



Procedural and Object-oriented Programming 2

Statement, Routine (Procedure, Function), "Stem"-Variable



Labels

- Identifier, followed by a colon (:)
- Serves as a target for an internal routine
 - **CALL**-statements (invoking procedures)
 - Function invocations
 - **SIGNAL**-statements (like a **GOTO** instruction in other languages)
 - Exception handling (**SIGNAL ON ERROR** resp. **CALL ON ERROR**)

Example

```
DO i = 1 TO 3
  SAY "Oho!" i
  IF i = 1 THEN SIGNAL fin
END
fin : SAY "C'est la fin!"
```

Output:

```
Oho! 1
C'est la fin!
```



Internal Routine, 1

- Grouping of statements which repeatedly get executed by different parts in a program
- Start with a label
- Invocation
 - **CALL** *label*
 - Statements after the label get executed
 - The **RETURN**-statement returns control (to the statement immediately following the **CALL**-statement)
- *A routine may also be called "procedure"*



Internal Routine, 2

```
/* A Rexx-Programm with an internal routine named 'TIMESTAMP' */  
CALL TimeStamp      /* call a subroutine */  
CALL SysSleep 5     /* sleep 5 seconds */  
CALL TimeStamp      /* call a subroutine */  
EXIT                /* exit program */  
  
TimeStamp :         /* label */  
    SAY "It is rather late ..."  
    RETURN
```

Output:

```
It is rather late ...  
It is rather late ...
```



Functions, 1

- Routines that return a value ("function value") to the caller via the **RETURN**-statement

- Invocation

- Variant 1

- Invocation: the name (label), immediately followed by a left and right parenthesis
 - The return value ("function value") replaces the invocation

```
today=DATE()
```

- Variant 2

- Invocation: **CALL** followed by the name (label) of the routine
 - Interpreter stores the return value in the special variable **RESULT**

```
call DATE
```

```
today = result
```



Functions, 2

```
/* A Rexx-Programm with an internal routine named 'TIMESTAMP' */  
CALL TimeStamp      /* call a subroutine */  
CALL SysSleep 5     /* sleep 5 seconds */  
CALL TimeStamp      /* call a subroutine */  
EXIT                /* exit program */  
  
TimeStamp :          /* label */  
    SAY "It is rather late ..."  
    RETURN
```

Output:

```
It is rather late ...  
It is rather late ...
```



Rexx Variables Used By Rexx

- The Rexx interpreter may set the value of the following variables
 - **RESULT**
 - If a routine gets invoked with **CALL** and returns a value with its **RETURN** statement then the Rexx interpreter stores that returned value in a variable named **RESULT**
 - **RC**
 - Stores the "*return code*" that the external command returned: a value of **0** (zero) indicates that the external command completed without any errors or failures
 - **SIGL**
 - Stores the line number which contains the **CALL** or **SIGNAL** statement that caused the jump to the routine ("*signal line*")



All Built-in Functions (BIF) of Rexx

- Rexx provides the following built-in functions ("BIF"), some are specific to **strings**, some to **filenames** ("*stream*"):

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()

ERRORTXT()

FILESPEC()

FORM()

FORMAT()

FUZZ()

GC()

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()



External Rexx Function Packages

- Standardised Interfaces to and from Rexx
 - Loading an external Rexx function package from Rexx, an example

```
IF RxFuncQuery("SysLoadFuncs") THEN DO
    CALL RxFuncAdd "SysLoadFuncs", "RexxUtil", "SysLoadFuncs"
    CALL SysLoadFuncs                /* no quotes! */
END
```

- Function packages, which supply new functions to Rexx that are not part of the language, an example
 - Mark Hessling's "RexxSQL" – direct access to the most important relational database management systems (DB2, Oracle, SQL-Server, MySQL, etc.)
 - [RexxSQL](#)



Function Package "RexxUtil" (Excerpt)

- Contains more than 60 useful (operating system dependent) functions, some of which are only available for **Windows** or **Unix** operating systems

...

RxMessageBox

RxWinExec

SysAddRexxMacro

SysBootDrive

SysClearRexxMacroSpace

SysCls

SysCreatePipe

SysCurPos

SysCurState

SysDriveInfo

SysDriveMap

SysDropRexxMacro

SysDumpVariables

SysFileCopy

SysFileDelete

SysFileExists

SysFileMove

SysFileSearch

SysFileSystemType

SysFileTree

SysFork

SysFormatMessage

SysFromUnicode

SysGetErrorText

SysGetFileDateTime

SysGetKey

SysGetMessage

SysGetMessageX

SysGetShortPathName

SysIni

SysIsFile

SysIsFileCompressed

SysIsFileDirectory

SysIsFileEncrypted

SysIsFileLink

SysIsFileNotContentIndexed

SysIsFileOffline

SysIsFileSparse

...



Searching for Routines, 1

- Searching order for routines
 1. Internal routines (in the program itself)
 2. Routines defined as directives (in the program itself)
 3. The built-in Rexx routines ("built-in functions (BIF)")
 4. External routines (e.g. Rexx programs)
- It is possible to use the names of the language built-in functions
 - Overlay the respective routines
 - The overlayed built-in function can always be invoked by enclosing its uppercased name in quotes!



Searching for Routines, 2

```
1 SAY date()      /* invoke self programmed function below */
2 SAY "DATE"()    /* invoke the Rexx built-in function      */
3 EXIT
4 DATE :          /* "DATE" is in effect a Rexx function ! */
5     RETURN "Date(), self programmed!"
```

Output (if run on 2059-05-20):

```
1 Date(), self programmed!
2 20 May 2059
```



Scopes, 1

- Define which variables and labels are seen in which part of a Rexx program
 - By default all variables in a program are *globally visible/accessible*, they belong to the *same scope*
 - Labels in a program are always *global*
- If the keyword instruction **PROCEDURE** follows a label, then a new (*local*) *scope* will be created for it
 - Should there be a need to access variables outside a *local scope*, then one must use the **EXPOSE** subkeyword of the **PROCEDURE**-statement denoting those variable names



Scopes, 2

```
1 a = 1
2 b = 2
3 SAY "a:" a "b:" b
4 CALL calc
5 SAY "a:" a "b:" b
6 EXIT
7 calc :
8     a = a * 2
9     b = b * 3 / 4
10    RETURN
```

Output:

```
1 a: 1 b: 2
2 a: 2 b: 1.5
```



Scopes, 3

```
1 a = 1
2 b = 2
3 SAY "a:" a "b:" b
4 CALL calc
5 SAY "a:" a "b:" b
6 EXIT
7 calc: PROCEDURE /* no access to global "a" und "b" ! */
8     a = 5        /* hence, "a" must be defined locally */
9     b = 6        /* hence, "b" must be defined locally */
10    a = a * 2
11    b = b * 3 / 4
12    RETURN
```

Output:

```
1 a: 1 b: 2
2 a: 1 b: 2
```



Scopes, 4

```
1 a = 1
2 b = 2
3 SAY "a:" a "b:" b
4 CALL calc
5 SAY "a:" a "b:" b
6 EXIT
7 calc: PROCEDURE EXPOSE b /* no access to global "a"; "b" now accessilbe ! */
8     a = 5                /* hence, "a" must be defined locally */
9     a = a * 2
10    b = b * 3 / 4
11    RETURN
```

Output:

```
1 a: 1 b: 2
2 a: 1 b: 1.5
```



"Stem" Variable (Associative Arrays), 1

- The *identifier* contains one or more dots
- The sequence of characters from the beginning up to and including the first dot is called *stem* (the remainder is called *tail*)

```
1  a.n                = "aha"
2  a.OnE              = 1
3  a.1                = "Richard"
4  Austria.Tyrol      = 750000
5  Austria.Tyrol.Innsbruck = 135000
6  SAY a.1 a.n a.OnE
7  SAY Austria.Tyrol
```

Output:

```
Richard aha 1
750000
```



"Stem" Variable (Associative Arrays), 2

- Some RexxUtil functions (e.g. `SysFileTree()`) use a convention, which mandates that after the dot only integer numbers be used
- `stem.0` – stores the total number of "elements"; this allows iterating over all stem entries starting with `1` going up to the number stored in the variable `stem.0`

```
1 file.1 = "max.doc"
2 file.2 = "moritz.doc"
3 file.0 = 2          /* maximum number of "elements"          */
4 DO i=1 TO file.0
5     SAY file.i      /* "i" in this context is also called "stem index" */
6 END
```

Output:

```
max.doc
moritz.doc
```



PARSE by Blank, 1

- **PARSE** statements allow parsing a string and assigning (parts of it) to Rexx variables in one step

```
1 text_=" _Stiegler_ _Seppl_ _Stumm_ _Zillertal/Tirol"  
2 PARSE VAR text_ famName_ firstName_ rest  
3 SAY_ famName  
4 SAY_ firstName  
5 SAY_ rest
```

Output:

```
Stiegler  
Seppl  
_Stumm_ _Zillertal/Tirol
```



PARSE by Blank, 2

```

1 ruler = COPIES("1234+6789|", 5)
2 text = "  Stiegler   Seppl   Stumm       Zillertal/Tirol"
3 PARSE VAR text famName firstName rest
4 SAY ruler; SAY text; SAY
5 SAY pp(famName); SAY pp(firstName)
6 SAY pp(ruler); SAY pp(rest)
7 EXIT
8 PP: RETURN "[" || ARG(1) || "]" " -- "pretty_print";)

```

Output:

```

1234+6789|1234+6789|1234+6789|1234+6789|1234+6789|
  Stiegler   Seppl   Stumm       Zillertal/Tirol

[Stiegler]
[Seppl]
[1234+6789|1234+6789|1234+6789|1234+6789|1234+6789|]
[  Stumm       Zillertal/Tirol]

```



PARSE by Pattern, 3

```

1  /*_____10_____20_____30_____40
2  _____1234+6789|1234+6789|1234+6789|1234+6789|_*_/
3  text = "____Ruaniger____Annelle_____Stumm____Zillertal____/____Tirol____"
4  PARSE VAR text before "/" after
5  SAY pp(before)
6  SAY pp(after)
7  EXIT
8  PP: RETURN "[" || ARG(1) || "]" " -- "pretty_print";)

```

Output:

```

[____Ruaniger____Annelle_____Stumm____Zillertal____]
[____Tirol____]

```



PARSE by Pattern, 4

```

1 pattern_ = "/"
2          /*          10          20          30          40
3          1234+6789|1234+6789|1234+6789|1234+6789|_*/
4 text_ = "  Ruaniger_Annelle          Stumm    Zillertal_/_Tirol_"
5 PARSE VAR text_ before_(pattern) after
6 SAY_pp(before)
7 SAY_pp(after)
8 EXIT
9 PP_: RETURN "[" || ARG(1) || "]" -- "pretty_print";)

```

Output:

```

[  Ruaniger_Annelle          Stumm    Zillertal_]
[_Tirol_]

```



PARSE by Position, Length, Blank, 5

```
1      /*          10          20          30          40
2      1234+6789/1234+6789/1234+6789/1234+6789/ */
3  text = "  Ruaniger Annelle          Stumm  Zillertal / Tirol  "
4  PARSE VAR text 3 famName +8 12 firstName city .
5  SAY pp(famName)
6  SAY pp(firstName)
7  SAY pp(city)
8  EXIT
9  PP : RETURN "[" || ARG(1) || "]" -- "pretty print" ;)
```

Output:

```
[Ruaniger]
[Annelle]
[Stumm]
```



PARSE, 6

```
1 text = "Sattler;Cilli;Stumm;Zillertal/Tirol"
2 PARSE VAR text famName ";" firstName ";" city
3 SAY pp(famName)
4 SAY pp(firstName)
5 SAY pp(city)
6 EXIT
7 PP : RETURN "[" || ARG(1) || "]" -- "pretty print" ;)
```

Output:

```
[Sattler]
[Cilli]
[Stumm;Zillertal/Tirol]
```



PARSE, 7 (First Character Is Encoding Character)

```
1 text = ";Sattler;Cilli;Stumm;Zillertal/Tirol"
2 PARSE VAR text 1 a +1 famName (a) firstName (a) city (a) .
3 SAY pp(famName)
4 SAY pp(firstName)
5 SAY pp(city)
6 EXIT
7 PP : RETURN "[" || ARG(1) || "]" -- "pretty print" ;)
```

Output:

```
[Sattler]
[Cilli]
[Stumm]
```



PARSE, 8 (VALUE ... WITH ...)

PARSE VALUE expression WITH pattern

- Makes it possible to immediately parse the result of an expression

Example:

```
PARSE VALUE date() WITH day month year
say "year:" year "month:" month "day:" day
```

Output (assuming `date()` returned the string `"20 May 2059"`):

```
year: 2059 month: May day: 20
```



PARSE, 9 (Input from Keyboard ("STDIN"))

- **PARSE PULL** statements allow parsing a string read from the keyboard ("*stdin*") and assigning (parts of it) to Rexx variables in one step

```
1 SAY "1. What is your name?" /* Keyboard input: "Max" */
2 PARSE PULL name
3 SAY "Your name is:" pp(name)
4 SAY "2. What is your name?" /* Keyboard input: "moritz" */
5 PULL name
6 SAY "Your name is:" pp(name)
7 EXIT
8 PP : RETURN "[" || ARG(1) || "]" -- "pretty print" ;)
```

Output (assuming `date()` returned the string "20 May 2059"):

```
1. What is your name?
Max
Your name is: [Max]
2. What is your name?
moritz
Your name is: [MORITZ]
```



PARSE, 10 (Retrieving Arguments PARSE ARG)

```
1 a = 1; b = 2
2 SAY "a:" a "b:" b
3 CALL calc a , b
4 SAY "a:" a "b:" b
5 EXIT
6 calc: PROCEDURE      /* caller's variables "a" and "b" not visible !      */
7     PARSE ARG a , b  /* fetch arguments and assign them to local variables */
8     SAY "calc: a:" a "b:" b
9     a = a * 2
10    b = b * 3 / 4
11    SAY "calc: a:" a "b:" b
12    RETURN
```

Output:

```
a: 1 b: 2
calc: a: 1 b: 2
calc: a: 2 b: 1.5
a: 1 b: 2
```

Contact

Modul University

Univ.-Prof. Mag. Dr. Rony G. Flatscher

Am Kahlenberg 1

1190 Wien/Vienna

Austria/Europe

www: www.modul.ac.at