

An Introduction to Procedural and Object-oriented Programming (Object Rexx) 2

Statement, Procedure, Function,
"Stem"-Variable

Prof. Rony G. Flatscher

Labels

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

```
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!
```

Procedures, 1

- Grouping of statements which repeatedly get executed by different parts in a program
- Starts with a label
- Invocation
 - **CALL** label
 - Statements in procedure are executed
 - The **RETURN**-statement returns control (to the statement immediately following the CALL-statement)

Procedures, 2

```
/* A Rexx-Programm ... */
CALL TimeStamp      /* call a subroutine */
CALL SysSleep 10    /* sleep 10 seconds */
CALL TimeStamp      /* call a subroutine */
EXIT                /* leave program */

TimeStamp :          /* label */
    SAY "It is rather late ..."
    RETURN
```

Output:

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

Functions, 1

- Procedures which return a value ("function value") to the caller via the **RETURN**-statement

- Invocation

- Variant 1

- Invocation: note the label, immediately followed by a round opening and closing bracket
 - The return value ("function value") replaces the invocation

```
today = DATE()
```

- Variant 2

- Invocation like procedure
 - Interpreter stores the return value in the variable **RESULT**

```
CALL DATE  
today = result
```

Functions, 2

```
/* A Rexx-Programm ... */
SAY TimeStamp() /* function call */
CALL SysSleep 10 /* sleep 10 seconds */
CALL TimeStamp /* procedure call */
SAY result /* show function value */
EXIT /* leave program */

TimeStamp : /* function label */
RETURN "It is rather late ..."
```

Output:

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

Special Rexx Variables

- After calling a procedure, a function or an external command, the Rexx runtime environment may set the following variables with values, that may have been returned
 - **RESULT**
Stores the function value, i.e. the value which is given with the **RETURN** statement
 - **RC**
"Return Code" of (external) commands
 - **SIGL**
"Signal Line" - number of the source code line, in which an exception (e.g. an error) occurred
[Rexx function **SourceLine(sigl)** returns the contents of the source code line, in which an exception occurred]

All Functions of the Language Rexx

- Rexx supplies the following functions, which are considered to be a part of the language:

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

External Rexx Function Packages

- Standardised Interfaces to and from Rexx
- Function packages, which supply new functions to Rexx that are not part of the language, e.g.
 - Direct access to the most important relational database management systems (DB2, Oracle, SQL-Server, MySQL, etc.)
 - E.g. Mark Hessling's "RexxSQL"
 - ftp- resp. TCP/IP socket programming
 - Loading of external Rexx function packages, e.g. of "RexxUtil" (usually gets distributed with Rexx):

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

Rexx Function Package "RexxUtil" (Excerpt)

- "RexxUtil" function package (a DLL)
 - Contains operating system dependent, "useful" functions
 - Appr. 90% of the functions available in all implementations
 - E.g. (excerpt from the Windows implementation):

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(), ...

Searching Procedures and Functions, 1

- Searching order for procedures/functions
 1. Internal procedures/functions that can be found in the program itself which invokes them
 2. The language builtin procedures/functions
 3. External procedures/functions (e.g. Rexx programs)
- It is possible to use the label names of the language builtin procedures/functions
 - Overlay the respective procedures/functions
 - The overlaid function can always be invoked by
 - ➔ enclosing the **uppercased** label in quotes!

Searching Procedures and Functions, 2

```
/* */
SAY date() /* invoke self programmed function below */
SAY "DATE"() /* invoke the Rexx builtin function */
EXIT

DATE : /* "DATE" is in effect a Rexx function ! */
      RETURN "Date(), self programmed!"
```

Output:

```
Date(), self programmed!
22 Oct 2016
```

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** keyword of the **PROCEDURE**-Statement denoting those variable names.

Scopes, 2

```
/* */  
a = 1  
b = 2  
SAY "a=" a "b=" b  
CALL calc  
SAY "a=" a "b=" b  
EXIT
```

```
calc :  
  a = a * 2  
  b = b * 3 / 4  
  RETURN
```

Output:

```
a= 1 b= 2  
a= 2 b= 1.5
```

Scopes, 3

```
/* */  
a = 1  
b = 2  
SAY "a=" a "b=" b  
CALL calc  
SAY "a=" a "b=" b  
EXIT
```

```
calc: PROCEDURE /* no access to global "a" und "b" ! */  
  a = 5          /* hence, variable "a" must be defined locally */  
  b = 6          /* hence, variable "b" must be defined locally */  
  a = a * 2  
  b = b * 3 / 4  
  RETURN
```

Output:

```
a= 1 b= 2  
a= 1 b= 2
```

Scopes, 4

```
/* */  
a = 1  
b = 2  
SAY "a=" a "b=" b  
CALL calc  
SAY "a=" a "b=" b  
EXIT
```

```
calc: PROCEDURE EXPOSE b /* no access to "a", but to "b" ! */  
  a = 5 /* hence, variable "a" must be defined locally */  
  a = a * 2  
  b = b * 3 / 4  
  RETURN
```

Output:

```
a= 1 b= 2  
a= 1 b= 1.5
```

"Stem" Variable (Associative Arrays), 1

■ "Stem" Variable

- Identifier contains one or more **dots**
- The sequence of characters from the beginning up to and including the first dot is called *stem*
- Examples:

```
a.n           = "aha"  
a.OnE        = 1  
a.1          = "Richard"  
Austria.Tyrol = 750000  
Austria.Tyrol.Innsbruck = 135000  
SAY a.1 a.n a.OnE  
SAY Austria.Tyrol
```

Output:

```
Richard aha 1  
750000
```

"Stem" Variable (Associative Arrays), 2

- Some functions from Rexx function packages (e.g. *SysFileTree()* in *RexxUtil*) use a convention, which mandates that after the dot only integer numbers be used

- stem.0

- Stores the total number of "elements" in the stem; this allows iterating over all stem entries starting with "1" and going up to and including the number stored in stem.0

```
file.1 = "max.doc"  
file.2 = "moritz.doc"  
file.0 = 2                /* maximum number of "elements" */  
DO i=1 TO file.0  
    SAY file.i           /* "i" is also called "index" */  
END
```

Output:

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

PARSE, 1

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

```
text = "  Stiegler  Seppl  Stumm  Zillertal/Tirol"  
PARSE VAR text famName firstName rest  
SAY famName  
SAY firstName  
SAY rest  
EXIT
```

Output:

```
Stiegler  
Seppl  
  Stumm  Zillertal/Tirol
```

PARSE, 2

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

```
lineal = COPIES("1234+6789|", 5)
text   = "  Stiegler  Seppl  Stumm      Zillertal/Tirol"
PARSE VAR text famName firstName rest
SAY lineal; SAY text ; SAY
SAY pp(famName); SAY pp(firstName)
SAY pp(lineal); SAY pp(rest)
EXIT
PP : RETURN "[" || ARG(1) || "]"
```

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, 3

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

```
          /*          10          20          30          40
          1234+6789|1234+6789|1234+6789|1234+6789| */
text = "  Ruaniger Annelle  Stumm  Zillertal / Tirol  "
PARSE VAR text before "/" after
SAY pp(before)
SAY pp(after)
EXIT
PP : RETURN "[" || ARG(1) || "]"
```

Output:

```
[  Ruaniger Annelle  Stumm  Zillertal ]
[ Tirol ]
```

PARSE, 4

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

```
pattern = "/"
/*          10          20          30          40
1234+6789|1234+6789|1234+6789|1234+6789| */
text = "  Ruaniger Annelle  Stumm  Zillertal / Tirol "
```

PARSE VAR text before **(pattern)** after

```
SAY pp(before)
SAY pp(after)
EXIT
PP : RETURN "[" || ARG(1) || "]"
```

Output:

```
[ Ruaniger Annelle  Stumm  Zillertal ]
[ Tirol ]
```

PARSE, 5

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

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

Output:

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

PARSE, 6

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

```
text = "Sattler;Cilli;Stumm;Zillertal/Tirol"  
PARSE VAR text famName ";" firstName ";" city  
SAY pp(famName)  
SAY pp(firstName)  
SAY pp(city)  
EXIT  
PP : RETURN "[" || ARG(1) || "]"
```

Output:

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

PARSE, 7

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

```
text = ";Sattler;Cilli;Stumm;Zillertal/Tirol"  
PARSE VAR text 1 a +1 famName (a) firstName (a) city (a) .  
SAY pp(famName)  
SAY pp(firstName)  
SAY pp(city)  
EXIT  
PP : RETURN "[" || ARG(1) || "]"
```

Output:

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

Input from "STDIN:" (Keyboard)

PARSE PULL, PULL

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

```
SAY "1. What is your name?" /* Keyboard input: "Max" */
PARSE PULL name
SAY "Your name is:" pp(name)
SAY "2. What is your name?" /* Keyboard input: "moritz" */
PULL name
SAY "Your name is:" pp(name)
EXIT
PP : RETURN "[" || ARG(1) || "]"
```

Output:

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

Retrieving Arguments

PARSE ARG

PARSE ARG statements allow to assign argument-values or parts of them to Rexx variables in one step

```
a = 1; b = 2
SAY "a=" a "b=" b
CALL calc a , b
SAY "a=" a "b=" b
EXIT
```

```
calc: PROCEDURE          /* caller's variables "a" and "b" not visible !*/
  PARSE ARG a , b
  SAY "calc: a=" a "b=" b
  a = a * 2
  b = b * 3 / 4
  SAY "calc: a=" a "b=" b
  RETURN
```

Output:

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