Abteilung für Wirtschaftsinformatik

Rexx Tutorial for Beginners, 1

Introduction, Overview, Statement, Procedure, Function

Prof. Rony G. Flatscher

Overview, 1

- Purpose
  - Basic concepts of the object-oriented paradigm
    - Standard application systems
  - Scripting language
    - Automation ("remote controlling") of applications
    - Automation of operating systems like MacOSX, Linux or Windows
  - Foils

    http://wi.wu-wien.ac.at/rgf/wu/lehre/autowin/material/foils/

    http://wi.wu-wien.ac.at/rgf/wu/lehre/autojava/material/foils/
  - Excercises

    http://wi.wu-wien.ac.at/rgf/wu/lehre/autowin/material/excercises/

Overview, 2

- Why Rexx? Why Object Rexx?
  - Simple syntax ("human-centric" language)
    - Easy and quick to learn
  - Powerful object-model
    - All important concepts of the OO-paradigm available
- Availability of Software

  http://www.ooRexx.org

History, 1
REXX

- 1979 - IBM (Mike F. Cowlishaw, IBM-Fellow)
  - Successor of a rather cryptic script language ("EXEC") on IBM mainframes
  - Goal: to create a "human-centric" language
  - Interactive (Interpreter)
  - REXX: Acronym for "REstructured eXtended eXecutor"
- Since 1987 IBM's "SAA" (System Application Architecture) "Procedural Language"
  - Strategic script language for all IBM platforms
  - Numerous commercial and open source versions of the language, available for practically all operating systems there are
- ANSI Rexx Standard in 1996
  - ANSI "Programming Language - REXX", X3.274-1996

History, 2
ooRexx (Open Object Rexx)

- Since the beginning of the 90ies
  - Going back on an initiative of the powerful IBM user interest group "SHARE" development of an object-oriented version of REXX started
- "Object-based REXX" a.k.a. "Object REXX"
  - Fully compatible with classic ("procedural") Rexx
  - Internally fully object-oriented
    - *All classic Rexx statements are transformed into object-oriented ones internally!*
  - Powerful object model (e.g. meta-classes, multiple inheritance)
  - Still a simple syntax
  - Availability
    - 1997 part of OS/2 Warp 4 (free) and free for Warp 3 (with SOM)
    - 1998 AIX (first evaluation version) and Linux (free)
    - 1998 for Windows 95 and Windows NT (with OLEAutomation/ActiveX)

History, 3
NetRexx

- 1996 development of "NetRexx" by the original author of Rexx, Mike F. Cowlishaw
  - Java in the "clothes" of Rexx
    - NetRexx-programs are translated into Java byte code
    - Simpler programming of the Java VM due to the simpler Rexx syntax
      - ~30% less Code (syntactical elements) than Java
    - Due to the Rexx syntax, easier to learn for the programming novice
  - IBM handed over source code to RexxLA
    - June, 8th, 2011 opensource released by RexxLA
    - Kick-off for new developments
- URLs for Rexx, Object Rexx, NetRexx

  http://www.RexxLA.org/

  http://www.ooRexx.org/

  http://www.NetRexx.org/

  news:comp.lang.rexx

A Taste of Rexx

```
say "Hello" world!    /* yields: Hello WORLD!                              */
say 1/3               /* yields: 0.333333333                              */
numeric digits 25     /* now use 25 significant digits in arithmetics     */
say "1"/3             /* yields: 0.3333333333333333333333333              */
"rm -rf *"            /* Unix: remove all files recursively               */
```

- "Everything is a string"
- Everything outside of quotes gets uppercased
- Arbitraryly precise decimal arithmetic
- ANSI Rexx rules served for defining the rules for other programming languages and is used for implementing decimal arithmetics in hardware
- Unknown statements are passed to environment

Basics
Minimal Rexx-Program

```
/* a comment */
SAY "Hello, my beloved world"
```

Output:

```
Hello, my beloved world
```

Basics
Notation of Program Text

- Upper or lowercase spelling irrelevant
  - All characters of a Rexx statement will be translated into uppercase and executed
  - Exception: Contents of a string remains unchanged
    - Strings are delimited by apostrophes (') or by quotes ("), e.g.

```
"Richard", 'Richard', "\{[]}\gulp!öäüß!{niX }"
```

- Multiple blank characters are reduced to one blank
  - Example

```
saY      "\{[]}\gulp!öäüß!{niX }"     reverse(    Abc )
```

becomes:

```
SAY "\{[]}\gulp!öäüß!{niX }" REVERSE( ABC )
```

Basics
Characters

- Characters outside of strings and comments must be from the following character set
  - Blank
  - **a** thru **z**
  - **A** thru **Z**
  - **0** thru **9**
  - Exclamation mark (**!**), backslash (****), question mark (**?**), equal sign (**=**), comma (**,**), dash/minus (**-**), plus (**+**), dot (**.**), Slash (**/**), parenthesis (**()**), square parentheses (**[]**), asterisk (*****), tilde (**~**), semicolon (**;**), colon (**:**) and underline (**_**)

Basics
Variables

- Variables allow storing, changing, and retrieving strings with the help of a discretionary name called an *identifier*

```
A = "Hello, my beloved world"
a="Hello, my beloved variable"
A    =      a          "- changed again."
say a
```

Output:

```
Hello, my beloved variable - changed again.
```

- Identifiers must begin with a letter, an exclamation mark, a question mark or an underline character, followed by one or more of these characters, digits, and dots.

Basics
Constants

- Constants never get their values changed
- It is possible to use literals which are string constants appearing verbatim in an expression
  - If one wishes to name constants, then there are two possibilities available
    - The constant value is assigned to a variable, the value of which never gets changed in the entire program, e.g.

```
pi = 3.14159
```

  - A constant directive in a class, e.g.

```
::class constants
::constant pi 3.14159
```

Basics
Comments

- Comments may be nested and are allowed to span multiple lines, e.g.

```
say 3  +    /* This /**/ is
    a            /* nested
  /* aha*/ comment*/ which spans
     multiple lines */ 4
```

Output:

```
7
```

- Line comments: at the end of a statement, comments follow after two consecutive dashes:

```
say 3  +  4 -- this yields "7"
```

Output:

```
7
```

Basics
Statements, 1

- Statements consist of all characters up to and including the semi-colon (;)
- There may an arbitrary number of statements on a line
- If the semi-colon is missing, then the end of a statement is assumed by the end of a line

```
/* Convention: A comment begins in 1. line, 1. column */
SAY "Hello, my dear world";
```

Output:

```
Hello, my dear world
```

Basics
Statements, 2

- Statements may span multiple lines, but you need to indicate this with the continuation character
  - Comma or Dash as the last character on the line

```
/* Convention: A comment begins in 1. line, 1. column */
SAY "Hello," -
    "my beloved world";
```

Output:

```
Hello, my beloved world
```

Basics
Block

- A block is a statement, which may comprise an arbitrary number of statements
- A block starts with the keyword **DO** and ends with **END**

```
DO;
   SAY "Hello," ;
   SAY "world"  ;
END;
```

```
  DO
    SAY "Hello,"
    SAY "world"
  END
```

Output:

```
Hello,
world
```

Basics
Comparisons (Test Expressions), 1

- Two values (constant, variable, results of function calls) can be compared with the following (Infix) operators (Result: 0=false or 1=true)

```
=          equal
<>    \=      unequal
<          smaller
<=         smaller than
>          greater
>=         greater than
```

- Negation of Boolean (0=false, 1=true) values

```
\          Negator
```

Basics
Comparisons (Test Expressions), 2

- Boolean values can be combined

```
&   "and" (true: if both arguments are true)

|   "or" (true: if either argument are true)

&& "exclusive or" (true: if one argument is true
        and the other is false)
```

- Boolean combinations can be evaluated in a specific order if enclosed in parentheses:

```
 0 & 1   | 1   Result: 1 (= true)
(0 & 1)  | 1   Result: 1 (= true)
 0 & (1  | 1)  Result: 0 (= false)
```

Basics
Comparisons (Test Expressions), 3

```
a=1
b=2
x="Anton"
y="  Anton   "

If a = 1 then ...        Result: 1 (= true)
If a = a then ...        Result: 1 (= true)

If a >= b then ...       Result: 0 (= false)

If x = y then ...        Result: 1 (= true)
If x == y then ...       Result: 0 (= false)

  a <= b & (a = 1 | b > a)Result: 1 (= true)
\(a <= b & (a = 1 | b > a))   Result: 0 (= false)
\a                  Result: 0 (= false)
```

Basics
Branch, 1

- A branch determines which statement (block) should be executed as a result of a comparison (of a Boolean value)
  - `IF test_expression=.true THEN statement;`
    - Example:

```
IF age < 19 THEN SAY "Young."
```

  - A branch can also determine what alternative statement (block) should be executed, in case the Boolean value is false
  - `IF test_expression=.true THEN statement; ELSE statement;`
    - Examples:

```
IF age < 19 THEN SAY "Young.";
            ELSE SAY "Old."
```

```
IF age < 1 THEN
DO
   SAY "Hello,"
   SAY "my beloved world"
END
```

Basics
Branch, 2

- **Multiple selections (SELECT)**

```
SELECT
   WHEN test_expression THEN statement;
   WHEN test_expression THEN statement;
   /* ... additional WHEN-statements */
   OTHERWISE statement;
END
```

Example:

```
SELECT
  WHEN age =   1 THEN SAY "Baby." ;
  WHEN age =   6 THEN SAY "Elementary school kid." ;
  WHEN age >= 10 THEN SAY "Big kid." ;
  OTHERWISE SAY "Unimportant." ;
END
```

Basics
Repetition, 1

- Principally a block can be executed repeatedly

```
DO 3
   SAY "Aua!"
   SAY "Oh!"
END
```

Output:

```
Aua!
Oh!
Aua!
Oh!
Aua!
Oh!
```

Basics
Repetition, 2

- Using a variable to control the number of repetitions

```
a = 3
...
DO a
   SAY "Aua!"; SAY "Oh!"
END
```

Output:

```
Aua!
Oh!
Aua!
Oh!
Aua!
Oh!
```

Basics
Repetition, 3

- Repetition using a control variable ("i" in this example)

```
DO i = 1 TO 3
   SAY "Aua!";SAY "Oh!" i
END
```

Output:

```
Aua!
Oh! 1
Aua!
Oh! 2
Aua!
Oh! 3
```

Basics
Repetition, 4

- Repetition using a control variable ("i" in this example)

```
DO i = 1 TO 3 BY 2
    SAY "Aua!";SAY "Oh!" i
END
```

Output:

```
Aua!
Oh! 1
Aua!
Oh! 3
```

Basics
Repetition, 5

- Repetition using a control variable ("i" in this example)

```
DO i = 3.1 TO 5.7 BY 2.1
   SAY "Aua!";SAY "Oh!" i
END
```

Output:

```
Aua!
Oh! 3.1
Aua!
Oh! 5.2
```

Basics
Repetition, 6

- Conditional repetition

```
i = 2
DO WHILE i < 3
    SAY "Aua!";SAY "Oha!" i
    i = i + 1
END
```

Output:

```
Aua!
Oha! 2
```

Basics
Repetition, 7

- Conditional repetition

```
i = 3
DO WHILE i < 3
    SAY "Aua!";SAY "Oha!" i
    i = i + 1
END
```

➔**No output, because block is not executed!**

Basics
Repetition, 8

- Conditional repetition

```
i = 3
DO UNTIL i > 1
    SAY "Aua!";SAY "Oha!" i
    i = i + 1
END
```

Output:

```
Aua!
Oha! 3
```

Basics
Execution, 1

```
/* */
 a = 3
 b = "4"
 say  a   b
 say  a b
 say  a ||b
 say  a + b
```

Output:

```
3 4
3 4
34
7
```

Basics
Execution, 2

```
/* */
 "del *.*"
```

or:

```
 /* */
 ADDRESS CMD "del *.*"
```

or:

```
 /* */
 a = "del *.*"
 a
```

or:

```
 /* */
 a = "del *.*"
 ADDRESS CMD a
```

Abteilung für Wirtschaftsinformatik

Rexx Tutorial for Beginners, 2

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

Prof. Rony G. Flatscher

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** 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!
```

Internal Routines, 1

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

Internal Routines, 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

- Routines that 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 routine 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()	ERRORTEXT()	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 for Routines, 1

- Searching order for routines
  1. Internal routines that can be found in the program itself which invokes them
  2. Routines def ned as directives in the program itself
  3. The language builtin routines
  4. External routines (e.g. Rexx programs)
- It is possible to use the label names of the language builtin routines
  - Overlay the respective routines
  - The overlayed routine can always be invoked by

  ➔ *enclosing the **uppercased** label in quotes!*

Searching for Routines, 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 2036
```

Scopes, 1

- Def ne 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
```