒    0     -- leere Zeichenkette
```

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

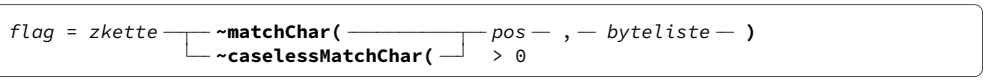
Zeichenketten prüfen (wahr/falsch)



Liefern 1, wenn Zeichenkette *heuhaufen* mit *nadel* beginnt/endet oder beide identisch sind, sonst 0.[1]



Liefert 1, wenn in Spalte *pos* von *heuhaufen* die Zeichenkette *nadel* beginnt, sonst 0. Mit *nadpos* und *nadlänge* kann erreicht werden, dass nur ein Teil von *nadel* (!) für den Vergleich benutzt wird.



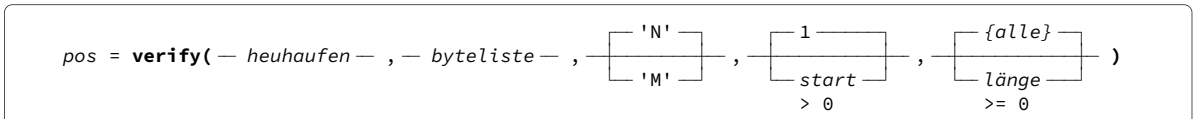
Liefert 1, wenn an Position *pos* von *heuhaufen* ein Zeichen aus *byteliste* steht, sonst 0. Liefert auch 0, falls *zkette* oder *byteliste* leere Zeichenketten sind oder *zkette* kürzer als *pos* ist.

```
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 | .false | .true
                               ├─ '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       32bit: <= 9stellig | 64bit <= 18
                               ├─ '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.

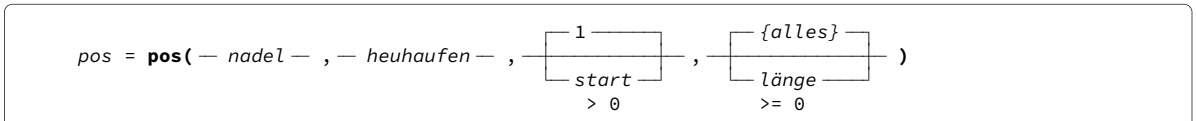
[1] **Caseless** erkennt Umlaute, ß oder diakritische Zeichen anderer Sprachen nicht als Buchstaben.

In Zeichenketten suchen

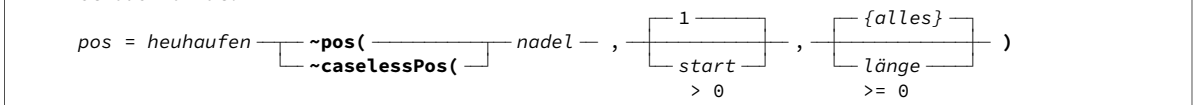



```
-- verify(4711,'0123456789')              ⇒   0    kein unerwünschtes Zeichen gefunden
-- verify(3.14,'0123456789')              ⇒   2    Position des ersten Fremdzeichens
-- verify('','0123456789')                ⇒   0    leere Zeichenkette
--
-- verify('Marder','0123456789','M')      ⇒   0    kein Zeichen der Suchliste gefunden
-- verify('Leopard2A7','0123456789','M')  ⇒   8    Position der ersten Ziffer
-- verify('Satzende. ','-!.;:','M')       ⇒   9    '.' steht an Position 9
```

Im Modus **N**[omatch] ist *byteliste* eine Liste erlaubter Zeichen. Wenn *heuhaufen* kein anderes Zeichen enthält ODER eine leere Zeichenkette ist, wird 0 geliefert. Sonst wird die Position der ersten falschen Zeichens in *heuhaufen* geliefert.
Im Modus **M**[atch] wird nach den Zeichen aus *byteliste* gesucht. Die Position des ersten Treffers in *heuhaufen* wird geliefert, sonst 0.
Mit *start* und *länge* kann die Suche auf einen Teil von *heuhaufen* beschränkt werden. Länge 0 liefert immer Ergebnis 0.

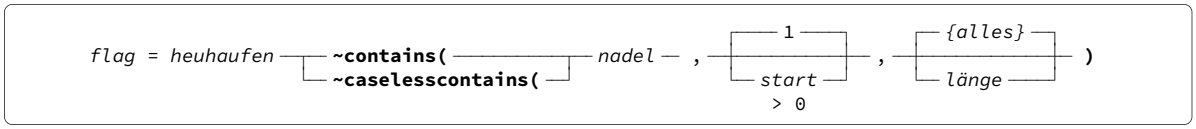



```
-- pos('9','12345678901234567890')        ⇒   9
-- pos('9','123456789012345678 90',15)    ⇒   19
-- pos('9','123456789012345678 90',15,3)  ⇒   0     Position 19 liegt außerhalb von 15-17

Methodenformat:
```

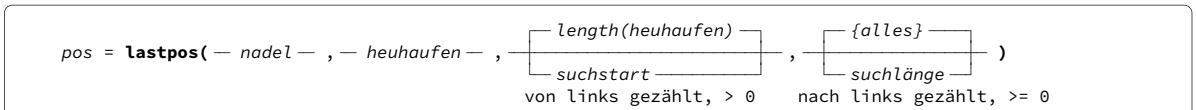



```
-- 'abcdefghijklmno'~caselesspos('DEF')   ⇒   4
```

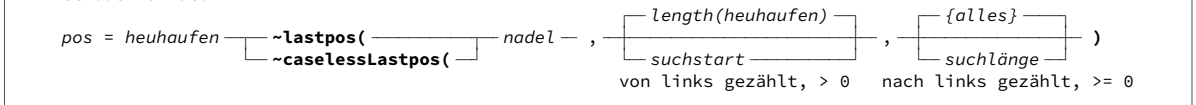
Liefert die Position des ersten Zeichens von *nadel* in *heuhaufen* oder 0 wenn *nadel* nicht in *heuhaufen* enthalten ist.



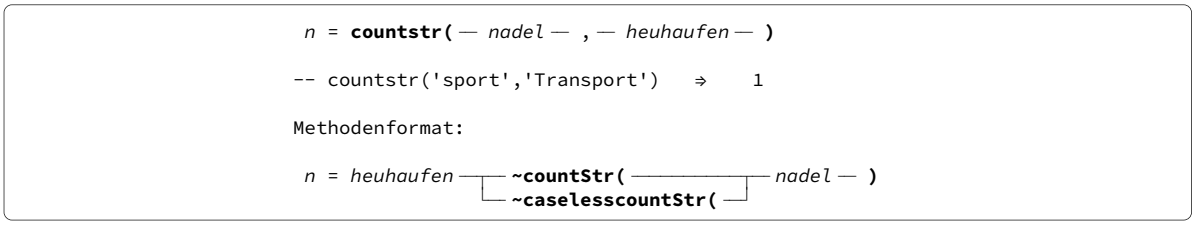
Arbeitet wie **pos()**, liefert aber nur 1 oder 0, wie in Vergleichsoperationen erwartet (Seite 56).




```
-- lastpos('9','12345678901234567890')        ⇒   19
-- lastpos('9','123456789012345678 90',15)    ⇒   9
-- lastpos('9','123456789012345678 90',15,3)  ⇒   0     Position 9 liegt außerhalb von 15-13

Methodenformat:
```

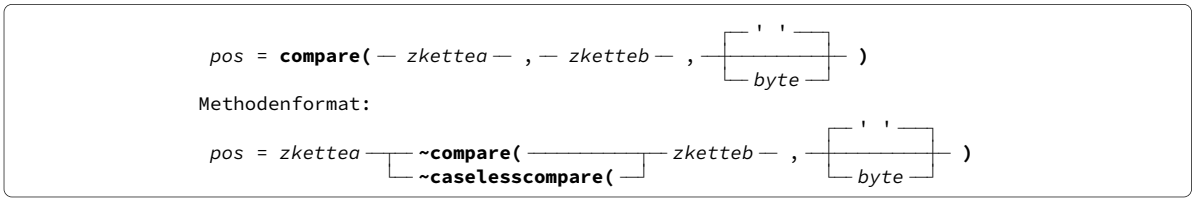


lastpos() arbeitet wie **pos()**, sucht jedoch am Ende von *heuhaufen* beginnend in Richtung Anfang. Die gelieferte Position ist aber vom Anfang aus –also links beginnend– gezählt.

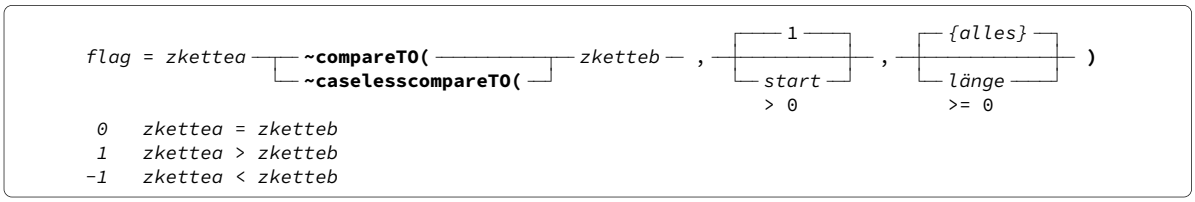


Zählt wie oft *nadel* vollständig in *heuhaufen* enthalten ist.

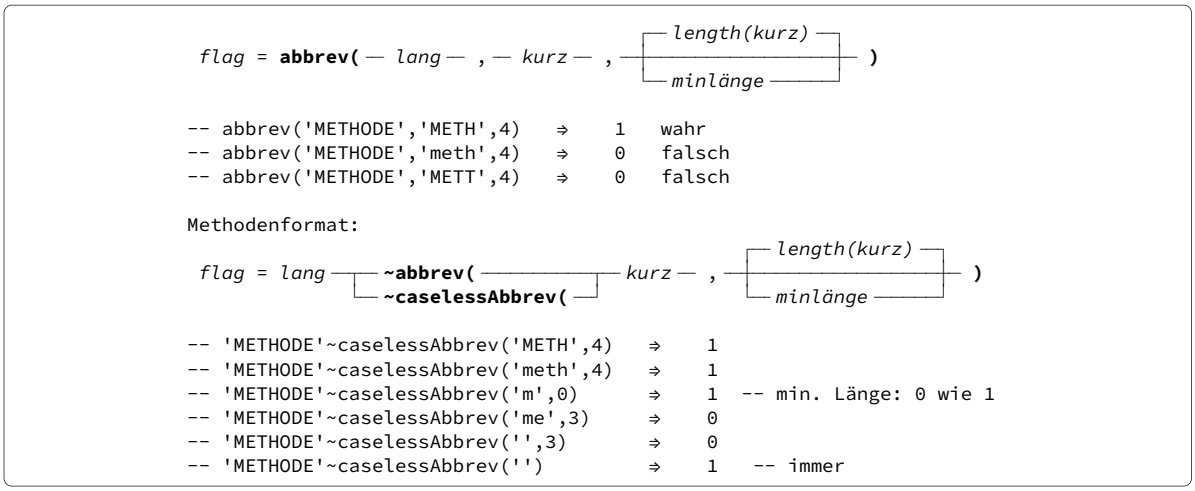
Zeichenketten vergleichen



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

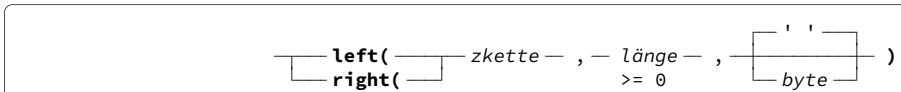


Durch 0, 1 oder -1 wird signalisiert, ob ein Unterschied zwischen den Zeichenketten besteht. Es erfolgt keine Längenanpassung. Kleiner oder größer ergibt sich aus der fiktiven Sortierfolge der Zeichenketten. Durch Angabe von *start* kann eine andere Startposition als der Anfang für den Vergleich gewählt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



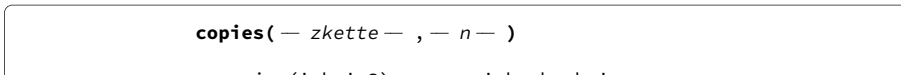
Liefert 1 wenn die Zeichen in *kurz* mit dem Anfang von *lang* übereinstimmen, sonst 0. Bei Angabe von *minlänge* wird 0 zurückgegeben, falls *kurz* kürzer ist.

1.2 Zeichenketten abschneiden oder verlängern



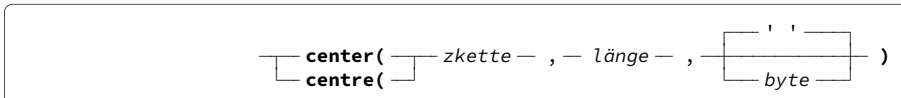

```
-- left('ooRexx',2)         ⇒    'oo'
-- left('ooRexx',12)        ⇒    'ooRexx      '
-- left('ooRexx',12,'_')    ⇒    'ooRexx______'
--
-- right('ooRexx',2)        ⇒    'xx'
-- right('ooRexx',12)       ⇒    '      ooRexx'
-- right('ooRexx',12,'_')   ⇒    '______ooRexx'
```

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('abc',3)   ⇒    'abcabcabc'
-- copies('abc',0)   ⇒    ''
-- copies('',5)      ⇒    ''          -- leere Zeichenkette bleibt leer
```

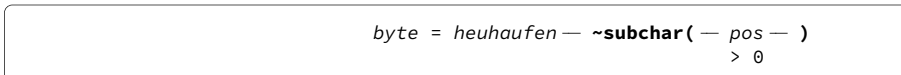
Liefert *n* Kopien von *zkette*.




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

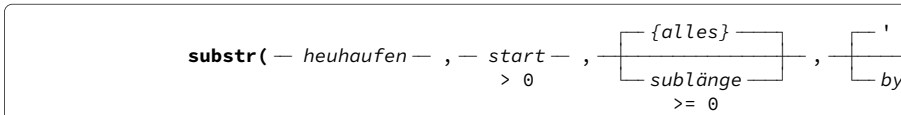
Liefert eine innerhalb von *länge* zentrierte *zkette*, die abgeschnitten oder mit Zeichen *byte* aufgefüllt ist. Muss eine ungerade Zahl von Zeichen abgeschnitten/hinzugefügt werden, wird rechts ein Zeichen mehr abgeschnitten oder hinzugefügt.

1.3 Teile von Zeichenketten lesen




```
-- 'ooRexx'~subchar(4)   ⇒    'e'
-- 'ooRexx'~subchar(7)   ⇒    ''
```

Liefert das an Stelle *pos* von *heuhaufen* stehende Zeichen oder die leere Zeichenkette, wenn *heuhaufen* kürzer als *pos* ist.




```
-- substr('abcdefgh',4)      ⇒    'defgh'
-- substr('abcdefgh',4,3)    ⇒    'def'
-- substr('abcdefgh',12)     ⇒    ''        start > length(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.

Exkurs: 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 beispiel

 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, Funktionswerte
-- oder Array-Elemente:

 parse value ─┬─ '   eins zwei    drei ' ─── with ─ zerlegeschema wie oben
              ├─ funktion(arg1,...,argn) ─┘
              └─ arrayname[n] ────────────          -- neu: Array-Element
```

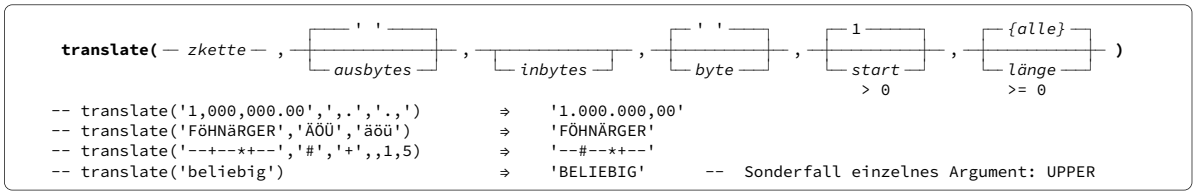
1.4 Daten in Zeichenketten ändern

Zeichen umwandeln oder löschen

```
 changestr( ── nadel ── , ── heuhaufen ── , ── neunadel ── , ┌─ {alle} ─┐ )
                                                             └─ anzahl ─┘
                                                               >= 0
-- changestr('abcd','ABCDabcdxyz','--')   ⇒    'ABCD--xyz'
-- changestr('abcd','ABCDabcdxyz','')     ⇒    'ABCDxyz'       leere Zeichenkette löscht
-- changestr('ab','abababababab','x',2)   ⇒    'xxababab'

Methodenformat:

 heuhaufen ─┬─ ~changestr( ─┐ nadel ─ , ── neunadel ─ , ┌─ {alle} ─┐ )
            └─ ~caselesschangestr( ─┘                   └─ anzahl ─┘
                                                          >= 0
-- 'ABCDabcdxyz'~CaselessChangeStr('abcd','--')   ⇒    '----xyz'
```

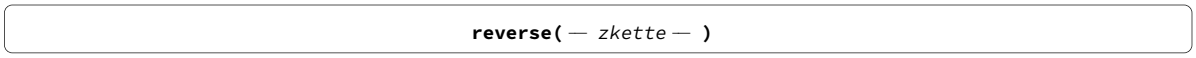
Liefert *heuhaufen*, in dem alle Vorkommen von *nadel* geändert sind in die Zeichenkette *neunadel*, die auch länger oder kürzer sein kann. Ist *neunadel* gleich der leeren Zeichenkette, werden die Vorkommen

von *nadel* gelöscht. Mit *anzahl* kann die Änderung auf die ersten Vorkommen von *nadel* beschränkt werden.

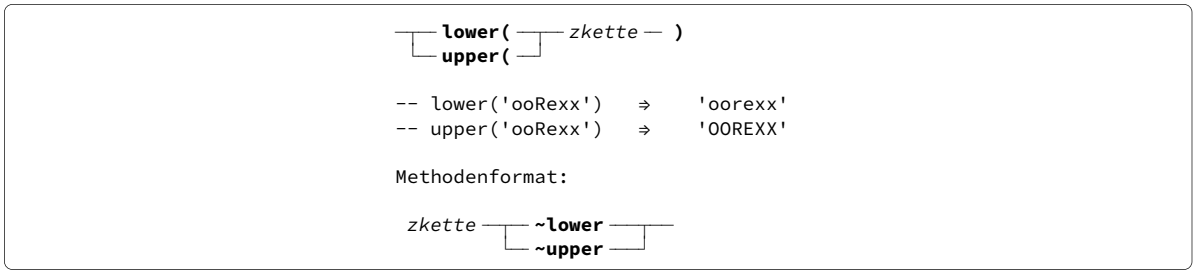



```
-- translate('1,000,000.00',',.','.,')       ⇒    '1.000.000,00'
-- translate('FöHNäRGER','ÄÖÜ','äöü')        ⇒    'FÖHNÄRGER'
-- translate('--+--*+--','#','+',,1,5)       ⇒    '--#--*+--'
-- translate('beliebig')                     ⇒    'BELIEBIG'      --  Sonderfall einzelnes Argument: UPPER
```

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.
Besteht *inbytes* aus einer Leerstelle oder der leeren Zeichenkette, erfolgt keinerlei Änderung. Fehlt Argument *inbytes* ganz, werden alle Zeichen in *zkette* durch Leerstellen ersetzt.
Ist Liste *ausbytes* kürzer als *inbytes*, wird sie mit Zeichen *byte* aufgefüllt. Besteht *ausbytes* aus der leeren Zeichenkette, werde alle in *inbyte* definierten Zeichen mit dem *byte* übrschrieben.

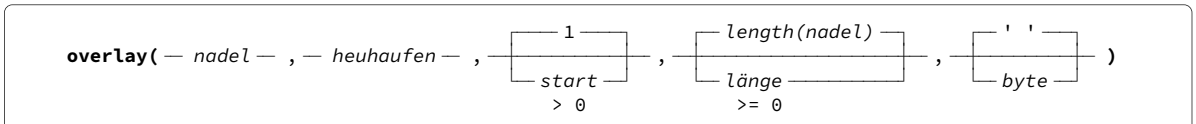


Liefert die Zeichen von *zkette* in umgekehrter Reihenfolge *ettekz*.



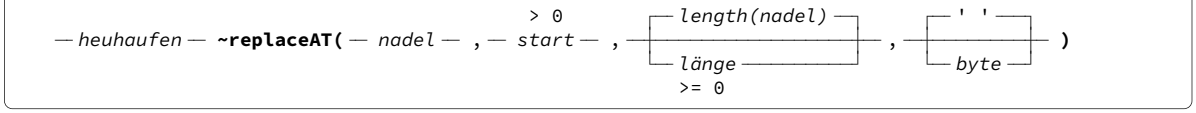
Auch hier ist zu bedenken, dass nur die gewöhnlichen 26 Buchstaben bearbeitet werden.

Positionen überschreiben, einfügen, löschen

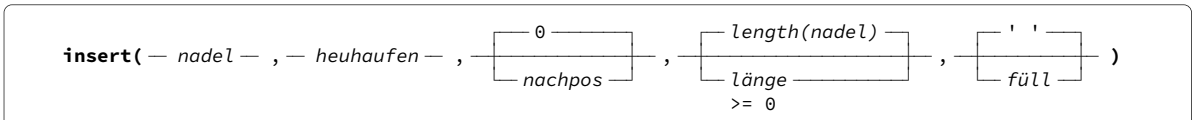



```
-- overlay('---','abcdefg')       ⇒    '---defg'
-- overlay('---','abcdefg',3)     ⇒    'ab---fg'
-- overlay('---','abcdefg',3,1)   ⇒    'ab-defg'
-- overlay('---','abcdefg',3,0)   ⇒    'abcdefg'           keine Änderung bei Länge 0
-- overlay('123','abcdefg',10)    ⇒    'abcdefg  123'      start > length(heuhaufen)
-- overlay('','abcdefg')          ⇒    'abcdefg'           keine Änderung
```

Format der äquivalenten Methode **replaceAT** (Argument *start* ist hier nicht optional!):

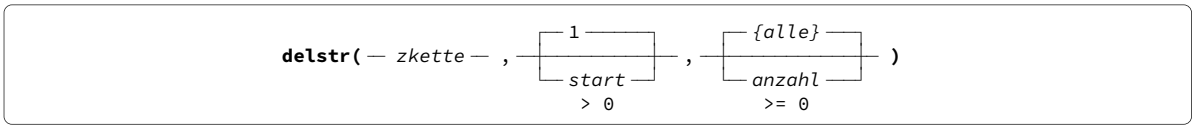


Liefert *heuhaufen*, der ab Position *start* überschrieben ist mit Zeichenkette *nadel*. Falls *nadel* eine leere Zeichenkette ist, erfolgt keine Änderung. Ist *länge* größer als *nadel*, wird mit Zeichen *byte* aufgefüllt, ebenso wenn *start* die Länge von *heuhaufen* übertrifft.



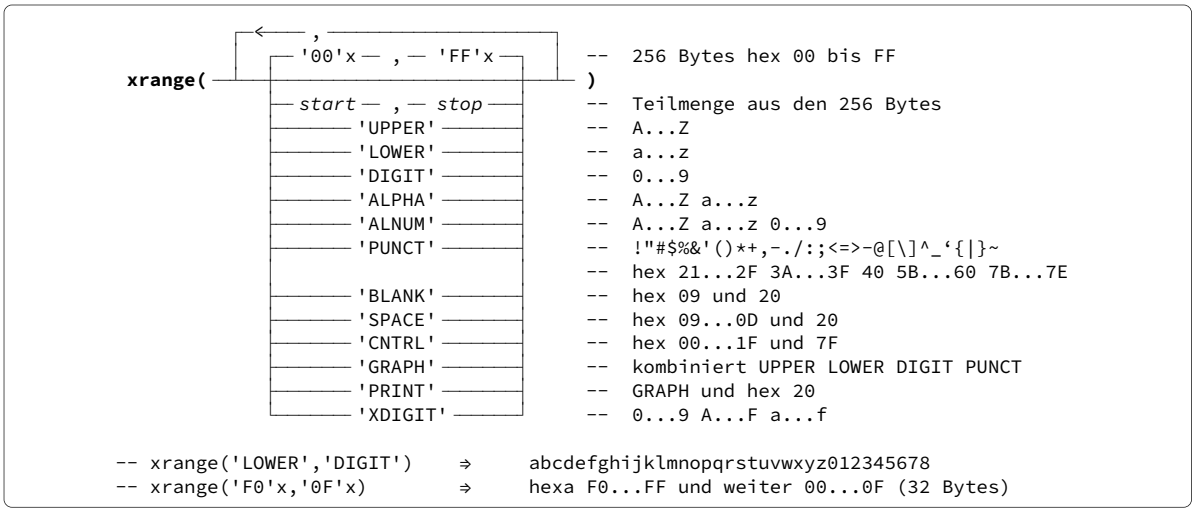

```
-- insert('---','abcdefg')       ⇒    '---abcdefg'
-- insert('---','abcdefg',3)     ⇒    'abc---defg'
-- insert('---','abcdefg',10)    ⇒    'abcdefg   ---'
-- insert('---','abcdefg',3,1)   ⇒    'abc-defg'
-- insert('','abcdefg')          ⇒    'abcdefg'         nadel darf leere Zeichenkette sein
-- insert('','abcdefg',3,5,'_')  ⇒    'abc_____defg'    5 Füllbytes eingefügt
-- insert('---','abcdefg',3,0)   ⇒    'abcdefg'         keine Änderung bei Länge 0
```

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.

1.5 Klassen von Zeichenketten

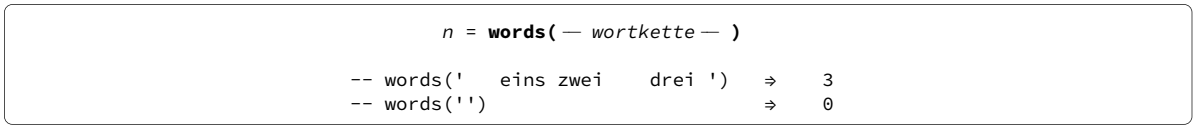



```
-- xrange('LOWER','DIGIT')    ⇒    abcdefghijklmnopqrstuvwxyz012345678
-- xrange('F0'x,'0F'x)        ⇒    hexa F0...FF und weiter 00...0F (32 Bytes)
```

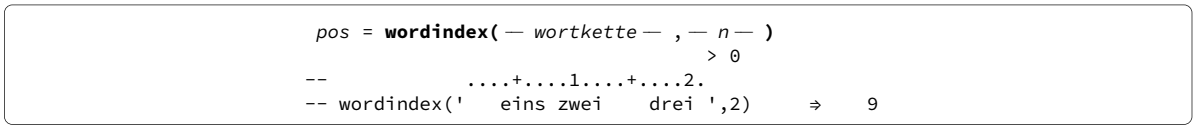
Liefert eine Zeichenkette mit den Bytes der gewünschten Klassen. Ist bei der Angabe von zwei einzelnen Bytes das Zeichen *start* größer als *stop*, besteht das Ergebnis aus den Bytes von *start* bis hexa `FF`, denen ohne Zwischenraum die Bytes von hexa `00` bis *stop* folgen.

2 Wortketten-Funktionen

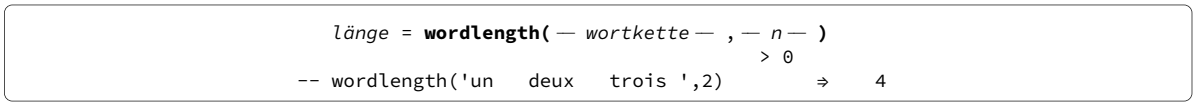
Wortketten enthalten **Leerstellen** (hexa 20) oder **Tabulatorzeichen** (hexa 09) als Trennzeichen. Als „Wort“ gilt jedes andere Zeichen oder eine Kette anderer Zeichen.



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

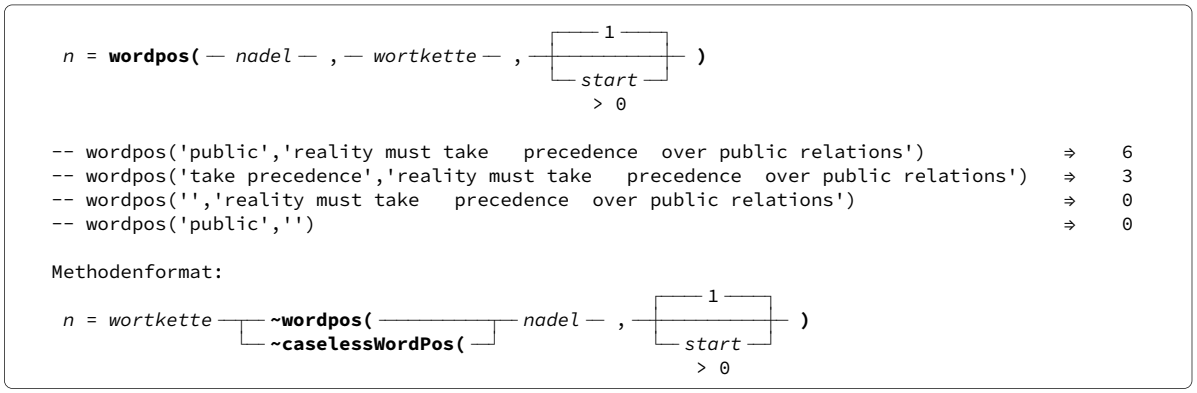


Liefert die Position in *wortkette* an der das *n*-te Wort beginnt.

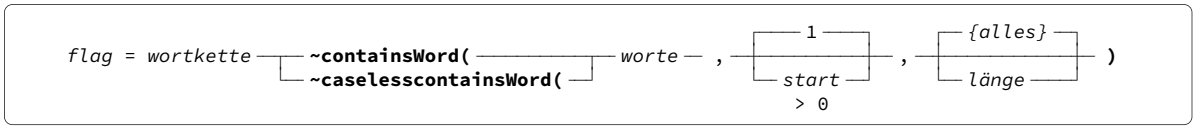


Liefert die Länge des *n*-ten Wortes in *wortkette*.

2.1 In Wortketten suchen

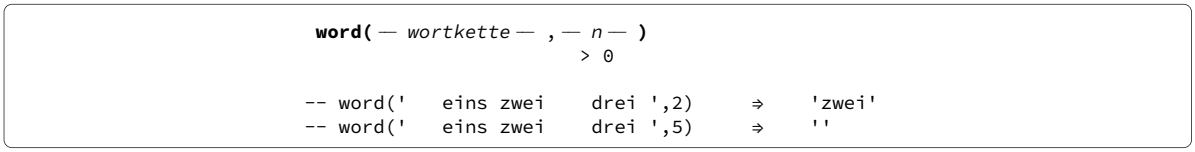


Liefert 0 falls *nadel* nicht in *wortkette* vorkommt, sonst die Nummer *n* des Wortes in *wortkette* an dem *nadel* beginnt. *nadel* kann ein einzelnes Wort oder eine Wortkette sein. Mit *start* kann die Suche bei einem anderen als dem ersten Wort von *wortkette* beginnen.
0 wird immer geliefert, wenn ein oder beide Wortargumente leere Zeichenketten sind oder *start* größer als die Wortzahl in *wortkette* ist.



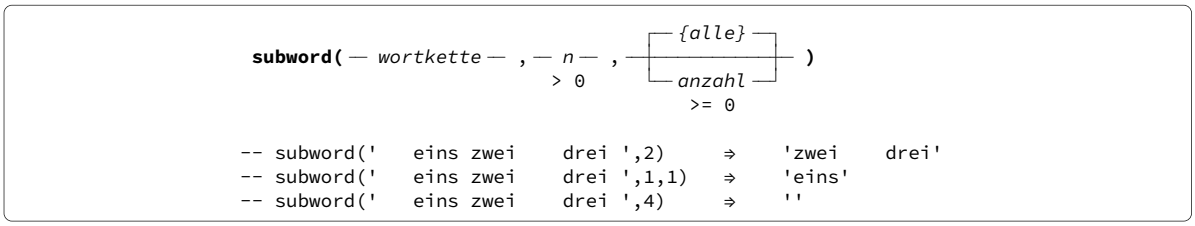
Arbeitet wie **wordpos()**, liefert aber nur 1 oder 0, wie in Vergleichsoperationen erwartet (Seite 56).

2.2 Teile von Wortketten lesen oder löschen



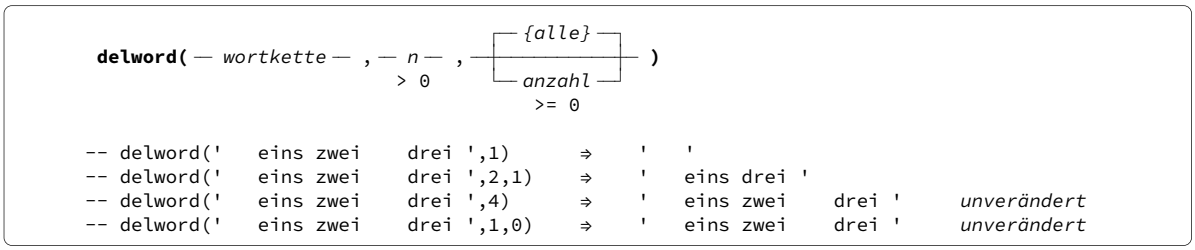

```
-- word('   eins zwei    drei ',2)     ⇒    'zwei'
-- word('   eins zwei    drei ',5)     ⇒    ''
```

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



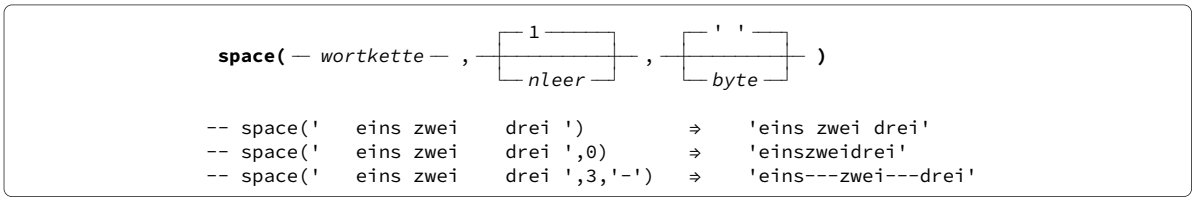

```
-- subword('   eins zwei    drei ',2)     ⇒    'zwei    drei'
-- subword('   eins zwei    drei ',1,1)   ⇒    'eins'
-- subword('   eins zwei    drei ',4)     ⇒    ''
```

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



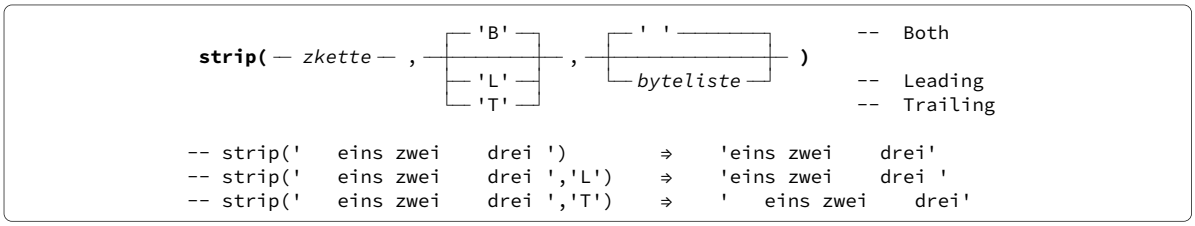

```
-- delword('   eins zwei    drei ',1)      ⇒    '   '
-- delword('   eins zwei    drei ',2,1)    ⇒    '   eins drei '
-- delword('   eins zwei    drei ',4)      ⇒    '   eins zwei    drei '     unverändert
-- delword('   eins zwei    drei ',1,0)    ⇒    '   eins zwei    drei '     unverändert
```

Liefert den Rest von *wortkette*, nachdem daraus *anzahl* Wörter gelöscht sind, beginnend mit dem *n*-ten Wort. Die jedem gelöschten Wort **folgenden Leerstellen** werden ebenfalls gelöscht.




```
-- space('   eins zwei    drei ')         ⇒    'eins zwei drei'
-- space('   eins zwei    drei ',0)       ⇒    'einszweidrei'
-- space('   eins zwei    drei ',3,'-')   ⇒    'eins---zwei---drei'
```

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()**.




```
-- strip('   eins zwei    drei ')        ⇒    'eins zwei    drei'
-- strip('   eins zwei    drei ','L')    ⇒    'eins zwei    drei '
-- strip('   eins zwei    drei ','T')    ⇒    '   eins zwei    drei'
```

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 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.

3.1 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
-- n = 0
-- do n    ⇒    Schleife wird übersprungen
```

Hier steht vor Beginn die Anzahl der Durchläufe fest. Dabei muss *ganzzahl* eine positive ganze Zahl sein. Möglich ist auch eine Variable oder eine Rexx-Anweisung, die eine solche Zahl liefert. Ist die Zahl 0, wird die Schleife ignoriert. Die Verarbeitung geht sofort in der Zeile hinter **end** weiter.

3.2 Schleifen mit Iterationsvariable

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

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


-- 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 ...
```

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 letzte Durchlauf erhalten und kann wie jede andere Variable benutzt werden.
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.

```
-- Mit der Iterationsvariable kann das zugehörige end markiert werden
-- (optional, aber verbessert die Lesbarkeit sehr)

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 nach der Schleife erhalten
```

Der Name der Iterationsvariable kann an das zugehörige **end**-Schlüsselwort angehängt werden. Werden Schleifen ineinander geschachtelt, verbessert das die Übersichtlichkeit sehr. Der Name kann auch an **leave** und **iterate** (siehe S. 15) 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 56 dargestellten Vergleichsoperatoren wie für **if** und **when**. Wie dort, sind die Operatoren `&` (UND), `|` (ODER), `&&` (exklusives ODER) und das Komma (bedingtes UND) möglich. 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
```

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
```

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

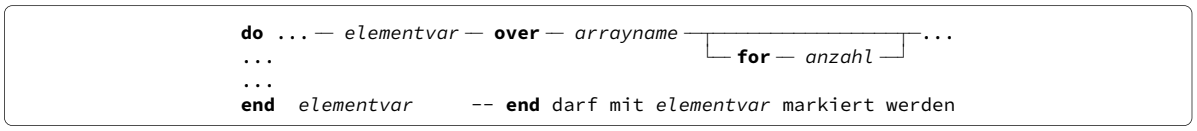
Abfolge der Prüfung auf Schleifenende

```
Vor jedem Schleifendurchlauf:
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.

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

Nach jedem Schleifendurchlauf oder sofort nach iterate :
7. Wenn die until Bedingung erfüllt ist, Schleife verlassen.
8. Iterationsvariable iter entsprechend by  inkr erhöhen.
9. weiter mit 1.
```

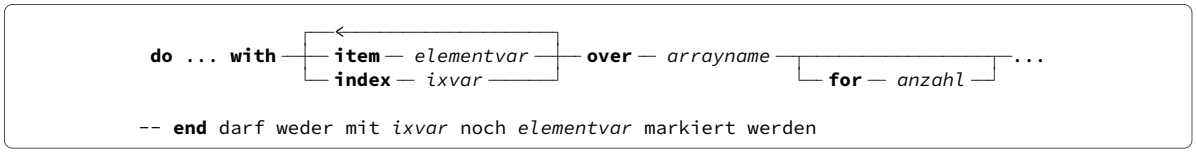
In Schritt 8 wird die Iterationsvariable *iter* am Schleifenende immer hochgezählt, auch wenn die folgenden Schritte 1 bis 3 vor Beginn des nächsten Durchlaufs die Schleife beenden. Daher hat *iter* am Schluss meist den Wert, der einem zusätzlichen Durchlauf entspricht (zum Beispiel 9 bei **i=1 to 8**).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

3.3 Neue Schleifenarten für Datenkollektionen

In ooRexx 5.1 sind 13 Datenkollektionen definiert. Dafür gibt es jetzt 2 Schleifenarten, die hier am Kollektionstyp **Array** erklärt werden, da ich mit den anderen Typen keine Erfahrungen habe. Es **muss nicht bekannt sein, wieviele Elemente** die Kollektion enthält. **Lücken in der Indexfolge** (unbesetzte Elemente) sind zulässig. Sie werden von der Schleife ignoriert.
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 *ixvar* und *elementvar* weiter und haben den Inhalt des letzten Durchlaufs.



Die Schleife wird einmal für jedes in *arrayname* vorhandene, besetzte Element durchlaufen. Dabei erhält Variable *elementvar* bei jedem Durchlauf eine Kopie 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 *elementvar* an das zugehörige **end**-Schlüsselwort angehängt werden, um das Schleifenende zu markieren. Schleifenbeispiele sind auf den Seiten 52, 54 und 46 gezeigt.



Diese Schleifenart **do...with...over** arbeitet *arrayname* auf dieselbe Weise ab wie es **do...over** tut. Das Schlüsselwort **item** kopiert jeweils den Inhalt des aktuellen Elementes in *elementvar*. Mit **index** kann zusätzlich, oder bei Bedarf allein, in *ixvar* die aktuelle Indexnummer geladen werden. Voraussetzung dafür ist, dass die jeweilige Kollektionsart einen Index hat, was für Array zutrifft. Die Reihenfolge der Schlüsselwörter **index** und **item** ist beliebig.
Bei dieser Schleifenart kann ein Name hinter `end` nur bei Benutzung des im nächsten Abschnitt beschriebenen Schlüsselworts **label** verwendet werden.

```
do with item elementvar over arrayname
    say elementvar
end
```

Das Beispiel zeigt, wie **do...with...over** kodiert werden muss, um dieselben Daten wie **do..over** zu liefern.

3.4 Zusätzliche Anweisungen zur Steuerung

Diese gelten für alle Schleifenarten.

```
do ┌<──────────────────────┐ ...   -- Beide müssen direkt hinter do bzw. loop  stehen
   ├ counter — loopzhl ──┤         -- Zähler der angefangenen Schleifendurchläufe 0, 1, 2 ...
   └ label — name ────────┘         -- Zur Identifizierung des end dieser Schleife

-- end darf mit name der label Anweisung identifiziert werden
```

Wenn benutzt, müssen diese Optionen unmittelbar hinter **do** in beliebiger Reihenfolge stehen.
counter bewirkt, dass Variable *loopzhl* mit 0 initialisiert wird. Jeder begonnene Schleifendurchlauf, auch wenn er mit **leave** oder **iterate** abgebrochen wird, erhöht Variable *loopzhl* um genau 1. Das unterscheidet sie von der Iterationsvariable, die beliebige Schrittweiten haben kann und auch innerhalb der laufenden Schleife änderbar ist. Variable *loopzhl* kann vom Programm gelesen werden. Änderungen überschreibt ooRexx 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.

Schleifen innerhalb von Schleifen

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.

4 Rechnen

Auf einem gewöhnlichen Lenovo E15G4 Notebook braucht ooRexx 0.30 Sekunden für die Berechnung einer 1000 m langen Flugbahn mit einem Runge-Kutta-Nyström Verfahren vierter Ordnung. Das ist über 7 mal schneller als die 2.15 Sekunden Flugzeit in der Realität. Für eine interpretierte Sprache rechnet ooRexx 5.1 sehr zügig, vor allem auf aktueller Hardware.
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.

4.1 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 19.
[2] Soweit möglich, speichert Rexx intern beide Darstellungsformen um Konvertierungen einzusparen.

4.2 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

```
-- Platzhalter {#} steht hier für: Ziffernzahl wie in zahl vorhanden bzw. für Exponent notwendig

format( — zahl — , ┌─{#}─────┐ , ┌─{#}─────┐ , ┌─{#}─┐ , ┌─digits()─┐ )
                   └─vorkomma─┘   └─nachkomma─┘   └─exp─┘   └─estart──┘
                      > 0            >= 0          >= 0        >= 0
                                                  0 verhindert
-- format(3.5678,4,2)   ⇒    '   3.57'            Exponentialdarstellung
-- format(-30,4,2)      ⇒    ' -30.00'
-- format(30,,,,0)      ⇒    '3.0E+1'
-- format(30,,,3,0)     ⇒    '3.0E+001'
-- format(30,4,2,3,0)   ⇒    '   3.00E+001'
```

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.

```
trunc( — zahl — , ┌─0─────────┐ )
                  └─nachkomma─┘

-- trunc(3.14159)            ⇒    3
-- trunc(3.14159,4)          ⇒    3.1415
-- trunc(3.14159,7)          ⇒    3.1415900
-- trunc(12345678987.123,2)  ⇒    12345679000.00
```

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.

4.3 Mitgelieferte Bibliothek RXMATH

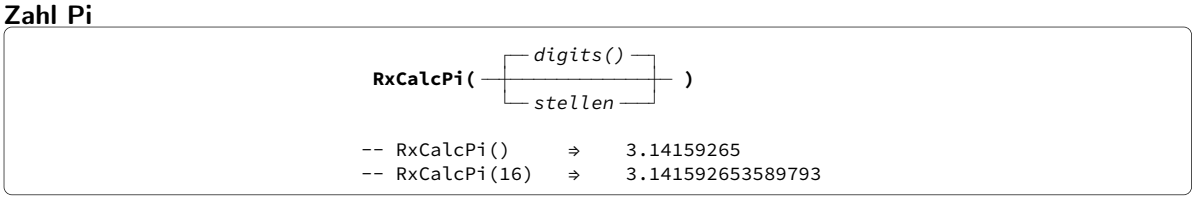
Mit Rexx wird eine mathematische Bibliothek auf der Basis von C-Bibliotheksfunktionen installiert (Datei `rxmath.dll`). Um sie in einem Rexx-Programm zu nutzen, ist **hinter** der letzten Programmzeile folgende Direktive einzufügen:

```
::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.

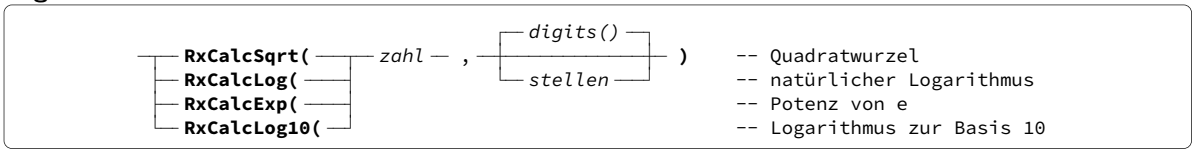
[3] *What's Wrong with Rexx ?* Vortrag, IBM Sindelfingen 2004

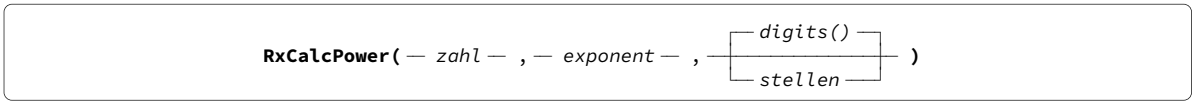
Zahl Pi



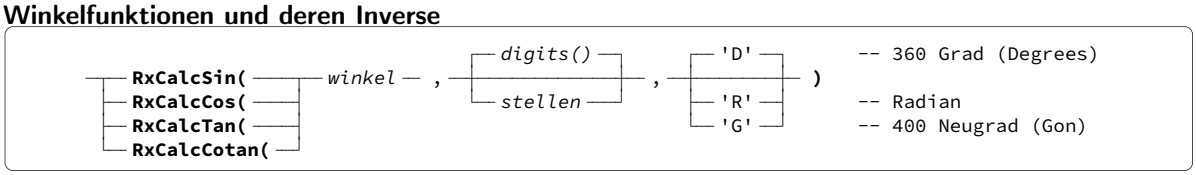
Für diese und alle anderen RXMATH-Funktionen gilt: Die Rechengenauigkeit des aufrufenden Programms, wie von **digits()** geliefert, gilt auch innerhalb der Funktion. Der gewünschte Wert kann auch als Zahl *stellen* oder als Variable übergeben werden, die eine ganze Zahl zwischen 1 und 16 sein muss. Größere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

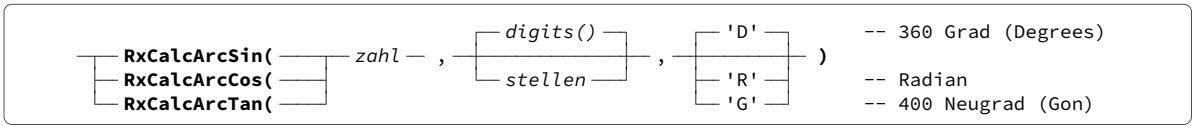
Logarithmen und Potenzen



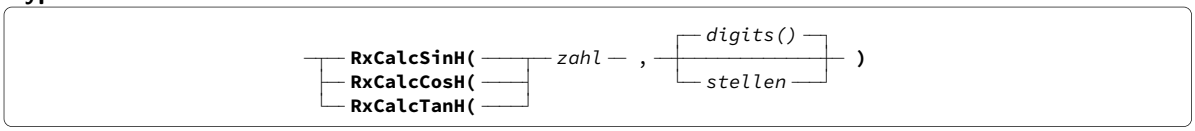


Winkelfunktionen und deren Inverse





Hyperbelfunktionen



4.4 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 16 erwähnte Flugbahnprogramm benötigt mit RXM bei 16 Stellen Genauigkeit 1.22 Sekunden statt 0.30. Bei 32 Stellen Genauigkeit sind es 2.69 Sekunden. RXM rechnet intern jeweils mit 10 Stellen (bei Logarithmen 100) mehr als angefordert.
Um RXM benutzen zu können, ist **hinter** der letzten Programmzeile einzufügen:

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

Seit Februar 2020 kann die Erweiterung wegfallen, da 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 eine Codezeile:

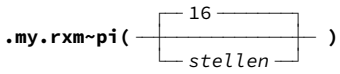
```
.local~my.rxm = .rxm~new(16,"D")  -- ursprüngliche Prologzeile
.local~my.rxm = .rxm~new(50,"R")  -- Achtung: geänderte Prologzeile 2024-12-23
```

Sie erzeugt ein neues Objekt mit dem Namen `.my.rxm`, das die Attribute und Methoden der Klasse RXM enthält. Zugleich wird die Genauigkeit auf 50 Stellen und das Winkelmaß auf 2π (Radian) voreingestellt.[4] 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
.my.rxm~type='D'
```

Diese Aufrufe ändern beispielsweise die Voreinstellung der Genauigkeit auf 32 Stellen und das Winkelmaß auf **D**egrees (360°). RXM vermag leider die Genauigkeit des aufrufenden Rexx-Programmes nicht automatisch zu übernehmen.

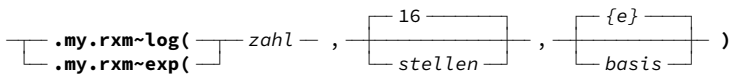
Methoden in RXM



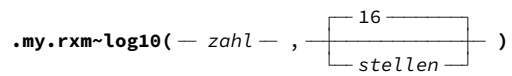

```
-- .my.rxm~pi      ⇒    3.141592653589793
-- .my.rxm~pi(64)  ⇒    3.14159265358979323846264338327950288419716939937510582097494459 2
```

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

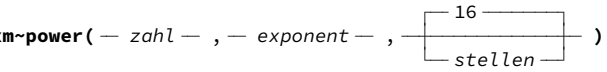
Logarithmen und Potenzen

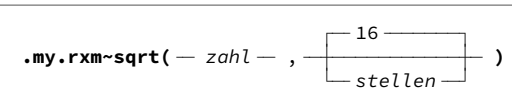



```
-- .my.rxm~log(2)       ⇒    0.6931471805599453      log Basis e
-- .my.rxm~log(2,,10)   ⇒    0.3010299956639812      log Basis 10
-- .my.rxm~log(2,,2)    ⇒    1                       log Basis 2
-- .my.rxm~exp(-0.5)    ⇒    0.6065306597126334      e hoch -0.5
```

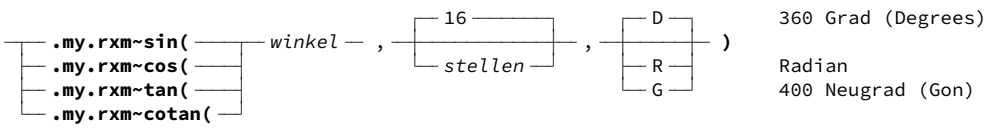



```
-- benutzt die Funktion log
```

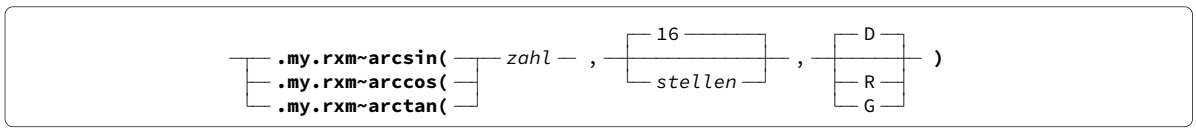




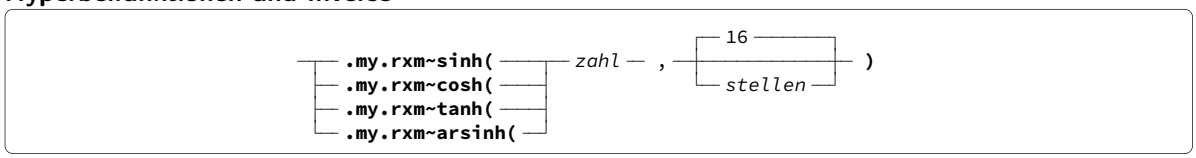
Winkelfunktionen und deren Inverse



[4] **Achtung:** Vor dem 23.12.2024 heruntergeladene `rxm.cls` haben 16 Stellen und den **360 Grad Kreis (D)** voreingestellt. Diese Defaults sind hier in den Diagrammen gezeigt.



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 19 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-Installationsverzeichnis kopiert werden. Dazu sind Administratorrechte erforderlich. Danach können alle Programme diese Bibliothek verwenden.

Zu Rosettacode
Link `rosettacode.org/wiki/Category:OoRexx` zeigt ein Verzeichnis der auf Rosettacode frei verfügbaren, zur Zeit über 230 ooRexx Programme.

5 Uhrzeit und Datum

In Rexx ist für jeden Tag seit Montag, den 01. Januar 0001 dessen laufende Nummer (Format *Basedate*) verfügbar, so das **Fristen** zwischen Daten einfach berechenbar sind. Es wird der Gregorianische Kalender verwendet, so als ob er damals schon gegolten hätte. Allerdings werden die seit 1972 eingeschobenen Schaltsekunden (zuletzt 2017) ignoriert. In Windows XP, 7, 8.1 und 10 beträgt die Auflösung der Zeitangaben eine tausendstel Sekunde (Nachkommastelle 3). Stellen rechts davon sind also immer 0.

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

          ┌─'N'─┐          ⇒  09:46:37
time( ────┼─────┼──── )
          ├─'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'─┤          ⇒  63692300797000000  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.
Ein in derselben Programmdatei stehendes Unterprogramm erbt den Stopuhrstand des Aufrufers. Unabhängig von Resets im Unterprogramm läuft die Stopuhr des Aufrufers weiter.

Uhrzeitformat umwandeln

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

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

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

Liefert die Uhrzeit *inzeit* im gewünschten Ausgabeformat.

Datum

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

                ┌─ 'N' ─┐      ⇒    1 May 2019
 date( ─────────┼───────┼──────────────────────────────── )
                │       └─ , ── , ── , ─ trenn ─┐      Trennzeichen
                ├─ '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' ─┤      ⇒    636922656000000000  Mikrosekunden seit 01.01.0001 00:00
                └─ 'T' ─┘      ⇒    1556668800          Sekunden      seit 01.01.1970 00:00

-- Beispiel für die Änderung des Trennzeichens:
--      date('E',,,'.')   ⇒    01.05.19
```

Für Rechnungen mit den Zeitformaten **Full** (*Mikro*sekunden) und **Ticks** (Sekunden) muss die Einstellung NUMERIC DIGITS auf mindestens 18 bzw. 12 erhöht werden. Siehe dazu Seite 16.

Datumsformat umwandeln

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

                ┌─ 'N' ─┐                ┌─ 'N' ─┐                ┌─ {default} ─┐
 date( ─────────┼───────┼─ , ─ indatum ─ ,┼───────┼─ , ─────────── ┼─────────────┼ )
                ├─ 'I' ─┤                ├─ 'I' ─┤      └─ trenn ─┘└─ intrenn ──┘
                ├─ 'S' ─┤                ├─ 'S' ─┤      in Ausgabe  indatum Trennzeichen
                ├─ 'E' ─┤                ├─ 'E' ─┤                  wenn \= Default für
                ├─ 'O' ─┤                ├─ 'O' ─┤                    indatum Format
                ├─ 'U' ─┤                ├─ 'U' ─┤
                ├─ 'M' ─┤                ├─ 'B' ─┤
                ├─ 'W' ─┤                ├─ 'F' ─┤
                ├─ 'B' ─┤                └─ 'T' ─┘
                ├─ 'D' ─┤              Format von indatum
                ├─ 'F' ─┤
                └─ 'T' ─┘
              Ausgabeformat

--  Umwandlung Basistag in Kalenderdatum:
--  date('I',737179,'B')            ⇒    2019-05-01
--  date('I',737179,'B','.')        ⇒    2019.05.01   -- Änderung Trennzeichen

--  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.

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

6 Dateien und Verzeichnisse verwalten

Als Erbe aus der Lochstreifenzeit sind (Unix- und) Windows-Dateien ein linearer Zeichenstrom ohne Zeilenstruktur. Die Zeilen werden durch das Zeilenende-Zeichenpaar hexa 0D0A (vormals Wagenrücklauf und Zeilenvorschub des Fernschreibers, kurz CRLF) gebildet. Zum Zählen der Zeilen muss das Betriebssystem die ganze Datei lesen. Dieser Vorgang ist mit aktuellen Windows-Versionen und moderner Hardware sehr viel schneller geworden.
Bei der zeilenweisen Verarbeitung (**L**ines) erledigt ooRexx „hinter den Kulissen“ das Anhängen von CRLF beim Schreiben in Dateien, sowie das Löschen beim Lesen daraus. Im Rexx-Programm sind die CRLF nicht sichtbar. Bei blockweiser Verarbeitung (**C**haracter) dagegen sind CR und LF gewöhnliche Zeichen und im Programm sichtbar.

6.1 Dateien verwalten

Existenz prüfen, löschen

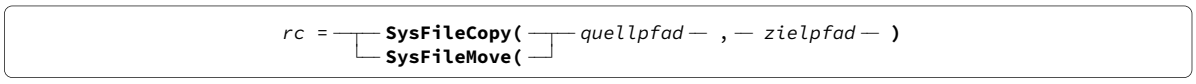
```
flag = SysIsFile( — dateiname — )

 0  existiert nicht
 1  existiert

-- Alternative meldet auch dann 1 wenn dateiname ein Verzeichnis ist:

flag = SysFileExists( — dateiname — )
```

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 ?.

```
rc = SysFileDelete( — dateiname — )

 0  gelöscht
 2  nicht gefunden
```

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 59).

Kopieren, Umsetzen, Umbenennen



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.

6.2 Verzeichnisse verwalten

Wenn kein Pfad angegeben ist, 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.

```
qualify( — nichtleer — )

-- das aktuelle Verzeichnis sei:          'D:\Sandbox'
-- qualify('abstract.txt')          ⇒    'D:\Sandbox\abstract.txt'
-- qualify('.\rexx\abstract.txt')   ⇒    'D:Sandbox\rexx\abstract.txt'
-- qualify('\rexx\abstract.txt')    ⇒    'D:\rexx\abstract.txt'
-- qualify(' ')                     ⇒    ''
```

Liefert den Pfad des aktuellen Verzeichnisses, dem wie ein Dateiname die Zeichenkette *beliebig* angehängt ist. Sie darf weder eine Leerstelle noch eine leere Zeichenkette sein.

```
flag = SysIsFileDirectory( ┬─ name ──────── )   -- äquivalent zu .\name
                           ├─ .\name ─────┤     -- Unterverzeichnis des aktuellen Verz.
 0  existiert nicht        ├─ ..\name ────┤     -- gleiche Ebene wie aktuelles Verz.
 1  existiert              ├─ ..\..\name ─┤     -- Ebene höher als aktuelles Verz.
                           └─ x:\pfad ────┘     -- kompletter Pfad
```

Liefert 1, falls das Verzeichnis existiert, sonst 0.

```
rc = ┬─ SysMkDir( ─┬─ pfadname — )
     └─ SysRmDir( ─┘
```

Entspricht den Windows-Befehlen `md` zum Anlegen (Make) und `rd` zum Löschen (Remove) von Verzeichnissen. Liefert 0 bei Erfolg, sonst einen Windows Returncode, wie auf Seite 59 beschrieben.

Elemente des Pfadnamens lesen

```
                                                -- Beispiel:  D:\Sandkasten\kurz.txt
filespec( ┬─ 'D' ─┬─ , — dateipfad — )     ⇒    'D:'
          ├─ 'L' ─┤                        ⇒    'D:\Sandkasten\'
          ├─ 'P' ─┤                        ⇒    '\Sandkasten\'
          ├─ 'N' ─┤                        ⇒    'kurz.txt'
          └─ 'E' ─┘                        ⇒    'txt'
```

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.

```
SysGetShortPathName( — pfadlangname — )

-- sysgetshortpathname('C:\Program Files (x86)')   ⇒    'C:\PROGRA~2'
-- sysgetshortpathname('C:\gibts nicht')           ⇒    ''
-- sysgetshortpathname('C:\Users')                 ⇒    'C:\Users'
-- sysgetshortpathname('C:\Benutzer')              ⇒    ''          nur Explorer kennt Deutsch

-- umgekehrter Weg:
SysGetLongPathName( — pfadkurzname — )
```

Tipp

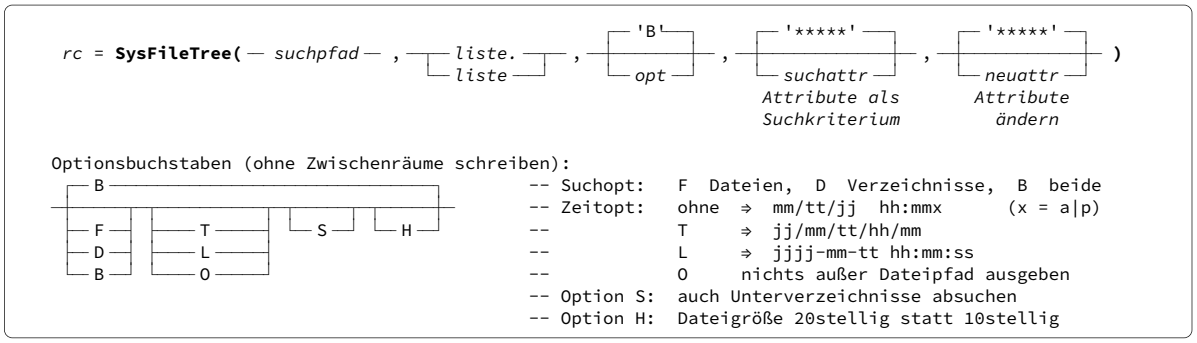
Zur schnellen Suche nach einem -auch nur teilweise bekannten- Dateinamen ist der eigentlich zur Attributänderung benutzte Windows-Befehl **ATTRIB** gut geeignet. Zum Beispiel:

```
ATTRIB *1916.pdf /S
```

sucht im aktuellen Verzeichnis und allen seinen Unterverzeichnissen (`/S`) nach allen PDF-Dateien deren Name mit „1916“ endet und gibt die vollständigen Pfadnamen am Bildschirm aus.
Auf Seite 36 ist beschrieben wie ein ooRexx-Programm solche Bildschirmausgaben weiter verarbeiten kann.

Seitdem *Solid State Disks (SSD)* die mechanischen Festplatten ersetzen, ist ATTRIB zu einem nochmals sehr viel schnelleren Werkzeug geworden.

6.3 In Verzeichnissen nach Datei-/Verzeichnisnamen suchen



suchpfad gibt das zu durchsuchende Verzeichnis und den Dateinamen an. Fehlt das Verzeichnis, wird im aktuellen Verzeichnis gesucht. Es kann wie im Windows üblich maskiert werden, zum Beispiel: `*.exe`
Die Suche kann auf Dateien mit bestimmten Attributen eingeschränkt werden (Archive, Directory, Hidden, Read-Only, System, kurz ADHRS). Diese sind als eine 5 Bytes lange Zeichenkette *suchattr* anzugeben, wobei * für beliebig, **+** für gesetzt und **-** für nicht gesetzt (gelöscht) steht.
Die Ausgabe der Trefferzeilen erfolgt in *liste.*. Endet dieser Name –wie hier– mit einem Punkt, wird das Ergebnis als Stammvariable[1] zurückgeliefert. Diese braucht nicht vorher zu existieren. Falls doch, wird sie überschrieben. Element *liste.0* enthält dann die Anzahl der gefundenen Treffer. Ist diese Zahl größer als 0, enthalten *liste.1 ...* die Daten zu jedem Treffer.
Falls der Ausgabename **nicht** mit einem Punkt endet (*liste*) **und** bereits ein Array dieses Namens existiert, wird dieses mit dem Ergebnis befüllt. Die Elemente *liste[1]...* enthalten die gefundenen Treffer. Existiert das Array nicht, erfolgt die Ausgabe als Stammvariable (*liste.*), wie es vor ooRexx 5.0 der Fall war.
Die Optionsbuchstaben für die Ausgabe des Resultats sind ohne Zwischenräume anzugeben, zum Beispiel 'FLS'.

```
Auswirkung der Zeitoption (ohne|T|L|O) auf die Feldpositionen 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

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

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!
....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8

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

Bei Option H stehen Attribute und Pfadnamen wegen der zusätzlichen Breite des Größenfeldes um 10 Positionen weiter rechts.
Mit der 5 Byte langen Zeichenkette *neuattr* lassen sich Dateiattribute ändern, wobei * für unverändert lassen, **+** für Setzen und **-** für Löschen steht. Die von **sysfiletree** zurückgelieferte Dateiliste enthält schon die geänderten Attribute.

[1] Stammvariable sind ab Seite 35 beschrieben, Arrays ab Seite 40.

7 Dateien lesen und schreiben (neu)

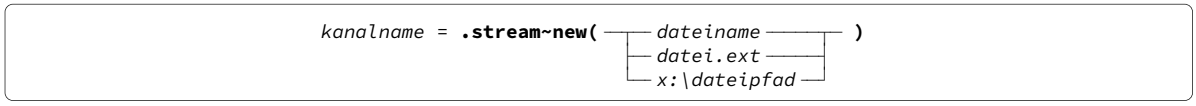
Zum Lesen der 3 Millionen Zeilen einer 120 Megabytes große Datei benötigt mein Notebook Lenovo E595 mit 1TB mechanischer Festplatte bei Methode **~arrayin** nur noch 7.4 Sekunden, mit der von *Jon Wolfers* vorgeschlagenen Anweisung nur 1.1 Sekunden! In der heutigen Zeit großer Hauptspeicher (RAM) kann man Dateien *in einem Rutsch* vollständig in RAM kopieren um sie dort zu bearbeiten. Das ist viel schneller als das herkömmliche stückweise Lesen von oder Schreiben auf Speichermedium.

- Hierbei wird die Objektart **Array** als Speicher des Dateiinhalts benutzt. Wie man mit Arrays arbeitet, ist ab Seite 40 beschrieben.
- Erforderlich ist außerdem die explizite Definition eines **Stream**-Objekts als Datenkanal zur Datei auf dem Speichermedium. Daher wird Stream hier mit **Kanal** übersetzt.

Die herkömmliche Methode wird im nächsten Kapitel ab Seite 30 beschrieben.

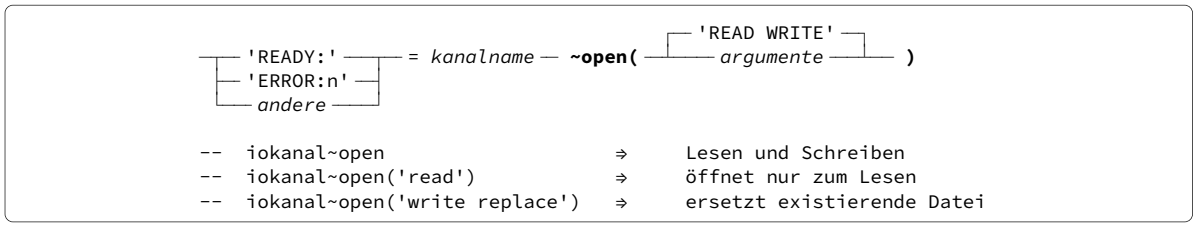
7.1 Lesen: Datei in ein Array kopieren

Schritt 1: Dateikanal zum Speichermedium definieren



Legt einen Kanal *kanalname* an, durch den auf die Datei zugegriffen werden kann. Es erfolgt **keine** Prüfung, ob die Datei oder das Laufwerk im Zugriff sind. Das geschieht beim nachfolgenden Öffnen.

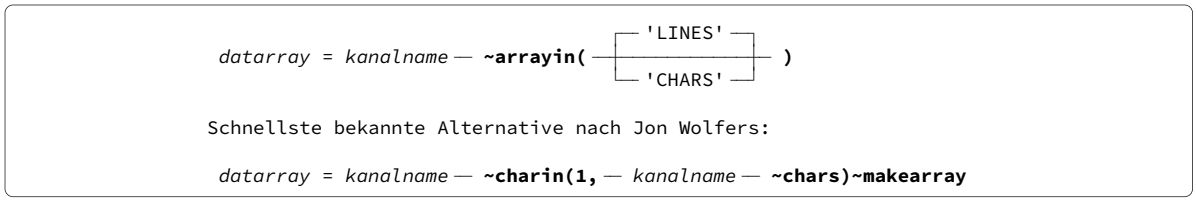
Schritt 2: Dateikanal öffnen



Wird Zeichenkette `READY:` geliefert, ist die Datei benutzbar. Bei einem Fehler wird zum Beispiel `ERROR:2` geliefert. Die Zahl –hier 2– ist ein Windows-Returncode, wie auf Seite 59 beschrieben. Andere Rückmeldungen wie NOTREADY, UNKNOWN sind dokumentiert, konnten bei meinen Tests aber nicht reproduziert werden.

Schritt 3: Array befüllen

Mit einer Operation wird der Inhalt der Datei in ein Array-Objekt kopiert. Damit stehen die Daten im Hauptspeicher (RAM) und können sehr schnell bearbeitet werden.



Falls Array *datarray* nicht existiert, legt Methode **~arrayin** es an. Falls das Array existiert, werden vor dem Kopieren alle darin vorhandenen Daten gelöscht. Im Lines-Modus enthält jedes Element des Arrays eine Dateizeile ohne CRLF am Zeilenende.
Die Alternative von Jon Wolfers benötigt nur 1.1 statt der oben erwähnten 7.4 Sekunden.

Schritt 4: Dateikanal schließen

```
kanalname — ~close
```

Es ist guter Programmierstil, nach **~arrayin** den ohnehin nicht mehr benötigten Kanal zu schließen. Ansonsten bleibt jeder Kanal bis zum Programmende geöffnet und belegt Ressourcen. Das gilt insbesondere, wenn viele Dateien abgearbeitet werden müssen.

Bei Bedarf: Informationen über die Datei; neu Positionieren

```
num = kanalname — ~lines          ⇒   Anzahl der noch ungelesenen Zeilen

flag = kanalname — ~lines('N')    ⇒   1: ungelesene Zeilen vorhanden, sonst 0

num = kanalname — ~chars          ⇒   Anzahl der noch ungelesenen Bytes
```

Diese Methoden sind selbsterklärend.

```
neupos = kanalname ─┬─ ~seek( ──────── 'zkette' — )
                    └─ ~position( ─┘

--  'zkette'  besteht aus:              -- ist immer in '..' oder "..." einzuschließen

                                        -- =1 steht für Dateianfang und <0 für Dateiende
                                        -- = ab Dateianfang, =0 wird wie =1 behandelt
   1.                 ┌ = ┐
                  ────┼ < ┼──── num ──
                      │ + │
                      └ - ┘            -- < ab Dateiende Richtung Anfang
                                        -- + ab der aktuellen Position Richtung Ende
                                        -- - ab der aktuellen Position Richtung Anfang
   2. (optional)  ──┬──────┬──         -- Nur benutzen wenn unterschiedliche Lese-
                    │ READ │           -- und Schreibpositionen notwendig sind
                    └ WRITE┘

                      ┌ C ┐
   3. (optional)  ────┼   ┼────       -- CHAR: num Bytes (unerwarteter Default)
                      └ L ┘            -- LINE: num Zeilen

--                                         aktuelle Position ist danach:
-- kanalname~seek('64')                 ⇒  Byte 64
-- kanalname~seek('=8 L')               ⇒  Zeile 8
-- kanalname~seek('< 0 L')              ⇒  Anfang der letzten Zeile
```

Falls nicht ab Dateianfang gelesen werden soll, kann mit **seek** die gewünschte Position eingestellt werden.
Wie man mit den ins Array kopierten Daten arbeiten kann, ist ausführlich im Kapitel über Arrays ab Seite 40 beschrieben.

7.2 Schreiben: Array in eine Datei kopieren

Im Normalfall wird jedes Arrayelement zu einer Zeile der geschriebenen Datei.
Da hier dieselbe Abfolge von vier Schritten wie beim Lesen gilt, genügt die Zusammenfassung in einem einzigen Syntaxdiagramm:

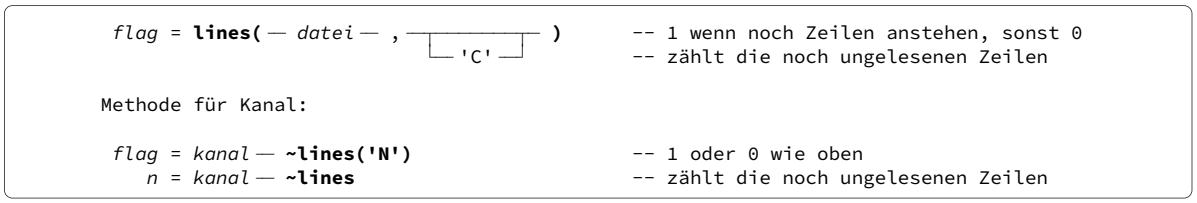
```
kanalname = .stream~new( — ausdatei — )     -- 1. Kanal zur Datei erzeugen

kanalname — ~open('write replace')          -- 2. Öffnen zum Schreiben und Ersetzen

kanalname — ~arrayout( — datarray — )       -- 3. Kopiervorgang

kanalname — ~close                          -- 4. Kanal/Datei schließen


Schnellere Alternative zu Schritt 3:

kanalname ~charOut(datarray~tostring,1)     -- 3a. Vorschlag Jon Wolfers
kanalname ~charout('0D0A'x)                 -- 3b. sonst fehlt der letzten Zeile das CRLF
```

In Schritt 3 werden alle Array-Elemente in einer einzigen Operation auf das Speichermedium kopiert und erhalten dabei jeweils ein CRLF-Paar angehängt. **Alternativ** können stattdessen die Schritte 3a und 3b angewendet werden.
Beim Schließen der Datei in Schritt 4 liefert die Methode **close** die Zeichenkette `READY:` zurück. Im Fehlerfall wird stattdessen ein Windows-Returncode geliefert; siehe Seite 59.
Um eine 3 Millionen Zeilen große Datei von 77 Megabytes zu speichern, benötigte bei einem Test die Methode **~arrayout** 9.4 Sekunden, der Vorschlag 3a von *Jon Wolfers* 2.4 Sekunden. Allerdings ist die dabei erzeugte Datei 2 Bytes kleiner, weil der letzten Zeile das CRLF-Paar fehlt. Das kann jedoch mit der zusätzlichen Anweisung 3b problemlos behoben werden.

8 Dateien zeilen-/blockweise lesen und schreiben (herkömmlich)

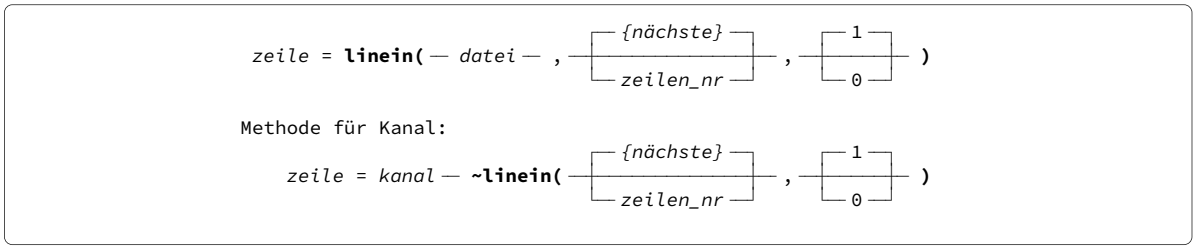
Pro Aufruf wird genau eine Zeile gelesen/geschrieben, oder ein „Block“ aus einem oder mehreren Bytes.[1] Diese Verarbeitung ist bei großen Dateien spürbar langsamer.
In zahllosen existierenden Rexx-Programmen trifft man auf diese Funktionen. Sie sind einfacher zu kodieren, weil die Kanaldefinition wegfällt und **kein** explizites Öffnen der Datei notwendig ist. Dies erfolgt beim ersten Zugriff automatisch.
Auch Dateien, auf die über einen **Kanal** zugegriffen wird, lassen sich zeilen- und blockweise verarbeiten. Die entsprechenden Methoden sind in den folgenden Diagrammen ebenfalls beschrieben.

Zeilenzahl feststellen



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.

Datei zeilenweise lesen



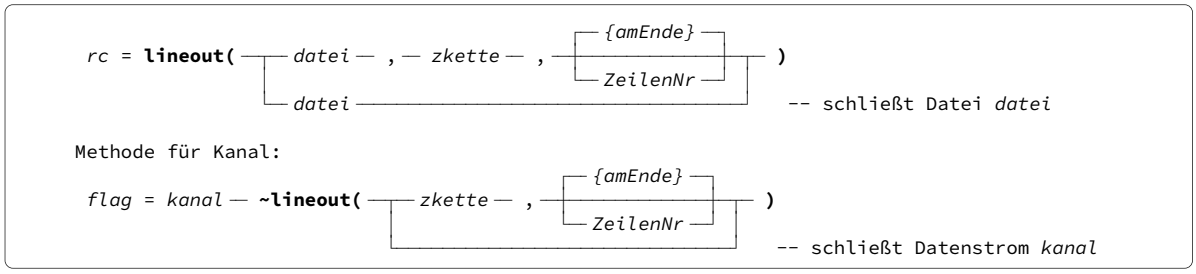
Liefert, beginnend mit Zeile 1 von *datei*, die jeweils nächste Zeile. Nur je 1 Zeile pro Aufruf ist möglich. Konzeptionell sind die Zeilen fortlaufend ab 1 nummeriert. Es kann direkt Zeile *zeilen-nr* gelesen werden. Die nächste *zeilen-nr* darf dann auch Richtung Dateianfang liegen.
Ist das dritte 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   -- jetzt empfohlen:   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.

[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.

Datei zeilenweise schreiben

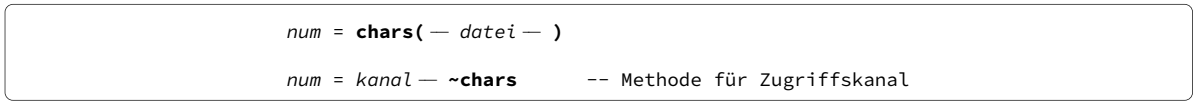


Hängt *zkette* als neue Zeile an das Ende von *datei* an. Wird *ZeilenNr* angegeben, überschreibt **lineout()** ab dem Anfang dieser Zeile die existierenden Daten, ohne Rücksicht ob alte und neue Länge übereinstimmen. *ZeilenNr* muss innerhalb der existierenden Daten liegen oder die erste neue Zeile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

8.1 Daten blockweise/zeichenweise lesen oder schreiben

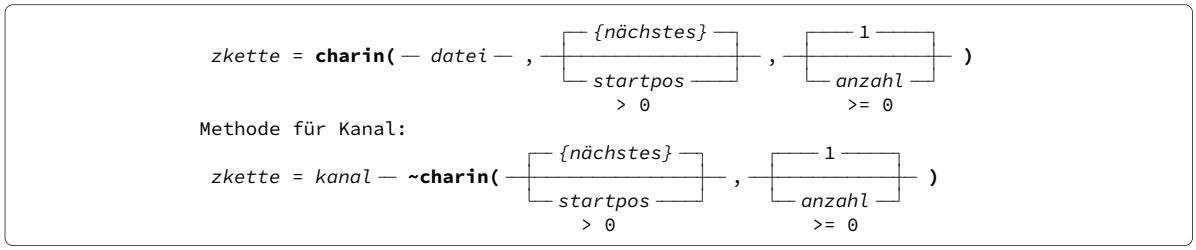
Hier hat das Zeilenende-Zeichenpaar (hexa 0D0A) keine Steuerfunktion und wird wie alle anderen Bytes auch behandelt. Ein Block kann minimal 1 Zeichen lang sein. Zeilenweise Struktur der Daten ist hier auch möglich, sofern die Daten *Zeilen identischer Länge* bilden, zum Beispiel eine Tabelle mit 244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

Anzahl vorhandener Zeichen feststellen



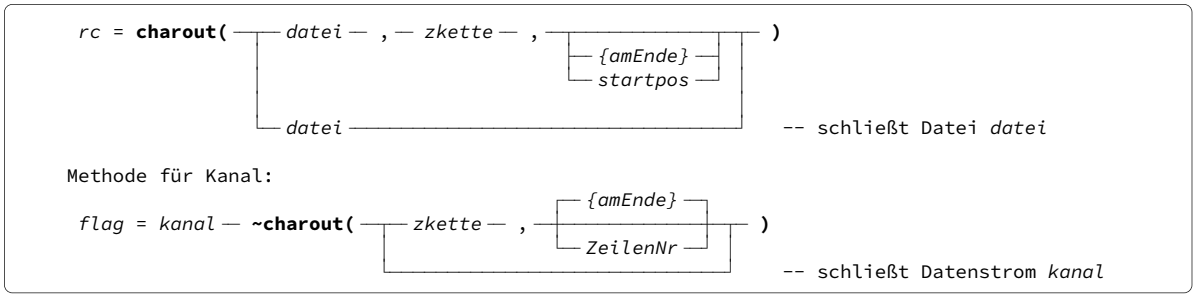
Liefert die *anzahl* der verfügbaren –noch nicht gelesenen– Zeichen in *datei* oder 0.

Blockweisen/zeichenweise lesen



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.

Blockweise/zeichenweise schreiben



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

Datei schließen

Alle vom Programm geöffneten Dateien werden bei Programmende automatisch geschlossen. Explizit kann jede Datei mit folgendem Aufruf geschlossen werden:

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

Es wird die Zeichenkette `READY:` geliefert oder die leere Zeichenkette, falls *datei* nicht offen war. Im Fehlerfall enthält *rc* den Windows Returncode; siehe Seite 59. Zum Schreiben geöffnete Dateien können auch mit **lineout()** und **charout()** geschlossen werden, wie oben beschrieben.

9 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.[2] Außerhalb der Zeichen `'20'x ... '7E'x` (Ziffern, Buchstaben und Interpunktion) muss man mit Unterschieden rechnen.
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.

[2] Im deutschen Windows 10 benutzt das Kommandozeilenfenster die Codepage 850, der Editor Notepad dagegen 1252. Der Hessling Editor änderte mit Version 4.0 seine Codepage von 850 in 1252. Hexa D6 wird in 850 als Í dargestellt, aber als Ö in 1252.

Umwandlungsbeispiele

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

Vorzeichenlose ganze Zahlen

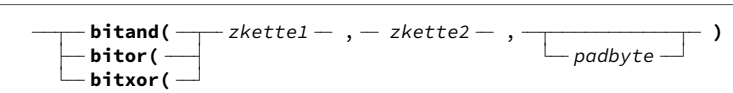
```
c2d('Z')      ⇒   90
x2d('FFFF')   ⇒   65535
d2x(123456)   ⇒   1E240     -- beachte Ausgabe hex 1 für hex 01
```

Ganze Zahlen mit Vorzeichen

```
x2d('FFFF',4)   ⇒   -1         --  4stellig = FFFF
x2d('FFFF',8)   ⇒   65535      --  8stellig aufgefüllt auf 0000FFFF
x2d('00FF',2)   ⇒   -1         --  da von rechts nach links gezählt: hex FF
x2d('FFFF',0)   ⇒   0
d2x(-1,2)       ⇒   FF
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')    ⇒   '0100 0001'b  =  '41'x  =  A
  bitor('e','Y')     ⇒   '0111 1101'b  =  '7D'x  =  }
  bitxor('e','Y')    ⇒   '0011 1100'b  =  '3C'x  =  <
```

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.

10 Multiwerkzeug Stammvariable

Die einfachste, schon im klassischen Rexx existierende Datenkollektion ist die Stammvariable (engl. *stem variable*). Zahllose existierende Programme arbeiten damit. Die mehrdimensionale Variante heißt zusammengesetzte Variable (engl. *compound variable*).
Der Name einer Stammvariable besteht aus mindestens zwei Teilen (Stamm und Index), 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 Index (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 47) hat ihre Bedeutung noch stark zugenommen, weil seitdem Haupt- und Unterprogramme gemeinsam große Datenstrukturen nutzen können, *ohne* interne Kopiervorgänge zu erfordern.

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

Die Schreibweise des Index mit [] erleichtert insbesondere innerhalb von Schleifen den Zugriff auf benachbarte Elemente. Damit kann zudem als Index wiederum eine Stammvariable benutzt werden, was ohne die [] nicht funktioniert.

```
index.999  =  5
say hanse.[index.999]    ⇒     Rostock
```

10.1 Durchgezählte Stammvariable

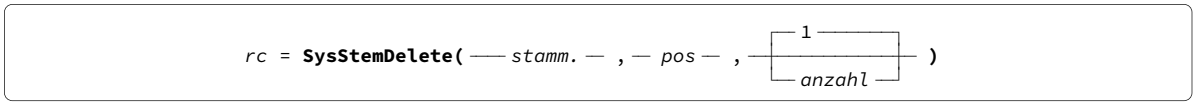
Dies ist die klassische Form für Datenübergabe und Rücklieferung von Daten. *Durchgezählt* erweitert die eben beschriebene Grundform um folgende Bedingungen:

- 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`.

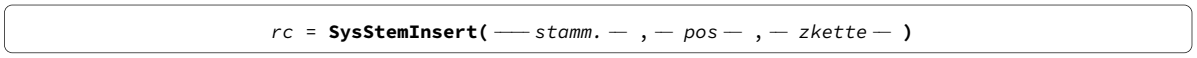
Mit natürlichen Zahlen indexierte Stammvariable, welche diese drei Bedingungen erfüllen, heißen in dieser Kurzreferenz *durchgezählte Stammvariable*. Rexx ist intern **nicht** auf diese Eigenschaften angewiesen. Sie gelten jedoch für die klassische Schnittstelle, unter anderem zu Funktionen wie **SysFileTree** (siehe Seite 26) und den vier nachfolgend beschriebenen.

Funktionen für durchgezählte Stammvariable

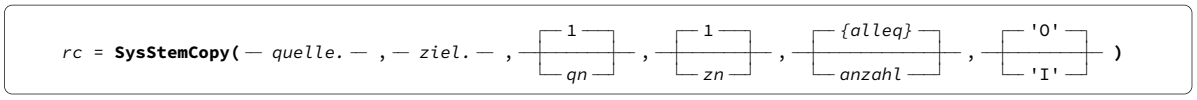
In allen Funktionsaufrufen kann *stamm.* auch ohne Punkt angegeben werden. Die Funktionen ändern Element *stamm.0* je nach Zugang/Abgang von Elementen.



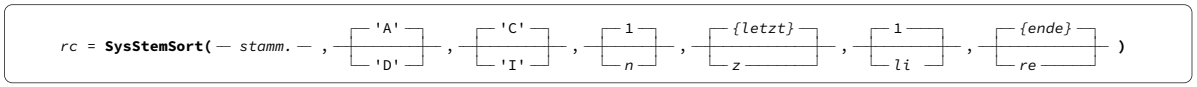
Löscht aus Stammvariable *stamm.* genau *anzahl* Elemente, beginnend mit Nummer *n*. Nachfolgende Elemente besetzen die dadurch freigewordenen Plätze. Wird Element 3 gelöscht, erhält das bisherige Element 4 die Nummer 3 und Element *stamm.0* wird um 1 verringert.



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.
Besonderheit im **Insert**-Modus, sofern *ziel.* existiert: Falls keine anderen Argumente angegeben werden, stehen im Ergebnis zuerst die Elemente aus *quelle.* und anschließend die aus *ziel.*.

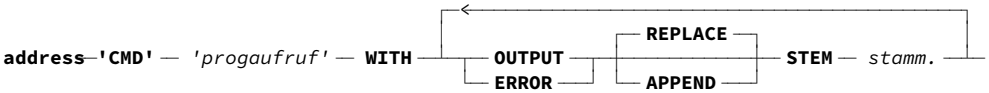


Sortiert die Elemente in *stamm.* nach deren Bytes in den Positionen *li* bis *re*. Voreingestellt ist die aufsteigende Sortierung (A) mit Berücksichtigung von Groß- und Kleinschreibung (C für Case). Argument D (Descending) bewirkt *absteigende* Sortierung. Argument I führt zum Ignorieren der Groß- und 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 43 beschreibt eine Alternative.

Bildschirmausgaben in Stammvariable umleiten

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.
Von einem Fremdprogramm per STDOUT und STDERR erzeugte Ausgaben am Bildschirm können seit ooRexx 5.0 mit der Erweiterung **WITH** des Schlüsselworts **ADDRESS** umgeleitet werden. Sie stehen dann dem aufrufenden Rexx-Programm als durchgezählte Stammvariable zur Verfügung. Diese Methode funktioniert zum Beispiel bei EXE-, BAT-, CMD- und auch REX-Programmen.

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




```
-- Beispiel: Umleitung der Bildschirmzeilen von PDFTK.EXE in die Stammvariable 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 *progaufruf* an Windows. Enthält *progaufruf* Leerstellen, muss er 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.
Wird 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.

Umgang mit Leerstellen in Pfadnamen

Manche älteren Programme kommen mit Leerstellen in Pfadnamen nicht zurecht. Jedes Verzeichnis und jede Datei hat aber auch einen Windows-internen Kurznamen ohne Leerstellen. Auf Seite 25 ist beschrieben, wie man ihn feststellen kann.

10.2 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:

```
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 55) 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 HB ... 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 mittels einer Zweisung wie `land. = xyz` *nicht* möglich.

11 Multiwerkzeug Array

Im ooRexx 5.1 sind 13 Klassen von Datenkollektionen vorhanden, von denen **.array** vor allem wegen seiner Fähigkeit zum schnellen Lesen/Schreiben von Dateien herausragt. Das war für mich der hauptsächliche Grund, die Array-Klasse in diese Kurzreferenz „für Klassiker“ aufzunehmen. Nützlich ist auch, das gezielt nach besetzten und unbesetzten Elementen gesucht werden kann.
Dazu kommt etwas bessere Performance, die ich bei der Verarbeitung von 3 Millionen Zeilen mit je einem X,Y-Koordinatenpaar beobachten konnte. Testweise Umstellung eines Programms von der Datenhaltung in zwei Stammvariablen auf zwei Arrays reduzierte die Laufzeit (ohne Plattenzugriffe) von 55.5 auf 38.5 Sekunden, also um 30 Prozent.

Gemeinsamkeiten und Unterschiede zwischen Array und Stammvariable

Gemeinsam ist Array und Stammvariable, dass jeder Eintrag aus der Adresse (**Index**) und der zugehörigen Zeichenkette (**Element**, Englisch *item*) besteht.
Die Stammvariable ist mit **Zeichenketten** indexiert. Auch wenn in der Praxis überwiegend Ziffern vorkommen, werden diese als Zeichenketten gespeichert. 40 und 040 adressieren **unterschiedliche** Indexpositionen, da sie verschiedene Zeichenketten sind. Stammvariable ermöglichen andererseits Zeichenketten als Index (zum Beispiel Ortskenner aus Autokennzeichen) und damit manche Gestaltungsmöglichkeiten, die dem Array fehlen.
Der wichtigste Unterschied besteht darin, dass das Array mit natürlichen Zahlen 1, 2, ... **numerisch** indexiert ist. 40 und 040 adressieren **dieselbe** Indexposition. Ein von der Anwendung zu aktualisierendes Element 0, wie es bei der durchgezählten Stammvariable standardmäßig als Zähler dient, gibt es im Array nicht. Das Array besitzt vom System bereitgestellte Zähler für besetzte Elemente und vorhandene Indexpositionen.

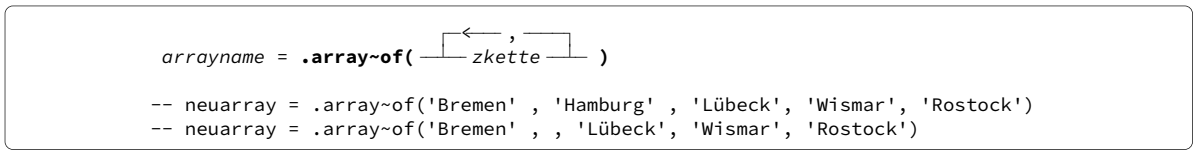
```
-- Stammvariable:
 stamm.6   = 'Stralsund'          -- übliche Schreibweise
 stamm.[7] = 'Greifswald'         -- alternative Schreibweise
 say stamm.[i+2]                  ⇒  Greifswald    -- vorausgesetzt: i = 5
 stamm.['HRO'] = 'Rostock'        -- Index ist nicht auf Ziffern beschränkt
 stamm.4.6.2                      -- Element in dreidimensionaler Stammvariable
 parse var stamm.i ...            -- Parse-Syntax für Variablen verwenden
 stamm~isA(.stem)                 ⇒  1 (sonst 0)   -- Stammname hier ohne Punkt

-- Array:
 hanse = .array~new               -- Array muss vor Benutzung angelegt werden
 hanse[7] = 'Greifswald'          -- wird  ohne  Punkt geschrieben
 say hanse[i+2]                   ⇒  Greifswald    -- vorausgesetzt: i = 5
 say hanse[0007]                  ⇒  Greifswald    -- numerisch 0007 = 7
 hanse[4,6,2]                     -- Element in dreidimensionalem Array
 parse value hanse[i] with  ...   -- andere Parse-Syntax verwenden
 hanse~isA(.array)                ⇒  1 (sonst 0)
```

Übersicht –zum Teil subtiler– Notations-Unterschiede. Methode **~ isA** prüft die Objektart.
Zum einfacheren Verständnis dient nachfolgend insbesondere das eindimensionale Zeichenketten-Array.

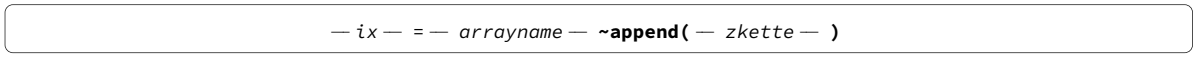
Array anlegen und mit Daten versehen

```
                          ┌──── 0 ────┐
 arrayname = .array~new( ─┴─ topindex ─┴─ )

-- neuarray = .array~new        ⇒   Array noch ohne Indexpositionen
-- neuarray = .array~new(100)   ⇒   Indexpositionen 1 bis 100 existieren
-- neuarray[7] = 'Greifswald'   ⇒   ordnet Indexposition 7 das Element 'Greifswald' zu
```

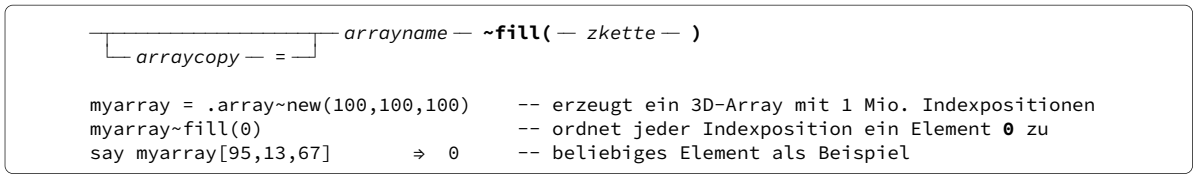
Diese Methode legt ein Array *arrayname* an. Im ersten Beispiel ist es völlig leer. Die Anzahl der existierenden Indexpositionen ist Null.
Im zweiten Beispiel verfügt das neue Array von Anfang über 100 existierende Indexpositionen, die aber keine Elemente haben. Man sagt, die Elemente sind unbesetzt.
Die dritte Anweisung versieht Indexposition 7 des Arrays mit einem zugehörigen Element, bei dem es sich um die Zeichenkette „Greifswald“ handelt. Index 7 hat jetzt ein besetztes Element.
In Kombination mit dem ersten Beispiel würden zusätzlich zur Indexposition 7 auch die noch nicht existierenden Indexpositionen 1 bis 6 erzeugt. Im Array existieren immer alle Indexpositionen unterhalb des höchsten definierten oder benutzten Index. Die zugehörigen Elemente sind erst einmal unbesetzt, bis ihnen ein Wert zugeordnet wird.



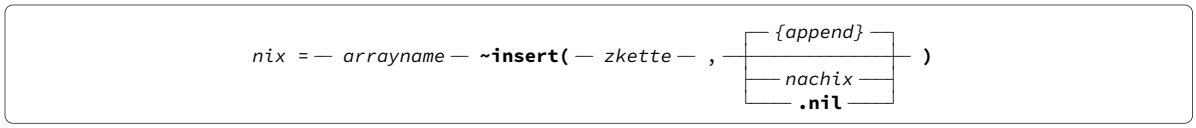
Legt für die im ersten Beispiel 5 übergebenen Argumente ein Array mit 5 Indexpositionen und versieht jedes mit einer Zeichenkette als Element.
Im zweiten Beispiel ist das 2. Argument vorhanden, aber leer (nichts zwischen den Kommas). In diesem Fall bleibt Index 2 unbesetzt, hat also kein Element.



Sucht das letzte besetzte Element im Array und ordnet der darauf folgenden Indexposition das Element *zkette* zu.



Unbesetzte Indexpositionen in *arrayname* erhalten ein Element zugeordnet. Dann werden alle Elemente mit *zkette* überschrieben. Wird der Rückgabewert benutzt, ist *arraycopy* eine Kopie von *arrayname*.

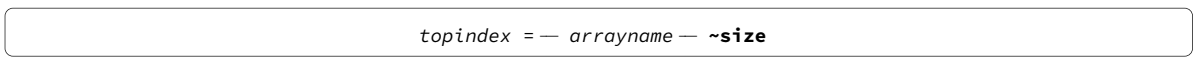


Fügt *zkette* als neues Element hinter Index *nachix* in das Array ein und liefert die neue Position *nix*. Alle folgenden werden um 1 erhöht. Ist *nachix* gleich **.nil** (siehe nächsten Abschnitt), wird am Anfang eingefügt. Fehlt *nachix*, ist die Funktion wie **~append**.

Das Kopieren einer Datei in ein Array mittels Stream-Methode **~arrayin** wurde bereits auf Seite 27 beschrieben.

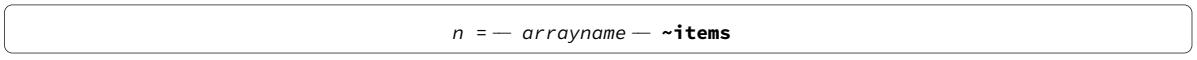
Informationen über ein Array

Um den Zustand „keine Daten vorhanden“ anzuzeigen, wird von Rexx herkömmlich die *leere Zeichenkette* benutzt. Seit ooRexx 3.0 gibt es das Objekt **.nil** zu diesem Zweck. In Vergleichen kann dessen Name **.nil** ganz normal in den Programmcode geschrieben werden. Man kann auch einer Variablen den Wert **.nil** zuweisen. Dort wo ooRexx eine Zeichenkette erwartet (Daten zur Ausgabe am Bildschirm, Schreiben von Dateizeilen), wird dieses Objekt durch die Zeichenkette „The NIL object“ ersetzt.

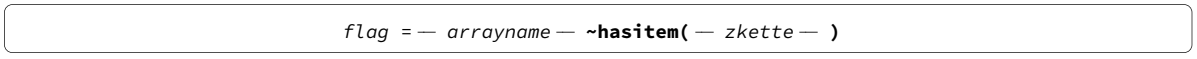


Liefert die Nummer des höchsten aktuell existierenden Index in dem Array, kann 0 sein.

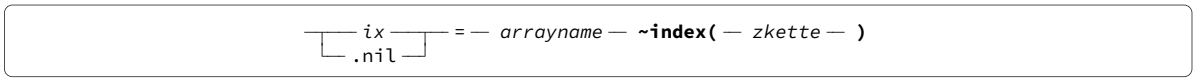
Wenn *dim* fehlt, wird die Anzahl der Dimensionen geliefert oder 0 für ein leeres Array. Falls Dimension *dim* angegeben ist, wird deren Anzahl Indexpositionen geliefert oder 0 falls die Dimension nicht existiert.



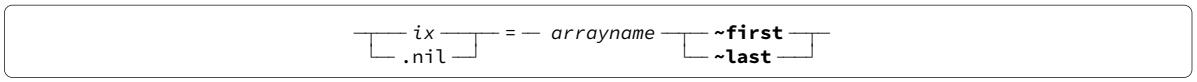
Liefert die Anzahl der besetzten Elemente im Array, kann 0 sein.



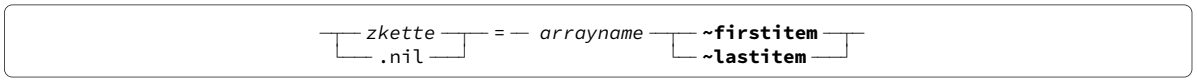
Sucht mit striktem Vergleich **==** (siehe Seite 56) ob *zkette* als Element im Array vorkommt. Wenn ja, wird 1 geliefert, sonst 0.



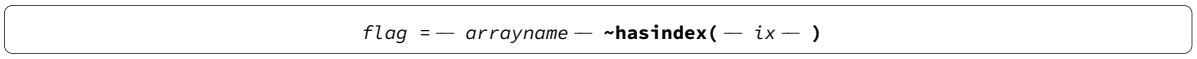
Führt dieselbe Suche wie **~hasitem** durch, liefert jedoch die **Indexposition** wenn gefunden, sonst **.nil**. Ist *zkette* mehrmals vorhanden, wird das erste Vorkommen gemeldet.



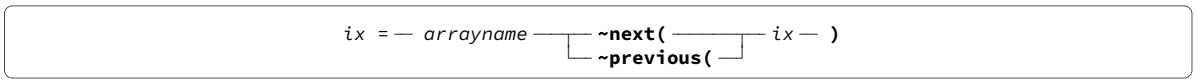
Diese beiden Methoden liefern den **Index** des ersten/letzten besetzten Elements. Wenn es kein besetztes Element gibt, wird **.nil** geliefert.



Diese beiden Methoden liefern den **Inhalt** des ersten/letzten besetzten Elements oder **.nil** wenn keines existiert.

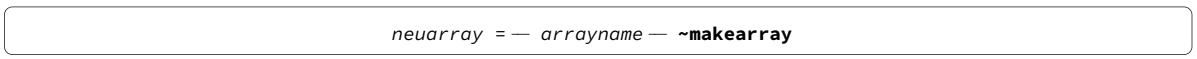


Liefert 1 falls zu Index *ix* ein besetztes Element gehört, sonst 0.



Ausgehend von der angegebenen Indexposition wird der Index des nächsten/vorigen besetzten Elementes geliefert. Ist in der Suchrichtung keines vorhanden, wird **.nil** geliefert. Der Index *ix* darf größer sein als der höchste im Array existierende Index; **~next** liefert dann **.nil**.

Array kopieren (Sonderfälle)

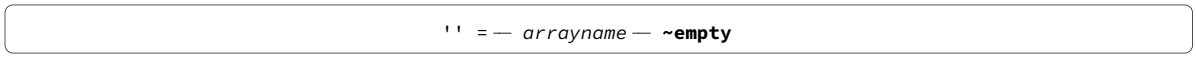


Liefert ein eindimensionales Array, das alle Elemente von *arrayname* in derselben Reihenfolge enthält. Unbesetzte Indexpositionen werden nicht kopiert.

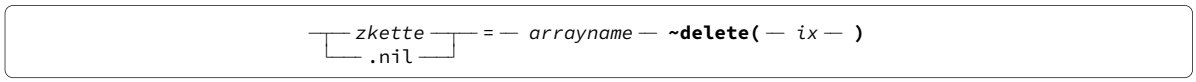
Ab Position *ix* des eindimensionalen Arrays *arrayname* werden *n* Indexpositionen[1] kopiert. Die erste kopierte Position erhält Index 1 in *neuarray*. Ist *n* größer als die Zahl der vorhandenen Positionen, wird nur bis zum Ende von *arrayname* kopiert.

[1] Laut *ooRexx Reference* S. 258 werden nur „items“, also besetzte Elemente kopiert, was aber nicht mein Testergebnis ist.

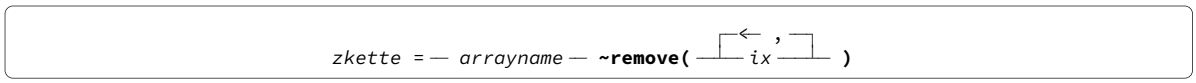
Daten aus einem Array löschen



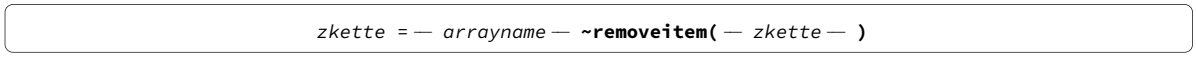
Löscht alle Elemente in dem Array, wobei die besetzt gewesenen Indexpositionen erhalten bleiben. Liefert die leere Zeichenkette.



Löscht die Indexposition *ix* und liefert das zugehörige gelöschte Element. War das Element unbesetzt, wird **.nil** geliefert. Im Array rücken alle folgenden Daten um eine Indexposition nach vorn. Existiert Index *ix* nicht, wird nichts gelöscht und ebenfalls das **.nil** zurückgeliefert.



Liefert den Inhalt des an Position *ix* stehenden Elementes und löscht es. Die Indexposition bleibt erhalten und repräsentiert danach ein unbesetztes Element. War das Element unbesetzt, wird **.nil** geliefert.[2]



Arbeitet wie **~remove**, jedoch wird mit strikten Vergleich **==** nach dem ersten Element gesucht, dessen Inhalt gleich *zkette* ist.

11.1 Arrays stabil sortieren mit SORT2

Stabil bedeutet, dass Daten mit identischem Sortierwert dieselbe Reihenfolge behalten wie vor den Sortieren. Die unveränderte Reihenfolge ist zum Beispiel für Logauswertungen wichtig.
Die externe Bibliothek **rgf_util2.rex**[3] bietet mit ihrer Funktion **sort2** Sortiermöglichkeiten, die **SysStemSort** (Seite 36) nicht hat und welche die **SortWith** Methoden in dieser leicht anzuwendenden Form nicht bieten:

- korrektes Sortieren von Zahlen nach dem numerischen, auch negativen Wert,
- Sortieren nach **mehreren** Feldern desselben Elements, sowie
- **dabei** Kombination von auf- und absteigender Sortierung.

```
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. SORT2 kann die Exponentialdarstellung verarbeiten. `960.2` und `9.602E2` haben denselben Sortierwert.
Für **sort2** müssen die Daten als Array vorliegen. Auf Seite 45 ist an einem konkreten Beispiel dargestellt, wie Stammvariable auf einfache Weise zur Sortierung in ein Array und wieder zurück konvertiert werden.

[2] Im Gegensatz zu dem was die *ooRexx Reference* auf Seite 257 sagt und das Diagramm zeigt, akzeptiert die Methode nur eine einzige Indexposition. Anderenfalls stoppt das Programm mit Returncode 93.926.
[3] Sie wurde von *Rony G. Flatscher*, Professor an der Wirtschaftsuniversität Wien, veröffentlicht, erstmals 2009.

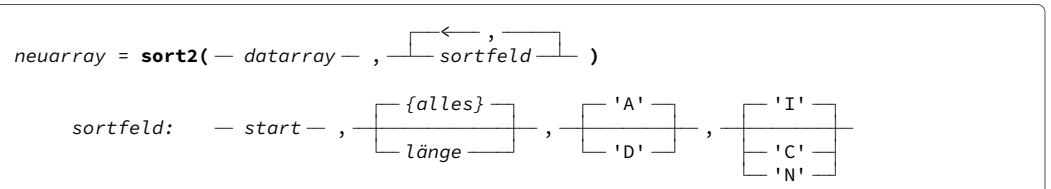
Array vor der Sortierung

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

Die Beispieldaten des Arrays `meteo` stammen von *MeteoSchweiz* und sind nach dem Stationsnamen geordnet.

SORT2 Syntax

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




```
-- Der Aufruf sortiert auch datarray neu.
-- Soll datarray unverändert bleiben, Methode ~copy an dessen Namen anhängen.
```

Argument *datarray* ist ein Array mit den zu sortierenden Daten. Jedes Element im Array entspricht einer Datenzeile mit mehreren Feldern. Rechts von *datarray* folgt mindestens eine Argumentengruppe *sortfeld*, die jeweils ein Feld innerhalb der Zeile definiert, nach dessen Inhalt sortiert werden soll. Mehrere, durch Komma getrennte Sortierfelder sind möglich.
Als erstes Argument innerhalb *sortfeld* gibt *start* die linke Grenze des Sortierfeldes (Spalte) an. Dem folgt dessen *länge* (Anzahl der Spalten).[4] Voreinstellung ist *bis zum Ende*.
Als nächstes Argument bewirkt ein Buchstabe `A` oder `D` (descending) wie gewohnt auf- oder absteigende Sortierung.
Es folgt der Sortiermodus. `I` ignoriert Groß- und Kleinschreibung, während `C` (case) Großbuchstaben vor Kleinbuchstaben einsortiert. Modus `N` bewirkt Anwendung der numerischen Sortierung auf dieses Sortierfeld.
sort2 erzeugt ein neues Array-Objekt *neuarray* mit den sortierten Daten.[5] 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.

1. Beispiel: Anwendung gewöhnlicher Sortierung

Die Elemente des Arrays `meteo` sollen **absteigend** nach der Stationshöhe umsortiert werden. Diese steht in den Spalten 22 bis 25 jedes Elementes. 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. Sortierbefehl und Ergebnis sehen so aus:

[4] 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.
[5] Wird die Funktion über CALL aufgerufen (S. 57), ist die erzeugte Systemvariable RESULT nicht wie üblich eine Zeichenkette, sondern ein Objekt vom Typ Array.

```
 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
```

Das Ergebnis der Sortierung nach Stationshöhe ist jetzt als 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 angewendet.

2. Beispiel: Anwendung numerischer Sortierung

```
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. Innerhalb gleicher Stationshöhe soll er **aufsteigend** sortiert sein. Die Druckangaben sind nicht spaltengleich untereinander angeordnet, sondern verteilen sich auf die Spalten 28 bis 36.
Somit muss der numerische Sortiermodus N aktiviert werden. Luftdruck-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.

Von der Stammvariable zum sortierten Array-Objekt und zurück

In existierenden Rexx-Programmen werden die Daten oft als als durchgezählte Stammvariable (Seite 35) vorliegen. Soll die Programmänderung möglichst klein sein, bleibt der Weg, ein Array zwischenzuschalten. Wir verwenden Stammvariable `zeile.` als Beispiel existierender Daten:

```
/*         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
```

Die Schleife kopiert diese Stammvariable in ein Array `meteo` das unmittelbar mit den oben gezeigten Aufrufen von `sort2` sortiert werden kann.
Nach dem Sortieren bleibt nur noch, das Ergebnisarray `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
-- zählt den Index hoch: 1, 2, ...

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

-- Alternative 3: DO WITH ... OVER Schleife
-- Benutzt den Array-Index für die Stammvariable

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

Alternative 2 und 3 nutzen die beiden neuen Schleifenarten (siehe Seite 14) für Datenkollektionen. Es muss aber ein Zähler als Index der Elemente in der Stammvariable mitgeführt werden. Der Zählername ist frei wählbar; alle drei benutzen **i**. Der Variablenname für das aktuelle Element, hier **elem**, ist ebenfalls frei wählbar.

Herunterladen der Bibliothek rgf_util2.rex

Die jeweils aktuelle Version der Programmbibliothek liegt BSF4ooRexx bei (siehe S. 59). Das Programm befindet sich im Verzeichnis `\bsf4oorexx` der ZIP-Datei.
Alternativ kann es von *Sourceforge* mit dem 1. Link der folgenden Übersicht heruntergeladen werden. Darin ist **850** die Nummer der BSF4ooRexx Version, die eventuell anzupassen ist.
Die übrigen drei Links führen zur Wirtschaftsuniversität Wien.

```
sourceforge.net/p/bsf4oorexx/code/HEAD/tree/branches/850/bsf4oorexx.dev/bin/rgf_util2.rex

wi.wu.ac.at/rgf/rexx/orx20/2009_orx20_RGF_UTIL2-20100120-refcard.pdf
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
```

In jedes Programm, das die Funktionen dieser Bibliothek nutzen will, ist die folgende Direktive am Dateiende(!) einzufügen:

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

12 Multiwerkzeug USE ARG

Statt für 1000 Berechnungen ein Unterprogramm 1000 mal aufzurufen, genügt eventuell ein einziger. Es geht darum, dem Unterprogramm direkten Zugriff zu den 1000 Werten zu geben, die es bearbeiten soll. Mit dem Schlüsselwort **USE ARG** können Haupt- und Unterprogramm **gemeinsam** auf große Datenmengen zugreifen, ohne dass diese kopiert werden müssen. Das ist möglich, wenn sie als Stammvariable oder als Array gespeichert sind.
Das Hauptprogramm benutzt den Variablen- oder Arraynamen als Argument beim Aufruf des Unterprogramms. Dort wird USE ARG dazu benutzt, auf diese Daten zuzugreifen. Ein einfaches Beispiel:

- Das Hauptrogramm verwaltet ein Array mit Geldbeträgen.
- Es ruft ein Unterprogramm auf und übergibt den Arraynamen als Argument.
- Das Unterprgramm greift per USE ARG auf das Array zu.
- Es arbeitet das Array ab und fügt jedem Eintrag die Mehrwertsteuer hinzu.
- Wenn das Unterprogramm endet, erhält das Hauptprogramm wieder die Kontrolle.
- Es findet die angepassten Rechnungsbeträge vor.

Das Unterprogramm kann Daten ändern, löschen oder hinzufügen. Für Arrays stehen dem Unterprogramm alle Methoden zur Arbeit mit besetzten und unbesetzten Indexpositionen zur Verfügung. Stammvariable müssen nicht durchgezählt sein.

```
-- 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
```

Programm **hauptprog.rex** stellt seine Stammvariable `hanse.` dem Unterprogramm zur Verfügung. Durch ein Komma getrennt, folgt normale Variable `testvar` als zweites Argument.

```
-- 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
-- später für das Hauptprogramm nicht sichtbar.
say einfach                         ⇒  Ostsee
einfach = 'abcd'

return 'Rückgabe'           -- herkömmliche Datenrückgabe
```

Das Unterprogramm benutzt USE ARG, um die Argumente zu lesen. Es kennt die vom Aufrufer benutzten Namen (hanse., testvat) nicht und verwendet beliebige eigene Namen. 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 des Unterprogramms in `demo.` findet das Hauptprogramm anschließend in seiner Stammvariable `hanse.` vor. Denn alle Manipulationen des Unterprogramms an `demo.` wurden in Wirklichkeit an `hanse.` ausgeführt.
Änderungen an gewöhnlichen Variablen wie `testvar` durch das Unterprogramm sieht das Hauptprogramm dagegen nicht.

```
call unterprog              -- Fehlendes Argument im UNTERPROG-Aufruf

-- Unterschiedliche Behandlung im aufgerufenen Unterprogramm:

arg awert                   -- arg   sieht die leere Zeichenkette
say awert                   ⇒  ''

use arg bwert               -- use arg   sieht eine undefinierte Variable
say bwert                   ⇒  'BWERT'
```

ARG verwendet für das unbenutzte Argument die *leere Zeichenkette*. **USE ARG** dagegen behandelt es als nicht initialisierte Variable. Ihr Wert ist der eigene Name in Großbuchstaben (siehe Grundregeln auf Seite 55).
Wie im Beispiel gezeigt, können **ARG** und **USE ARG** im selben Unterprogramm nebeneinander verwendet werden.
USE ARG akzeptiert mehrere Argumente nur, wenn sie **durch Kommas getrennt** sind. Steht aber hinter **USE ARG** kein Name zwischen den Kommas, ignoriert **USE ARG** stillschweigend das an dieser Position übergebene Argument.
Initialisieren von Stammvariablen (Seite 38), auf die über **USE ARG** zugegriffen wird, ist im Unterprogramm nutzlos. Diese Aktion erzeugt ein ganz neues Objekt und kappt damit die logische Verbindung zum Hauptprogramm. In `hanse.` wird dadurch weder diese Initialisierung wirksam, noch irgendwelche andere Änderungen im Unterprogramm.

12.1 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 44 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

if verzeichnis~isA(.array) then nop    -- optional: Test auf Array
else return -1                         -- kein Array

say verzeichnis[5]                     ⇒  'beliebige Zeichenkette'

verzeichnis[5] = 'geänderte Zeichenkette'
return 0                               -- Fehlercode 0 zurückgeben
```

Im Unterprogramm wird das übergebene Array unter dem Namen `verzeichnis` in Zugriff genommen. Methode **~isA** kann benutzt werden, um zu prüfen, ob es sich wie erwartet um ein Array-Objekt handelt. Denn das Unterprogramm ist ja mit arraytypischen Anweisungen codiert. Hier sind es die Ausgabe des Elements 5 am Bildschirm und seine anschließende Änderung.
Der gemeinsame Zugriff auf dasselbe Objekt (Array) erfolgt nach denselben Regeln wie im vorigen Abschnitt für die Stammvariable beschrieben. Alle Änderungen des Unterprogramms am Array `verzeichnis` sieht das Hauptprogramm in „seinem“ Array `telefonbuch`. Denn es handelt sich letztlich um dasselbe Objekt.

12.2 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 ::DIREKTIVE 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 an kegel

::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 19. 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 Programmdateien. Das gilt auch für Aufrufe innerhalb derselben Bibliothek.
Interne Unterprogramme einer Routine, wie das zu Routine **kegel** gehörende, mit Label `uprog:`

beginnende, sind unsichtbar für die anderen Routinen. Das Unterprogramm seinerseits sieht nur die Variablen der eigenen Mutter-Routine. 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
-- in Funktionsbibliothek BIBLIOTHEK.REX
...

call pyramide argument1 , argument2 ...

...

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

Dieses Beispiel zeigt den Aufruf von Unterprogramm **Pyramide** mittels **CALL** Schlüsselwort. Genauso kann die Funktions-Syntax benutzt werden (vorausgesetzt, dass eine Zeichenkette zurückgeliefert wird):

```
result = pyramide(argument1,argument2)
```

Um Routinen einer Funktionsbibliothek nutzen zu können, muss sie am Ende des aufrufenden Programms per **::REQUIRES** bekannt gemacht werden. Die Erweiterung **.rex** kann weggelassen werden, da seit Februar 2020 Rexx zuerst eine **.cls**-Datei dieses Namens und danach eine **.rex**-Datei sucht.

13 Klassik versus Objekt

Dieses Kapitel vergleicht eine *klassisch* programmierte Lösung mit einer **objektorientierten**. Die Datei `hanse.dat` mit den zu sortierenden Beispieldaten sieht 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.

13.1 Methode SORTWITH in einem klassischen Programm

Das Programm beginnt mit der Wahl, ob nach Vorwahlnummern oder Ortsnamen sortiert werden soll:

```
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
```

Argument **V**[orwahl] oder **O**[rtsname] setzt die Sortierspalten. Ortsnamen mit weniger als 10 Zeichen sind kein Problem.

```
indatei = .stream~new{'hanse.dat'}    -- zu lesende Datei ...
tabelle = indatei~arrayin             -- in Array TABELLE kopieren
```

Methode `arrayin` legt Array `tabelle` an und kopiert den Dateinhalt hinein.

```
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. Variablenname `idx` ist beliebig.

```
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 hier keine unbesetzten Elemente 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, kennt sie nur die Buchstaben a-z.

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

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

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

13.2 Objektorientiertes Programm zum Vergleich

Dieser Abschnitt[1] ist ausdrücklich keine Anleitung.
Für die Datenelemente müssen Objektnamen erdacht werden. Jede Zeile in `hanse.dat` enthält 2 Felder (Attribute), hier `vorwahl` und `ortsname` genannt. Als Name für das resultierende Satzformat aus diesen zwei Feldern wird `hansesort` gewählt.

Klassendefinitionen

Klassendatei `oohanse.cls` wird 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] Er ist aus dem mit Rexx gelieferten Beispielprogramm `sortComposite.rex` abgeleitet.

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 Feld `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. Die eingebauten Vergleichsmethode `compareto` erzeugt in dem komplizierten Ausdruck hinter `return` den Rückgabewert 1, 0 oder -1. Damit zeigt Methode `compare` dem Rexx-Interpreter an, ob *größer*, *gleich* oder *kleiner* für den Vergleich von `links` mit `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 zeigt, wie durch eine winzige Änderung die Umkehrung der Sortierfolge von auf- nach absteigend erreicht werden kann: das zusätzliche Minuszeichen am Beginn des Ausdrucks rechts von `return`. Aus -1, 0 und 1 wird damit 1, 0 und -1.
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 51.

```
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 die benötigte Klassendatei mit. Seit Februar 2020 kann die Erweiterung `.cls` weggelassen werden, da ooRexx zuerst nach `cls`-Dateien sucht.
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.
Die Datensätze im Array `tabelle` sind jetzt Objekte, wie in Klasse `hansesort` definiert. Unsere Methode `string` hat sie für die Ausgabe mit `say` aufbereitet.

13.3 Vergleich des Aufwandes

Die Beschreibung der klassischen Wegs in 13.1 ab Seite 51 belegt auf dem Papier 265 mm Satzhöhe, die objektorientierte Version 13.2 ab Seite 52 belegt 530 mm, also genau das Doppelte.

14 Erinnerung an einige Grundregeln

14.1 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
```

Schritt für Schritt:

- **SAY** erkennt der Interpreter als Rexx-Schlüsselwort. Es bewirkt abschließend die Ausgabe des Zeileninhalts am Bildschirm.[1]
- Die in Hochkommas (') eingeschlossene Zeichenkette `Es ist` bleibt unverändert.
- `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 `Uhr` mit den Leerstellen darin bleibt unverändert.
- **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 Binärwert (siehe S. 33) 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 57.

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 REXX mit einem Kommentar beginnen */
/*
   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 (seit 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 Unix-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 seit ooRexx 5.0 neuen **select case** (Beispiel S. 51) macht immer strikte **==** Vergleiche.
Vergleiche können logisch mit & (UND), | (ODER) sowie && (exklusives ODER) verknüpft sein. Es werden immer *alle* verknüpften Vergleiche einer Anweisung ausgeführt.

```
if datatype( result ,'N') , result > 0 , result < 100 then ...
```

Seit ooRexx 3.2 gibt es das **Komma** als bedingtes UND. Dabei müssen alle Vergleiche 1 (WAHR) ergeben. Sobald einer 0 ergibt, werden die rechts davon stehenden nicht mehr durchgeführt.

```
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.

---

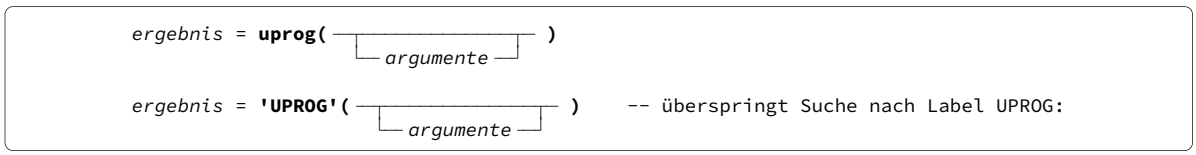
[2] Für die dargestellten 12 Operatoren gibt es weitere 12 alternative Schreibweisen mit gleicher logischer Bedeutung, die hier weggelassen sind.
[3] Vorsicht. Da manche hexadezimalen Werte wie 0E01 von Rexx für numerisch gehalten werden, denn in Exponentialdarstellung ist $0E01 = 0 \cdot 10^1 = 0$, sollten hexadezimale Zeichenketten immer strikt verglichen werden.
[4] Genauer gesagt, alle Zeichen der Klasse SPACE, wie bei **xrange** auf Seite 9 beschrieben.

14.2 Aufrufen von Unterprogrammen

Ausführung wie in der Kommandozeile eingegeben

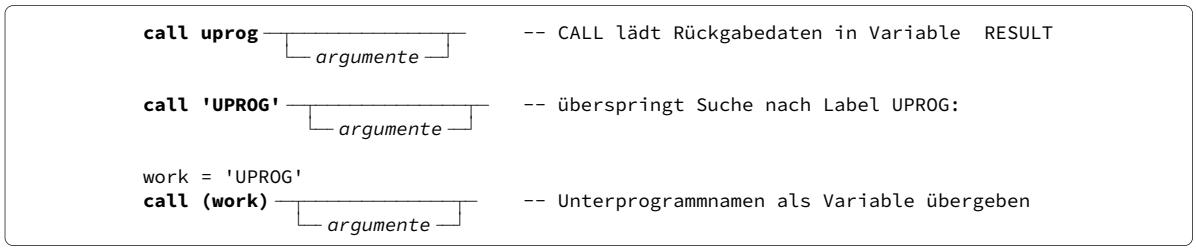
Wenn Rexx nach dem Einsetzen von Variableninhalten und Funktionswerten ein unbekanntes Wort am Zeilenanfang findet, wird diese Zeile als Systembefehl aufgefasst und ans Betriebssystem weitergeleitet. Dort wird sie wie eine Eingabe in der Kommandozeile behandelt. Windows sucht in der Reihenfolge: Systembefehl, 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 kann das laufende Programm ein anderes Programm starten wie in der Kommandozeile. Dieses 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 59). Der Returncode ist als Variable **RC** im aufrufenden Programm verfügbar.

Aufruf als Funktion



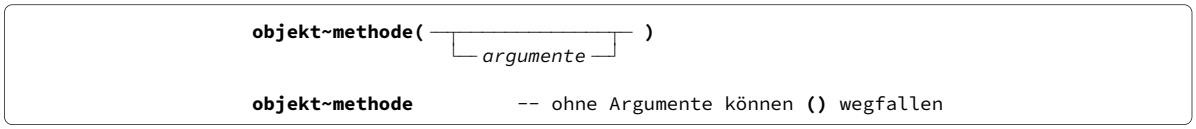
Ein Funktionsaufruf ist erkennbar an der Klammer unmittelbar hinter dem Namen. Es können bei Bedarf *mehrere*, durch Komma *logisch* geteilte Zeichenketten übergeben werden (*Argumentenkette*). Siehe Syntaxbeispiel S. 60 oben.
Soll ein Unterprogramm als Funktion aufrufbar sein, **muss** es mit dem Schlüsselwort `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.
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.
- Werden keine Argumente übergeben, können die Klammerzeichen **()** weggelassen werden.

Für die Suchreihenfolge kommt es auf die aktuelle Klassenhierarchie an.

14.3 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.[5] 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[6] und
  - die in Bibliothek **rexxutil.dll** mitgelieferten **Rx...** und **Sys...** Funktionen.[7]
- 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?
- 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*

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

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 18).

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, zum Beispiel ein plattformübergreifendes Grafisches User Interface (GUI)
Die jeweils aktuelle Version ist heruterladbar über URL:
`sourceforge.net/projects/bsf4oorexx/files/GA`

14.4 Bedeutung von Windows Returncodes

```
SysGetErrorText( — rc — )

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

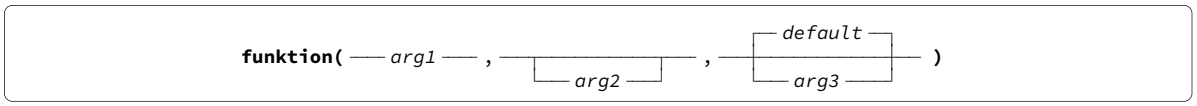
Kann Betriebssystem Windows eine Aktion nicht ausführen, wird der Grund dafür per numerischem Returncode gemeldet. Der zum Returncode gehörende Fehlermeldungstext kann über diese Funktion geliefert werden. Nachfolgend eine Auswahl von Returncodes:

```
     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  Kein Zugriff, da Datei 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.
```

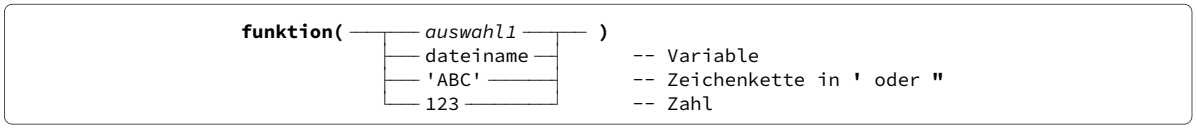
Der Rexx-Interpreter benutzt für Programmfehler eigene Returncodes. Deren Bedeutung läßt sich in gleicher Weise über die Funktion **ErrorText()** anzeigen.
Anhang C der Handbuchs (Datei *rexxref.pdf*) enthält außerdem eine ausführliche Liste.

15 Erklärung der Syntaxdiagramme

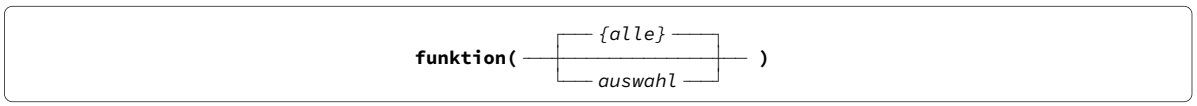
Kursivschrift in den Diagrammen sind symbolische Namen, für die konkret einzusetzen sind: ein Variablenname, eine konstante Zeichenkette in Hochkommas (" oder ') oder eine Zahl. Zahlen brauchen keine Hochkommas.



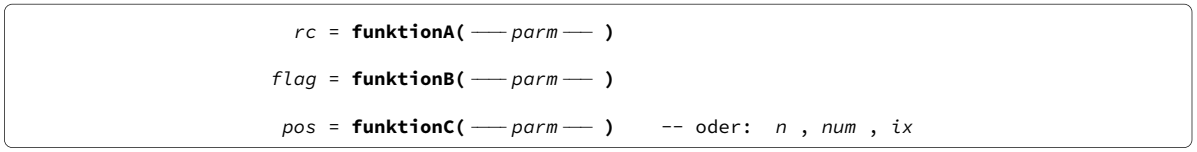
Dieser Funktion können beim Aufruf 3 durch **Komma** getrennte *Argumente* (auch Parameter genannt) als *Argumentenkette* mitgegeben werden. Im Beispiel muss *arg1* vorhanden sein, während die anderen beiden auch weggelassen werden können.
Der oberhalb der Hauptlinie angegebene Wert *default* ist wirksam, wenn das 3. Argument weggelassen wird.
Die trennenden **Kommas** sind in den Diagrammen der Einfachheit halber immer dargestellt. Es gilt die Regel, dass *rechts vom letzten benutzten Argument alle Kommas weggelassen werden können.*



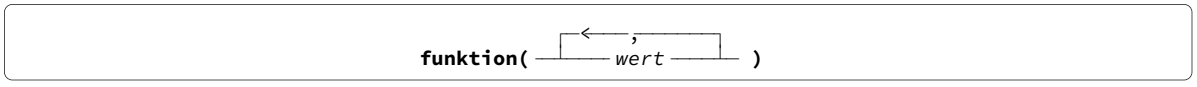
Eine „Leiter“ zeigt die zulässigen Argumente an, von denen eines ausgewählt werden muss.
Die Zeichenkette in Hochkomma kann anstelle einer Variablen verwendet werden, wenn dies einfacher oder leichter lesbar ist. Durch die Hochkommas wird sicher verhindert, dass statt des gewünschten Werts ABC der Inhalt einer zufällig existierenden Variablen ABC an die Funktion übermittelt wird. Das ist besonders bei kurzen Variablennamen eine typische Fehlerquelle.



Manchmal kann der konkrete Defaultwert, zum Beispiel die restliche Länge der Zeichenkette, nicht angegeben werden. Dann steht dort in *{ }* eine erklärende Ersatzbezeichnung wie *{alle}*.



Wie eine Wertzuweisung im Programm sieht das Syntaxdiagramm aus, wenn es wichtig ist, die Art der zurückgelieferten Information darzustellen. Eine feststehende Bedeutung haben:
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 59.
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*, *n*, *num* oder *ix* zeigt, dass die Funktion eine Längenangabe, Zählwert oder Indexposition liefert, also eine positive ganze Zahl. Außer beim Index ist auch 0 möglich.



Diese Funktion akzeptiert auch mehrere Werte, die durch Komma zu trennen sind.

Index

Inhaltsverzeichnis