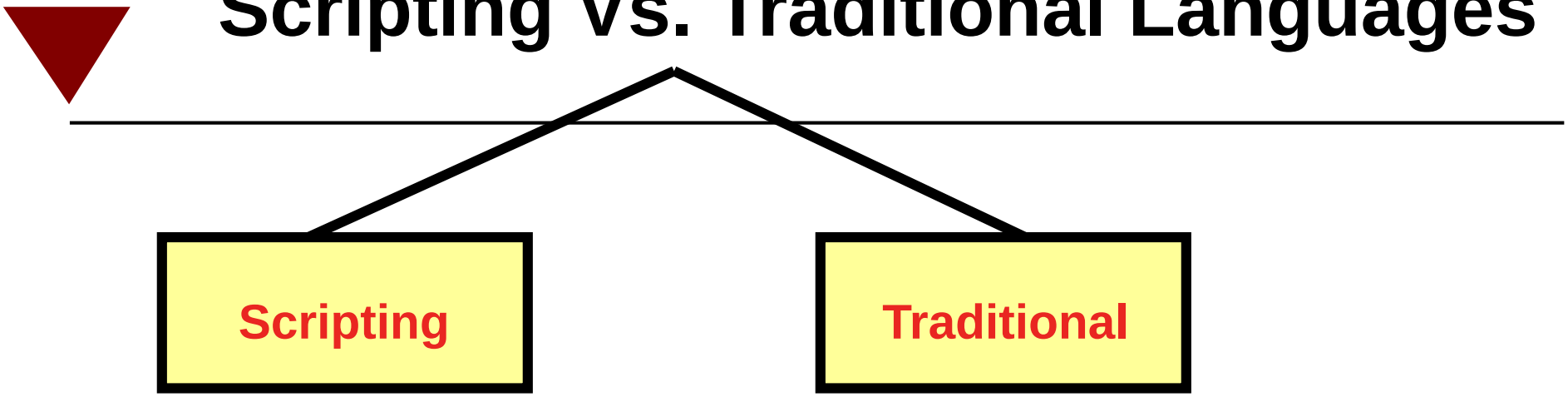
REXX Einführung

REXX = REstructured eXtended eXecutor
1979 von IBM Fellow Mike F. Cowlishaw vorgestellt
(hatte vorher u. a. PL/I entwickelt)

REXX ist ein Standard: ANSI X3.274-1996

Scripting Vs. Traditional Languages



- -- High level
- -- Interpretive
- -- More productive
- -- Varying degrees of automatic variable management
- -- Shifts burden to the machine
- -- Acceptable execution speed

- -- Lower level
- -- Compiled
- -- More detail-oriented
- -- Manual variable manageme
- -- Pre-declared variables
- -- More programmer effort
- -- Optimize execution speed

REXX, Perl, Python, Tcl/Tk, Ruby, others

C/C++, COBOL, Java, Pascal, others

Warum REXX ?

Releativ simpel …

Minimale Syntaxregeln

- **wenige Spezialzeichen, reservierte Variablen etc.**
- **formatfreie Sprache**
  - **ignoriert überschüssige Leerzeichen, Tabs ...**
- **Kommentare: 2 Möglichkeiten**

  A: beginnt mit /* und endet mit */ (über mehrere Zeilen)

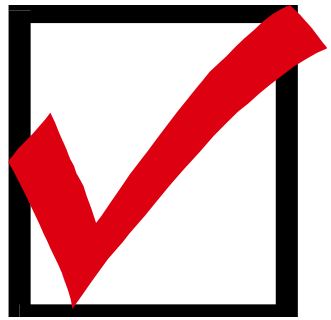
  B: --bis Zeilenende

Automatisches Speichermanagement

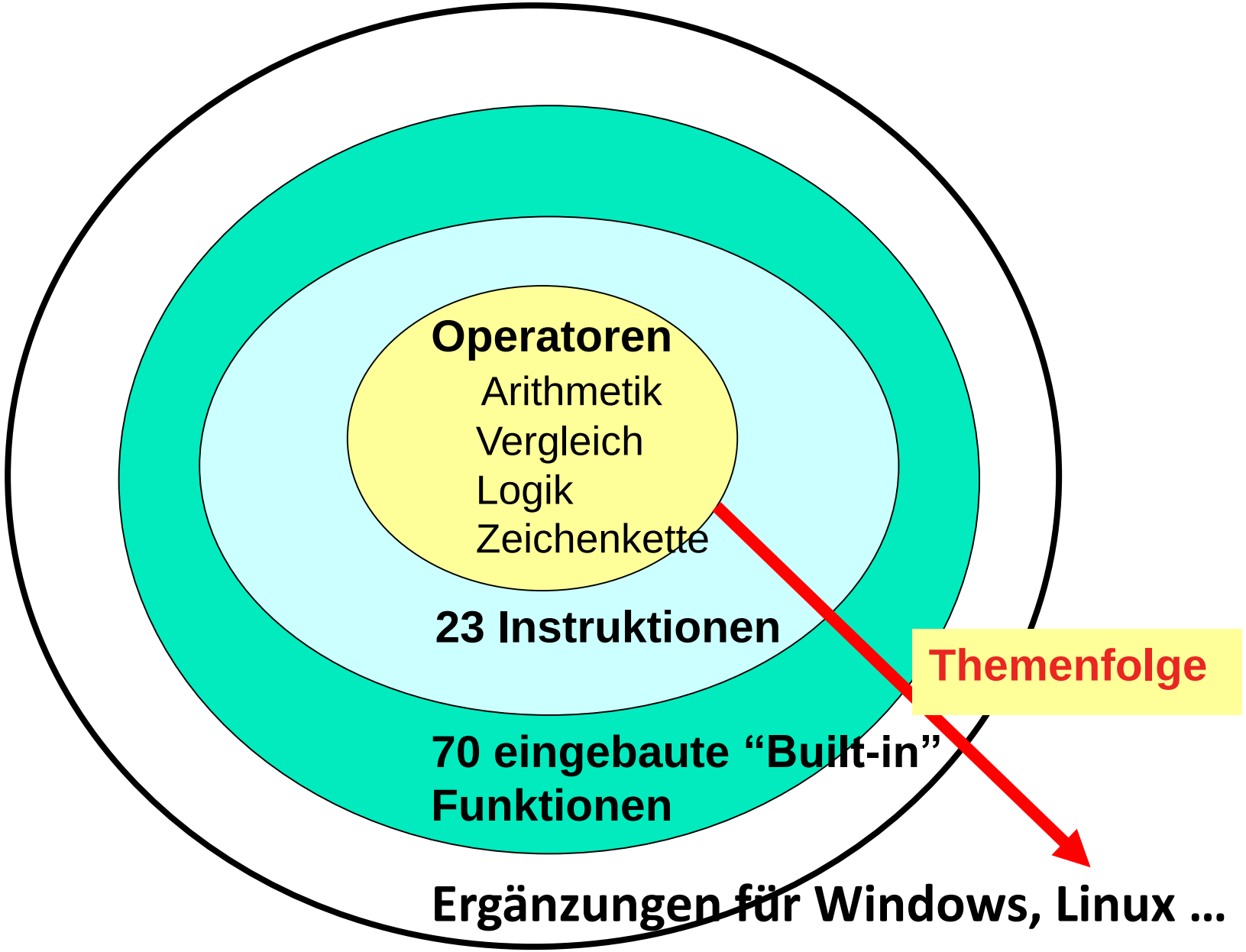
Automatisches Variablenmanagement

- **speichert Variablen intern in GROSS**
- **keine Definitionen notwendig**

  speichert intern alles als Zeichenfolge



Einfache Struktur



REXX läuft überall

Linux--	all versions
Unix--	all versions
BSD--	all versions
Windows--	all versions
Mac OS--	all versions
DOS	all versions (32- and 16- bit)
Handhelds--	Windows CE / Mobile, Android, Symbian, EPOC, DOS emulation, iOS
Embedded--	Embedded Linux, DOS, Windows
Mainframes--	z/OS, z/VM, z/VSE (all versions)
IBM iSeries--	i5/OS, OS/400 (all versions)
Many others--	AmigaOS, OpenVMS, OS/2 …

ooREXX installieren

Bitte herunterladen und installieren:

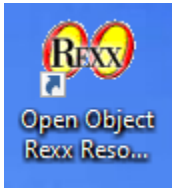
Google „ooREXX Download“

-oder direkt-

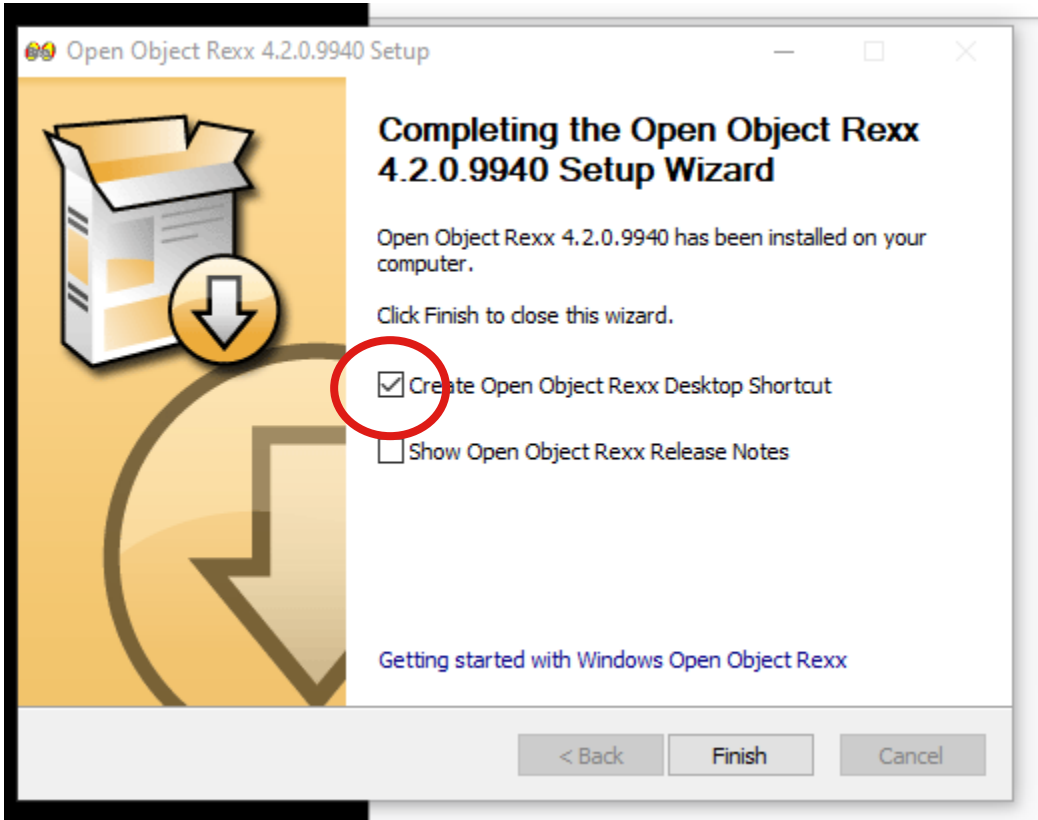
sourceforge.net/projects/oorexx/files/oorexx/4.2.0/ooREXX-4.2.0.windows.x86_32.exe

Hinweise zur Installation ooREXX

- Windows: bitte 32-bit Version installieren, wenn später mal die Nutzung von IBM DB2 auf Windows (kostenlos!) gewünscht wird
- Alle Parameter können im Default bleiben, beim letzten Dialogbild (rechts) bitte dies Häkchen (roter Kreis) setzen.

→ 

Desktop-Icon



Beispiele

Erstes Beispiel

```
/*** Erster Test: Variablen   ***/
Say 'Hallo';  Say 'bmw3'
Say ,
'Hallo'
Say 5 - 2
Say '5-2'
Say '5'- 2

C250 = ‘Hallo’
A = '1'
Bd = '2'
C250 = A + Bd + '3'; Say c250
? = 4; Say ?+3
```

Variablenname bis zu 250 ZeiCHen (a-Z, 0-9 und § # Ü ¢ $? . _)

Ausprobieren in der “DOS-Box”: rexxtry (oder GUI oorexxtry.rex)

Brauche ich 2020 einen neuen Kalender?

```
1   /*1.1. und 1.3. gleicher Wochentag*/
2   Parse Arg xx
3   If Datatype(XX)<> 'NUM' Then xx = Left( Date('S'), 4)
4   Say 'Kalender passend zu' xx
5   Call init
6   Do ii=(xx-1) to 1975 by -1
7     If date('W',xx'0301','S') <> date('W',ii'0301','S') Then Iterate ii
8     a.ii.?s = 'ab M„rz'
9     If date('W',xx'0101','S') <> date('W',ii'0101','S') Then Iterate ii
10    a.ii.?s = 'OK'
11  end
12
13  Do ii=(xx-1) to 1975 by -1
14    If a.ii.?s<>'' Then Say ii':' a.ii.?s
15  End
16  Say 'oder' a.1900
17  Exit 0
18
19  INIT:
20  a.= ''
21  a.1900 = 'OtmarAlt'
22  a.1996 = 'Rhein.Expression'
23  a.1997 = 'Expessionist Campendonk'
24  a.1998 = 'Rhein.Expression'
```

Kalender passend zu 2020
2015: ab März
2009: ab März USA/A4
1998: ab März Rhein.Expression
1992: OK
1987: ab März
1981: ab März KunstDetail
oder OtmarAlt

Datei verändern

```
/*Textdatei: Zeilen verdrehen*/
ein = "D:\altdat.txt"
aus = "D:\neudat.txt"

dat = charin(ein, 1, 999999)  /*knapp 1 Mb einlesen*/
nl = '0d0a'x  /*CRLF*/

Do while dat<>''
  Parse Var dat zeile (nl) dat
  neuez = Length(zeile) '-' Reverse(zeile)
  Call Lineout aus, neuez
End
Exit 0
```

Aus

```
1  Otto
2  nur du Gudrun
3  Nie, Erika, fette Fakire ein
4  Reliefpfeiler
5  Eine treue Familie bei Lima feuerte nie
```

wird

```
4 - ottO
13 - nurduG ud run
28 - nie erikaF ettef ,akirE ,eiN
13 - reliefpfeileR
39 - ein etreuef amiL ieb eilimaF euert eniE
10 - samO-aomaS
```

IP Adresse in Hex-Schreibweise

```
/**/
Parse Arg ip
Parse Var ip i1 '.' i2 '.' i3 '.' i4
Say ip '=',
    d2x(i1,2),
 || d2x(i2,2),
 || d2x(i3,2),
 || d2x(i4,2)
```

ip2x 192.168.0.66

192.168.0.66 = C0A80042

Etwas REXX ...

REXX Komponenten

- Instruktionen:
  - **Schlüsselworte**: Identifizieren die Instruktion.
    Beispiele: Do, End, Leave
  - **Zuweisungen**: Wert einer Variable wird verändert
    Beispiel: a = ‘ +31.12 ‘
  - **Sprungmarke (Label)**: Name mit einem Doppelpunkt (‘:’) am Ende
  - **Kommando**: Betriebssystembefehl wie “DIR”

- **“Built-in” Funktionen**: große Sammlung von Funktionen zur Textverarbeitung und Konvertierung

- **externe Funktionen** – in OS/2, Windows und *ix: rexxutil

Interne Schlüsselworte (8 fehlen hier)

Schlüsselwort	Bedeutung
ARG	Zerlegung der Aufrufargumente eines Programms
CALL	Aufruf von Prozeduren
DO	Blockbeginn, Schleifenbeginn
END	Ende Schleife/Block
EXIT	ProgrammEnde
IF	Verzweigung
ITERATE	ändert den Kontrollfluß in einer Schleife
LEAVE	verläßt eine Schleife vor deren Ende
NOP	Leerer Befehl (No OPeration)
PARSE	Einstellung eines Datenstroms gemäß einer Variablenliste
RETURN	Rücksprung aus einer Funktion/Prozedur
SAY	Ausgabe auf die Standardausgabe
SELECT	Mehrfachauswahl
SIGNAL	Sprunganweisung
TRACE	Debugging

Operatoren

Vergleichsoperatoren

=	Gleichheit
<>	Ungleichheit
><	
¬=	
\=	
<	Kleiner als
>	Größer als
\<	Nicht kleiner als
¬<	
\>	Nicht größer als
¬>	
<=	Kleiner oder gleich
>=	Gößer oder gleich

logische Operatoren

&	Und (AND)
|	Oder (OR)
&&	Exklusiv-Oder (XOR)
\	Nicht (NOT)
¬	

Beispiel:

```
IF 4 < 9 & 2*2=4 THEN
    DO
      SAY "Stimmt"
    END
  ELSE
    SAY "Nee"
```

Operatoren zum Rechnen

Arithmetik

+	addieren
-	subtrahieren
*	multiplizieren
/	dividieren
%	dividieren: ohne Rest
//	dividieren: nur Rest
**	potenzieren

Beispiel:
Say 28/3 28%3 28//3

numerische Datentypen

Ganze Zahlen: -4711; 42

Fließkommazahlen: 3.1415; 47.11

Exponentialzahlen: -3E24; 3.34e-12

REXX speichert als Zeichenkette

```
a='- 12'; b='.23'; C="          34   "
say a+b+c
```

REXX rechnet beliebig genau

```
Say 1000**1000/3
```

(Normalerweise 9 Stellen, aber „numeric digits 500“ klappt)

Rechnen und Logik

Operand	meaning	example
+	add	Say 7.5 + 2.5 → 10.0
-	Substract	Say 7.5 - 2.5 → 5.0
*	multiply	Say 7.5 * 2.5 → 18.75
/	Divide	Say 7.5 / 2.5 → 3
%	divide & truncate	Say 7.5 % 2 → 3
//	divide & return rest	Say 7.5 // 2 → 1.5
**	exponential	Say 7.5 ** 2 → 56.25
= , \=, <>	eqal, not equal	Say 7 <> 14/2 → 0
==	identical (incl. length)	Say ‘ ‘ \== ‘‘ → 1 /*not equal*/
>, >=	greater, greater equal	Say 7 <= 14/2 → 1
&, |	and, or	Say (1&1) + (0|1) → 2
&&	exclusive or	Say (1&&1) + (0|1) → 1

Operatoren und Priorität
(wann kann man Klammern weglassen?)

Operator	Prio	Beschreibung
\	8	logisches "nicht"
+	8	Präfix
-	8	Präfix
**	7	Potenzierung (Exp. ganzzahlig !)
*	6	Multiplikation
%	6	ganzzahlige Division
/	6	Division
//	6	Rest der Division
-	5	Subtraktion
+	5	Addition
Leerz.	4	Verkettung mit Leerzeichen
||	4	direkte Verkettung

Operator	Prio	Beschreibung
\==	3	absolut ungleich, alternativ auch ^== möglich
==	3	absolut gleich
\=	3	ungleich, alternativ auch ^= und <> bzw. >< möglich
=	3	gleich
>=	3	größer/gleich
>	3	größer
<=	3	kleiner/gleich
<	3	kleiner
&	2	logisches UND
|	1	logisches ODER
&&	1	logisches exklusiv ODER

Zeichenketten

Verbinden

Blank(s) verbinden mit Blank
|| verbinden ohne Blank

Beispiel:
```
Say 'A'    'B'    '[    C33]'
Say 'A' || 'B' || '[    C33]‘
```

```
Say Length('12 45')
Say Right('12345', 10)
Say Translate('12345', '+-', '43')
Say Word('AB CD XY', 2)
Say Reverse('12345')
Say Strip('  123  45  ')
Say Wordpos('XY', 'AB CD XY')
```

Funktionen für Zeichenketten

Length()	Länge der Zeichenkette
Words()	liefert Anzahl der Wörter
Word()	liefert ein einzelnes Wort
Left(),Right()	liefert letzte/erste n Zeichen
Strip()	entfernt Leerzeichen
SubStr()	liefert Teilzeichenkette
Delstr()	löscht Teilzeichenkette
Delword()	löscht Wörter
Pos()	liefert Buchstaben-Position
Lastpos()	liefert letzte Position
Wordpos()	liefert Wort-Position
Reverse()	dreht Zeichenkette um
Lower(),Upper()	klar
Translate()	tauscht Zeichen aus

Ende 1. Teil

Beispiel-REXX #1 …

```
/*********************************************************************/
/*  Find Payments:                                                   */
/*     Reads accounts lines one by one, and displays overdue         */
/*     payments (lines containing the phrase PAYMENT_OVERDUE).       */
/*********************************************************************/

arg filein                                  /* Read the input file name*/

do while lines(filein) > 0                  /* Do while a line to read */

   input_line = linein(filein)              /* Read an input line      */

   if pos('PAYMENT_OVERDUE',input_line) > 0 then

      say 'Found it:' input_line            /* Write line if $ overdue */

end
```

Fehlersuche mit *TRACE*

TRACE aktiviert die Ablaufüberwachung

Option	**Bedeutung**
ALL	'A': Anzeigen aller Klauseln vor ihrer Ausführung.
COMMANDS	'C': Anzeige aller Kommandos und Returncodes <> 0
ERROR	'E': zeigt Kommandos an, die Probleme bereiten
FAILURE	'F': zeigt Kommandos an, die nicht ausgeführt werden
INTERMEDIATE	'I': Zeigt alle Klauseln vor und bei der Berechnung an
LABEL	'L': Zeigt alle Sprunglabel an, die durchlaufen werden
OFF	'O': Schaltet *TRACE* ab.
RESULTS	'R': Zeigt alle Klauseln an und liefert die Ergebnisse

Möchte man den Programmablauf in Einzelschritten durchführen, so muss man der Option ein ?-Zeichen voranstellen:

```
TRACE '?i'
```

Sprungbefehl, Fehlerbehandlung

In REXX heißt der GOTO-Befehl einfach „SIGNAL“

```
SIGNAL Zielpunkt
...
Zielpunkt:
...
```

ON/OFF	Bedeutung
Error	schwerer Fehler aufgetreten
Failure	leichter Fehler aufgetreten
Halt	Programmabbruch (Strg+C)
NotReady	I/O-Fehler aufgetreten
NoValue	Variable nicht initialisiert
Syntax	Syntaxfehler festgestellt

```
SIGNAL ON SYNTAX
SIGNAL ON NoValue
SAY 'Los geht''s'
Parse Pull t1
SAY t1 a
EXIT 0
/*Fehlerbehandlung*/
SYNTAX:
SAY 'Syntax Zeile' sigl
EXIT 1
Novalue:
SAY Sourceline(sigl)
Say 'V=' Condition('D')
EXIT 2
```

Funktion, Unterprogramm

Im Prinzip gleich,
Unterschied:

```
Call myTIME 'Start'
Say 'Ergebnis=' result
Say myTime('Mitte', 2)
Exit 0
```

```
/*Unterprogramm Funktion*/
mytime:
Parse Arg text, opt
Select
  When opt='' Then a=Time()
  When opt=2
       Then a=Time('S')
  Otherwise a = '?'
End
Return text a
```

REXX liefert diese Funktionen mit

Rot = „nicht auf ‚z‘ verfügbar“, grün = ??

ABBREV()
ABS()
ADDRESS()
ARG()
B2X()
BEEP()
BITAND()
BITOR()
BITXOR()
C2D()
C2X()
CENTER()
CHANGESTR()
CHARIN()
CHAROUT()
CHARS()
COMPARE()
COPIES()
COUNTSR()
D2C()
D2X()
DATATYPE()
DATE()
DELSTR()
DELWORD()
DIGITS()
DIRECTORY()
ENDLOCAL()
ERRORTEXT()
FILESPEC()
FORM()
FORMAT()
FUZZ()
INSERT()
LASTPOS()
LEFT()
LENGTH()
LINEIN()
LINEOUT()
LINES()
MAX()
MIN()
OVERLAY()
POS()
QUEUED()
RANDOM()
REVERSE()
RIGHT()
SETLOCAL()
SIGN()
SOURCELINE()
SPACE()
STREAM()
STRIP()
SUBSTR()
SUBWORD()
SYMBOL()
TIME()
TRACE()
TRANSLATE()
TRUNC()
VALUE()
VAR()
VERIFY()
WORD()
WORDINDEX()
WORDLENGTH()
WORDPOS()
WORDS()
X2B()
X2C()
XRANGE()

REXX liefert „externe Funktionen“ mit

In ooREXX aktiviert durch:

```
If RxFuncQuery("SysLoadFuncs") Then Do
 CALL RxFuncAdd "SysLoadFuncs","RexxUtil","SysLoadFuncs"
 CALL SysLoadFuncs
End
```

In Windows (und zu 90% in Linux-Unix..) sind dann verfügbar:

RxMessageBox()	SysFileSystemType()	SysSetPriority()
SysCls()	**SysFileTree()**	SysShutdownSystem()
SysCurPos()	SysMkDir()	**SysSleep()**
SysCurState()	SysOpenEventSem()	SysSwitchSession()
SysDriveInfo()	SysQueryRexxMacro()	SysTempFileName()
SysDriveMap()	SysQuerySwitchList()	SysTextScreenRead()
SysElapsedTime()	SysRmDir()	SysWaitForShell()
SysFileDelete()	SysSaveRexxMacroSpace()	SysWaitNamedPipe()
SysFileSearch()	SysSearchPath()	SysWildCard(), …

PARSE Schlüsselwort

Zeichenfolgen (“Strings”) verarbeiten

abc + abc → abcabc

Verketten

...mit oder ohne (||) Blank

abcdef

Trennen

abc def

mit PARSE

abcabc

Zergliedern

PARSE analysiert die Zeichenfolge und trennt sie

abcdef

Muster erkennen

Find “def”

z.B. mit pos() oder lastpos()

PARSE: alternative Schablonen

trennt in Wörter

Separate **by** **words**

trennt nach Muster

Separate **,** **using** **,** **commas**

trennt nach Spalten

abc abc abc

Spalten: **1** **5** **9**

PARSE – trennt nach Worten

```
/*          1...+....1....+....2....+....3....+.. */
mystring= 'This is    a blank-delimited string  '
parse var mystring w1 w2 w3 w4 w5 . -oder-
parse value 'This is    a blank-delimited string  ',
    with w1 w2 w3 w4 w5 .
parse var mystring x1 x2 . x4  x5
```

ergibt:

w1	=>	'This'
w2	=>	'is'
w3	=>	'a'
w4	=>	'blank-delimited'
w5	=>	'string'

x1	=>	'This'
x2	=>	'is'
x3	=>	(nicht definiert)
x4	=>	'blank-delimited'
x5	=>	'string ‘ <- Achtung

- der ‘.‘ oben in den Schablonen funktioniert wie ein ‘Dummy‘
- Wenn am Ende der Schablone der ‘.‘ fehlt, bekommt die letzte Variable den Rest mit

PARSE – trennt nach Muster

```
Tmp = time()                  /* z.B. tmp = '11:33:20' */
Parse var tmp hours ':' mins ':' secs       -oder-
Parse value time() with hours ':' mins ':' secs
-oder-   dp = ':'
Parse var tmp hours (dp) mins (dp) secs
```

Was steht in w1 bzw. w2 ?

```
/*          1...+....1....+....2....+....3....+..*/
mystring= 'This is    a blank-delimited string  '
parse var mystring  w1 '-' w2 .       -oder-
Trenn='-'
Parse var mystring w1 (trenn) w2 .
```

PARSE – trennt nach Spalten

```
tmp = time()
dp = ':'
Parse var tmp hh (dp) mm (dp) ss
-oder-
Parse var tmp . 3 dp +1 . 1 hh (dp) mm (dp) ss


        ....+....1....+....2....+....3....+....4
string='Saywer          Tom               Rexx Guru'
Parse var string 1 lname +15 fname +15 title +9 .
-oder-
Parse var string 1 lname 16 fname 31 title 40 .
Say lname
Say fname
say title
```

PARSE – Sonderformen

```
Call mystr 'test', 2, 10
Exit
mystr: procedure
PARSE arg    str,startpos,    strlen
Say str '/' startpos '/' strlen
Return
```

```
PARSE source opsys . exfn
Say opsys `/` exfn
/*WindowsNT # test.rex*/
```

```
/*liest von Tastatur*/
Say 'Nachn, Vorname'
Parse Pull nn ',' vn
Say strip(vn) nn
```

```
PARSE version a b c
Say a'/' b'/' c'/'
>REXX-ooRexx_4.2.0(MT)_32-bit/ 6.04/ 22 Feb 2014/
```

REXX Datenstrukturen

Datenstrukturen werden durch Variablen mit Punkt „.“ gebildet

```
Zeile. = ''    /*initialisiert 'Stamm'*/
zeile.1 = 'erste'
zeile.3 = 'dritte'
zeile.0 = 3     /*Konvention: Anzahl in .0*/
Do ii = 1 To 3
  Say ii zeile.ii
end
Exit
```

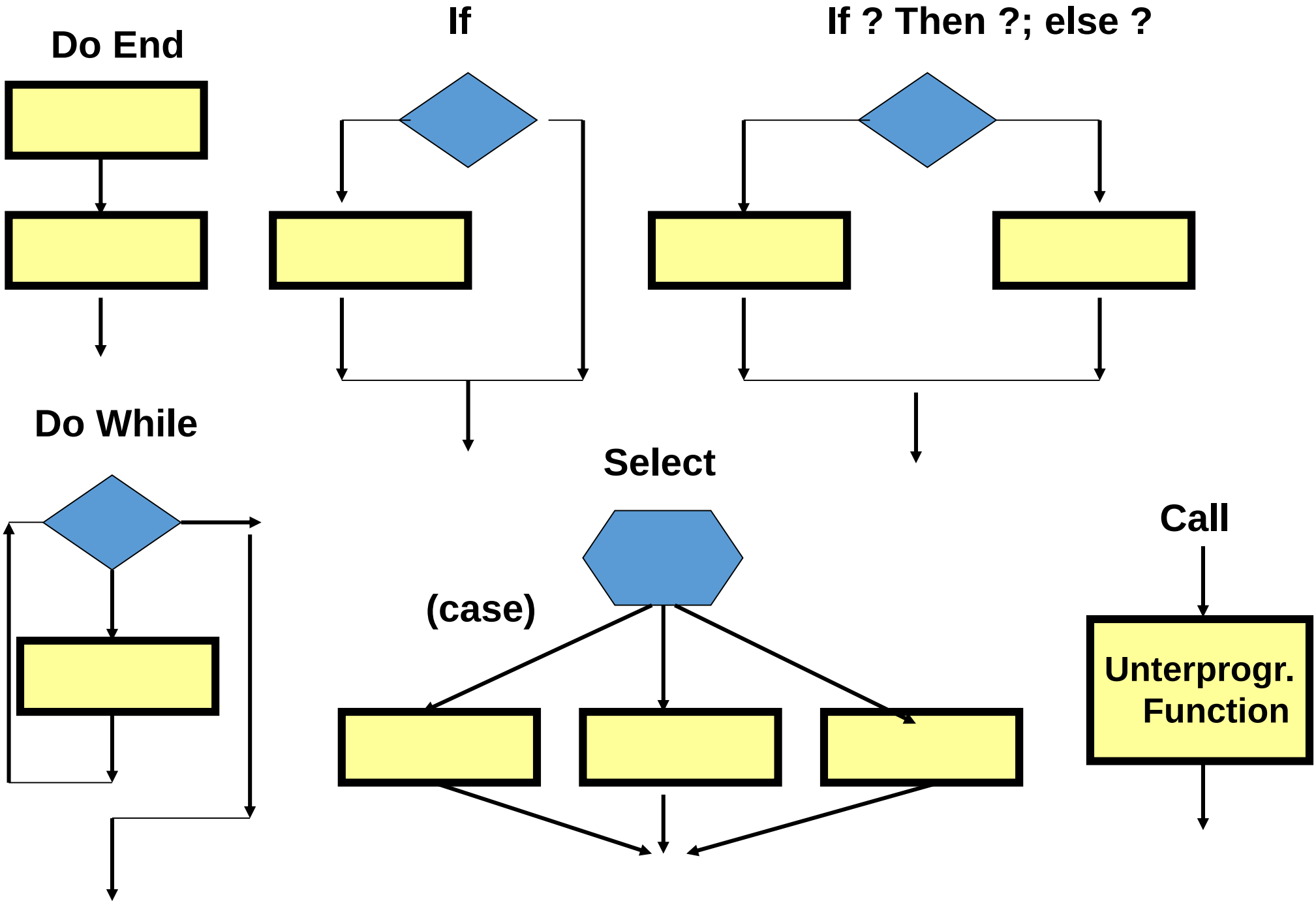
Man kann „zeile.“ beliebig erweitern

```
zeile.?font.1 = 'ARIAL'
zeile.11.aa.44.1.5 = 4711
```

Man kann „zeile.“ dann auch an Unterprogramme übergeben ...

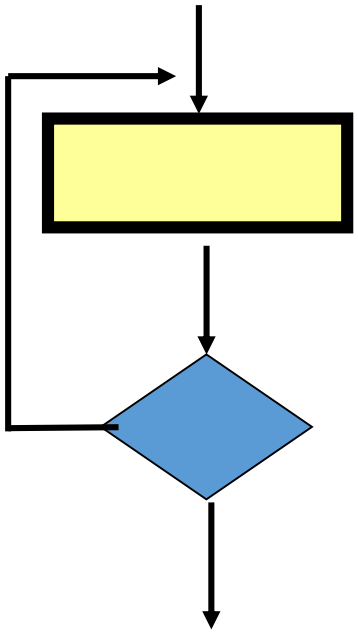
Strukturierung in REXX

strukturierende Instruktionen

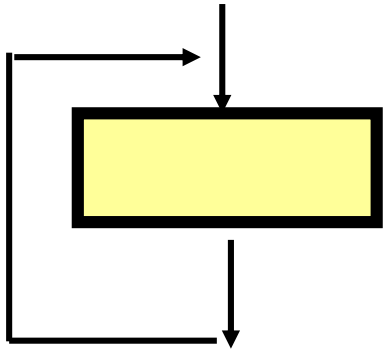


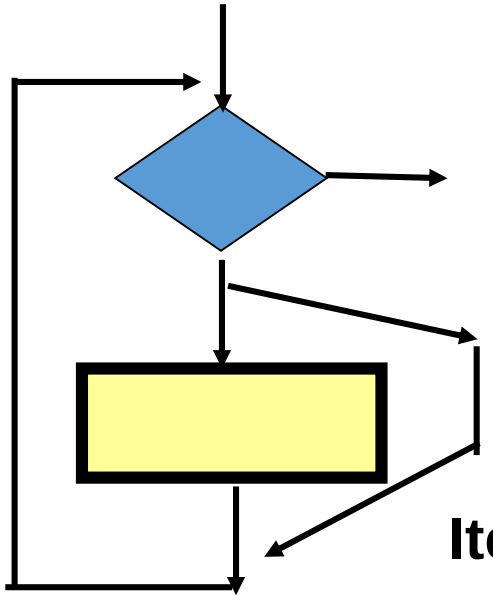
nicht-strukturierende Instruktionen

Do Until

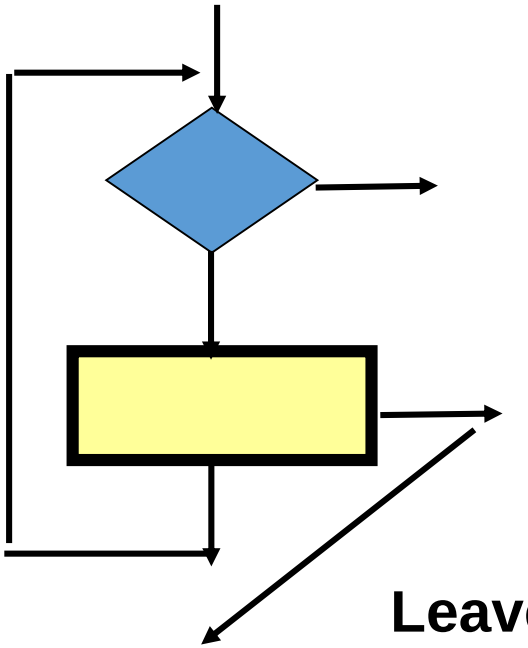


Do Forever

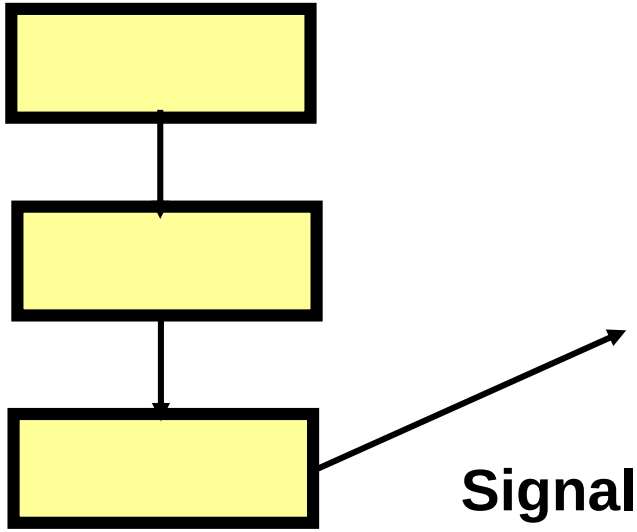




Iterate



Leave



Signal

(goto)

Beispiel REXX # 2... (part I)

```
/*  Find Books:                                                    */
/*     This program illustrates how arrays may be of any dimension */
/*     in retrieving book titles based on their keyword weightings. */

keyword. = ''          /* Initialize both arrays to all null strings */
title.   = ''

/* The array of keywords to search for among the book descriptors    */

keyword.1 = 'earth'    ;   keyword.2 = 'computers'
keyword.3 = 'life'     ;   keyword.4 = 'environment'

/* The array of book titles, each having several descriptors         */

title.1 = 'Saving Planet Earth'
   title.1.1 = 'earth'
   title.1.2 = 'environment'
   title.1.3 = 'life'
title.2 = 'Computer Lifeforms'
   title.2.1 = 'life'
   title.2.2 = 'computers'
   title.2.3 = 'intelligence'
title.3 = 'Algorithmic Insanity'
   title.3.1 = 'computers'
   title.3.2 = 'algorithms'
   title.3.3 = 'programming'
```

Beispiel REXX # 2... (part II)

```
arg weight      /* Get number keyword matches required for retrieval */

say 'For weight of' weight 'retrieved titles are:'  /* Output header */

do j = 1  while title.j <> ''                  /* Look at each book    */
   count = 0

   do k = 1  while keyword.k <> ''             /* Inspect its keywords */

      do l = 1  while title.j.l <> ''          /* Compute its weight   */
         if  keyword.k = title.j.l  then count = count + 1
      end

   end

   if count >= weight then   /* Display titles matching the criteria */
      say title.j
end
```

Sonstiges

I/O-Alternativen in ooREXX

Line-oriented

Process one line at a time

Character-oriented

Process n characters

linein
lineout
lines

charin
charout
chars

- **Keine Binärdaten wie TIFF, JPEG, PDF, EXE…**

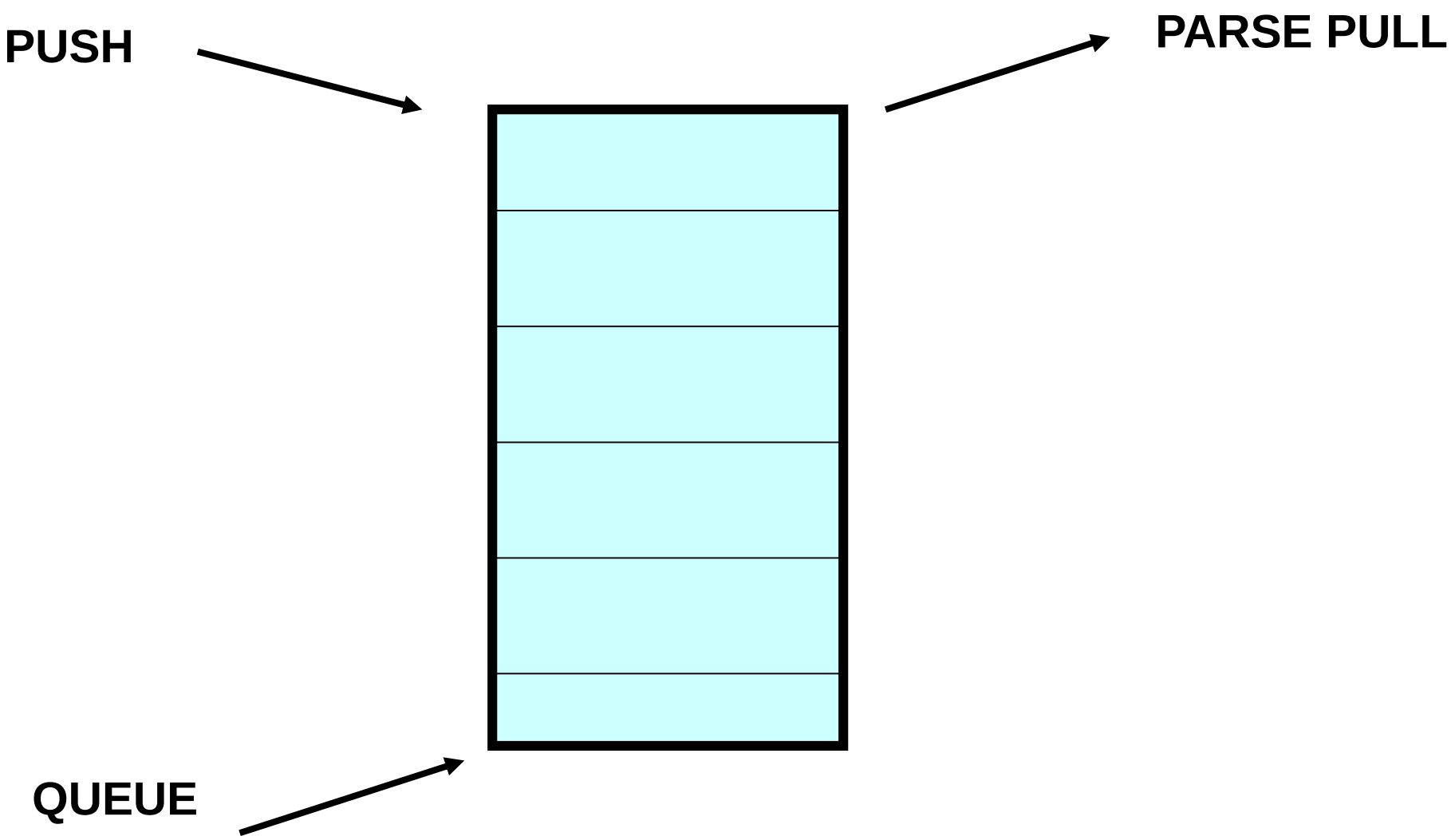
I/O Alternativen in ooREXX

```
Call Stream filename [, 'C' , 'Kommando']
If result<>'READY' Then ...  /*Ergebnis*/
```

Kommando	**Bedeutung**
OPEN	Datenstrom öffnen für Lesen und Schreiben
OPEN READ	- Nur zum Lesen öffnen
OPEN WRITE	- Nur zum Schreiben öffnen
CLOSE	Datenstrom schließen
SEEK	Positionierung des Schreib-Lesezeigers
QUERY EXISTS	Liefert Pfad zurück
QUERY SIZE	Liefert die aktuelle Größe in Bytes
QUERY DATETIME	Liefert die Zeitmarke der Datei im Verzeichnis

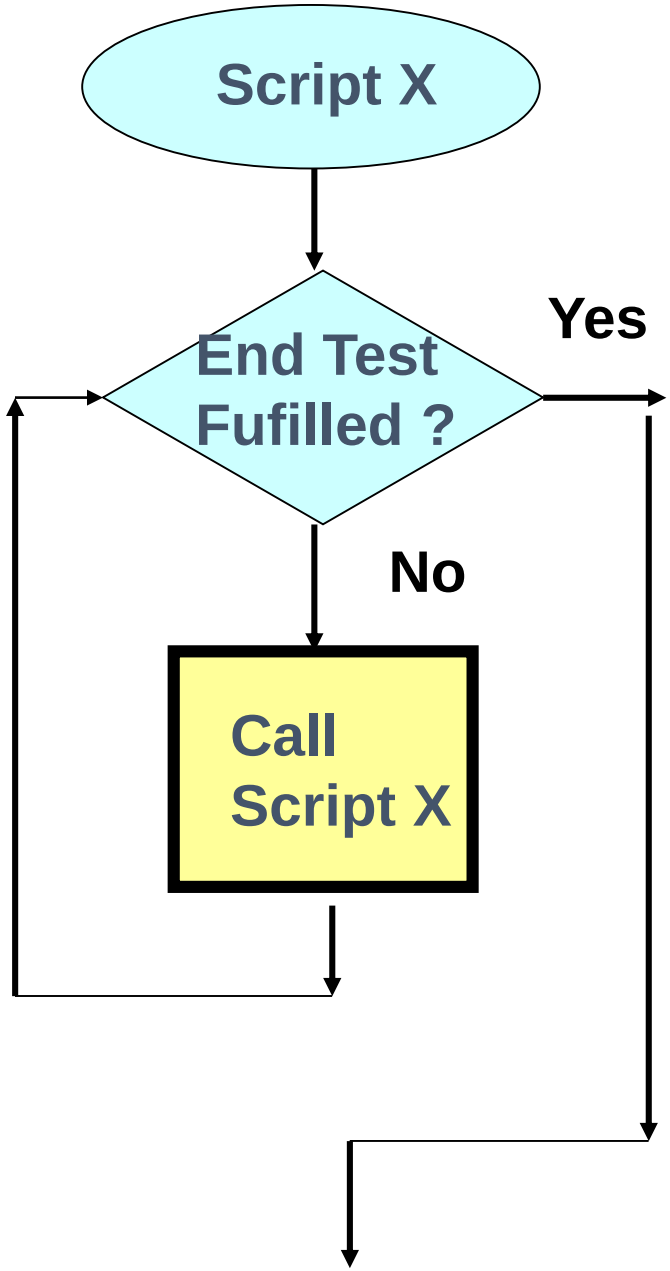
Ergebnis	**Bedeutung**
ERROR	Allgemeiner Fehler beim Zugriff mit ‚OPEN' oder ‚CLOSE'
READY	Der Datenstrom steht bereit
Size ! Date	+ time ! Pfad kommen bei ‚QUERY ...'

The *Stack* is both a *Stack* and a *Queue*



REXX's *Stack* is a generalized communications mechanism

REXX Supports Recursion



Beispiel aus der Mathematik
5! = 5*4*3*2*1 = 120

```
Numeric digits 30
Say Fakt(5)
Exit

Fakt:
Parse Arg num
Say num
if num = 2
 Then Return num*2
 Else Return num*Fakt(num-1)
```