

Introduction to REXX

Rich Smrcina
10 July 2026

- References
- What REXX is
- REXX programming
 - Syntax
 - Style
 - Functions
 - Arguments
 - Parsing

REXX/VM User's Guide

- Phenomenal resource for learning REXX

REXX/VM Reference

- The paper version of HELP REXX MENU

IBM Training Digital Learning Platform

- <https://learn.ibm.com> - Free course for learning the basics of REXX

REXX Language Association

- <https://rexsla.org>

Open Object REXX

- <http://www.oorexx.org>

Regina REXX

- <https://regina-rexx.sourceforge.io>

- REXX (REstructured eXtended eXecutor)
- Interpreted Language
 - There is a compiler, but it is rarely used
- “Typeless”
 - Data is not a particular type such as binary, packed-decimal, etc
- Runs on many platforms
 - z/VM, z/OS, VSEn, Linux, Windows

From “A brief history of REXX”

The first specification for the language is dated 29 March 1979. This was written before any implementation was even designed, and it was circulated to a number of people for comment: this began the tradition of documentation before implementation that characterized the development of Rexx. This first specification included three sample programs written in Rexx to show how the language would look; those programs would seem familiar to today's Rexx programmers, although some details have changed.

- <https://www.rexxla.org/rexxlang/history/mfc/rexxhist.html>

Is it REXX or Rexx?

REXX programs must have a filetype of EXEC in CMS

```
XEDIT MYFILE EXEC A
```

REXX programs must ALWAYS begin with a comment

```
/* this is a comment */
```

```
MYFILE      EXEC      A1  V 255  Trunc=255 Size=2 Line=0 Col=1 Alt=7
====>
T...+...1...+...2...+...3...+...4...+...5...+...6...
00000 * * * Top of File * * *
00001 /* this is the comment for my rexx program */
00002
00003 * * * End of File * * *
```

Syntax

- REXX clauses are typically delimited by a new line

```
a = "cat"
```

```
b = "dog"
```

- They can also be delimited by a semicolon

```
a = "cat"; b = "dog";
```

Clauses can be:

Assignments

```
a = "Fish"
```

Labels

```
LABEL1:
```

Host Command

```
"QUERY DISK"
```

Null line

REXX Instructions

```
Signal On ERROR
```

Style

Discussions of style fall under coding standards and can vary

- Variables lowercase
- Labels uppercase
- Host commands uppercase enclosed in double quotes
- REXX clauses/keywords should have the first character capitalized

Examples

```
company = "Acme, Inc"
```

```
LABEL1:
```

```
"CP QUERY TIME"
```

```
Call LABEL1
```


Style - Indents

- Usually two characters (spaces)
- 'Do/End' and 'Select/End' have the same indentation
- Anything in the 'Do block' or 'Select block' should be indented
- Indent clauses for 'If', 'Else', 'When', 'Otherwise'

Examples

```
|...+....1....+....2....+....3....+....4....+....5...
00000 * * * Top of File * * *
00001 /* Simple Style */
00002   Parse Arg fname lname
00003   If fname <> '' Then
00004       Say 'Hello' fname '!'
00005   Else
00006       Do
00007           Say 'I expected to see your name'
00008           Say 'Try: MYEXEC firstname lastname'
00009       End
00010 Exit
00011 * * * End of File * * *
```

Variables

- Unassigned variables have a value of their name translated to uppercase
`name` → `NAME`
- An initialized variable has a value of what it was assigned to
`name = "Andrew"` → `Andrew`
- It is always good practice to initialize variables to avoid errors or confusion later
`name = ""`

Variables – Examples and a Quiz

```
      |...+.1...+.2...+.
000000 * * * Top of File * * *
000001 /* rexx comment */
000002
000003 name1 = "Sam"
000004 name2 = ""
000005 name3 = sam
000006
000007 Say "Name1 is:" name1
000008 Say "Name2 is:" name2
000009 Say "Name3 is:" name3
000010 Say "Name4 is:" name4
000011
000012 * * * End of File * * *
```

```
Name1 is:
Name2 is:
Name3 is:
Name4 is:
```

Quotes

- Quotes are your friend
- Use either single (') or double quotes (")
- Strings written without quotes are translated to upper case
 - Why? Or worse, the string will not be what you intended

Say "That's his cat" → That's his cat

Say That's his cat → THAT'S HIS CAT

Strings

- Use double vertical bars to concatenate values

`say "C" || "A" || "T" → CAT`

- Concatenated hex values must ALWAYS have a space after unless concatenation (||) is used

`say "c3"x → C`

`say "c3"x"AT" → c3xAT`

`say "c3"x || "AT" → CAT`

Compound variables

- Also known as stems or arrays

```
day.1 = 'Monday'
```

```
day.2 = 'Tuesday'
```

- Variables are substituted right to left

```
num = 1
```

```
Say day.num → 'Monday'
```

- Initialize the entire compound variable

```
day. = '' /* sets all potential values to null */
```

Compound variables

- Stem references need to be numbers or variables
 - `day.today = "Monday" /* today var can be set, or not */`
 - `day.1 = "Monday"`
- A reference to a function will fail

```
day.datenum() = "Monday"
```

Running that code would likely produce:

```
DMSOPN062E Invalid character . in fileid RXDAY.DA MODULE
```

```
6 +++ day.datenum() = "Monday"
```

```
DMSREX475E Error 40 running TRY EXEC, line 6: Incorrect call to  
routine
```

1. A REXX program must always start with what?
2. What is the value of an unassigned variable?

1. A REXX program must always start with what?

A comment

2. What is the value of an unassigned variable?

The string in upper case

REXX Instructions

- Say
 - Write the subsequent values/expressions to a single line

```
Say "Hello there" yourname
```

```
Say "Multiply 3 by 5=" 3*5
```

Arg

- Gets values from the command line that are passed to the EXEC

```
Arg var1 var2 var3
```

- Command line arguments are translated to upper case
- Each word in the command line is assigned to a symbol/variable
 - Too many args, the last symbol contains the remainder of the parameter list
 - Too few args, the unused symbols are set to null
- Use Parse Arg to respect mixed case values

Writing a REXX program

Arg - Examples

```
/* */  
arg var1 var2 var3  
say 'var1=' var1  
say 'var2=' var2  
say 'var3=' var3
```

```
examp3 mine yours  
var1= MINE  
var2= YOURS  
var3=
```

```
examp3 mine yours ours theirs  
var1= MINE  
var2= YOURS  
var3= OURS THEIRS
```

```
/* */  
Parse Arg var1 var2 var3  
say 'var1=' var1  
say 'var2=' var2  
say 'var3=' var3
```

```
examp3 mine yours  
var1= mine  
var2= yours  
var3=
```

Pull

- Read values from the console or program stack

```
Pull var1 var2 var3
```

- Works like Arg; each word entered from the console or from the program stack is assigned to a symbol/variable
- Use Parse Pull to respect mixed case values

```
/* */  
Pull var1 var2 var3  
say 'var1=' var1  
say 'var2=' var2  
say 'var3=' var3
```

```
examp4  
one two three four  
var1= ONE  
var2= TWO  
var3= THREE FOUR
```

1. Where does Arg get values from?
2. Where does Pull get values from?
3. What keyword is used to preserve the value case for Arg/Pull?

1. Where does Arg get values from?

The Command line

2. Where does Pull get values from?

The console or program stack

3. What keyword is used to preserve the value case for Arg/Pull?

Parse

If/Then/Else

If expression Then

instruction1

Else

instruction2

- Evaluates an expression (the condition)
- If true, executes instruction1
- If false, executes instruction2
- Instructions can be
 - nop No operation
 - Do/End A block of clauses/instructions
 - If/Then/Else Another If structure

```
/* */ 87,799 COLUMNS  
num1 = 10 SCROLL ==  
num2 = 20 ZZ6034I TN3270  
If num1 = num2 Then  
say 'Equal' IP..PORT: 72.2  
Else  
say 'Not equal' EZZ6034I TN3270
```


Arithmetic operators

Addition	+	$7 + 3 = 10$
Subtraction	-	$10 - 6 = 4$
Multiplication	*	$8 * 4 = 32$
Division	/	$14 / 2 = 7$
Power	**	$6 ** 2 = 36$
Integer Divide	%	$10 \% 3 = 3$
Remainder Divide	//	$10 // 3 = 1$

Comparison operators

Equal	=
Strictly equal	==
Not equal	<> != ^= \= ¬=
Greater than	>
Less than	<
Logical And	&
Logical Or	

Do Loops

```
Do condition
  instructions
End
```

- Evaluates condition
- If true, executes instructions
- If false, skips the loop
- Additional Do features
 - Do Forever Loop until told to terminate
 - Do ... Until *expr* Evaluate expression at **end** of loop
 - Do ... While *expr* Evaluate expression at **top** of loop

Writing a REXX program

Do examples

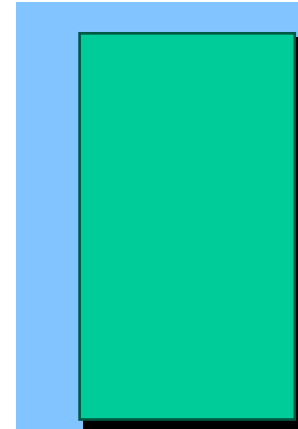
```
/* */
Do i = 1 to 10
    say i 'Cat'
End
```

```
1 Cat
2 Cat
3 Cat
4 Cat
5 Cat
6 Cat
7 Cat
8 Cat
9 Cat
10 Cat
```

```
Do Forever
  say 'Cat'
End
```

[illegible]

```
/* */
Do i = 1 to 10 by 2
    say 'Cat'
End
```



Do Loops

- Use the LEAVE command to get out of a loop
- Use the ITERATE command to go to the top of a loop

```
/* */  
i = 0  
Do Forever  
  i = i + 1  
  say i 'Cat'  
  If i > 10 Then  
    Leave  
End
```

```
1 Cat  
2 Cat  
3 Cat  
4 Cat  
5 Cat  
6 Cat  
7 Cat  
8 Cat  
9 Cat  
10 Cat  
11 Cat
```

```
/* */  
Do i = 1 to 10  
  If i = 4 Then  
    Iterate  
  say i 'Cat'  
End
```

```
1 Cat  
2 Cat  
3 Cat  
4 Cat  
5 Cat  
6 Cat  
7 Cat  
8 Cat  
9 Cat  
10 Cat
```

Do it again

- A Do...Forever loop needs LEAVE or EXIT to terminate
- A FOREVER loop can increment a counter with no end

```
Do i = 1
  say 'This will not stop' i
End
```

- A repetitive loop and conditional loop can be combined

```
Do i = 1 to 10 Until name <> "James"

  ...

End
```

Functions and Subroutines

- Both are code that is used to perform repetitive tasks
 - Avoid repeating the same logic in your code
- Can be internal or external
- Zero or more parameters can be passed
- A function must return a value
- A subroutine can return a value, but is not required to

Functions and Subroutines - Examples

```
/* */  
arg num1 num2  
sum = addthem(num1 num2)  
say 'Sum=' sum  
Exit  
  
addthem: procedure  
arg val1 val2  
res = val1 + val2  
return res
```

```
/* */  
arg num1 num2  
Call addthem num1 num2  
say 'Sum=' result  
Exit  
  
addthem: procedure  
arg val1 val2  
sum = val1 + val2  
return sum
```

```
/* */  
Arg num1 num2  
Call addthem  
Say 'Sum=' sum  
Exit  
  
addthem:  
sum = num1 + num2  
return
```

```
examp7 2 2  
Sum= 4
```


Functions and Subroutines – Variable references

- All variables from the caller are available to a function or subroutine by default
 - Conversely, any variable changed in the function or subroutine is available to the caller
- Variable isolation can be achieved by using 'Procedure' after the label

```
MyCode: Procedure
```

- The code can have visibility to the caller's variables with 'Expose' after the 'Procedure'

```
MyCode: Procedure Expose var1 var2 var3
```

- The caller's variables are protected from the function/subroutine unless included in the Expose list

String instructions

Length(string)	<i>Returns length of string</i>
Right(string, length, pad)	<i>Right justify string with pad to length</i>
Left(string, length, pad)	<i>Left justify string with pad to length</i>
Strip(string, opt, char)	<i>Strip characters from string (leading, trailing, both)</i>
Space(string, n, pad)	<i>Returns a string with n pad characters between each word; If n=0, removes all pad characters</i>
Substr(string, n, len, pad)	<i>Returns a portion of the string starting at position n for len. If not enough chars, use pad character</i>
Center(string, len, pad)	<i>Centers string within len characters, padded on each side</i>
Copies(string, n)	<i>Returns n copies of string</i>

String samples - Length

```
name = 'John'
```

```
Say 'Name length='Length(name)
```

```
-----
```

```
Name length=4
```

String samples - Right

```
state = 'New York'  
Say 'Last four='Right(state,4)
```

```
Last four=York
```

String samples – Right with pad

The pad specification comes into play when the count is larger than the string

```
state = 'Ohio'  
Say Right(state, 6, '.')
```

..Ohio

String samples - Strip

```
state = 'Florida  '
Say ':'state': '
Say ':'strip(state)': '
```

```
:Florida  :
:Florida:
```

Word handling

Words(string)	<i>Returns the number of words in string</i>
Wordlength(string, n)	<i>Returns the length of the n'th word in string</i>
Wordpos(phrase, string, start)	<i>Searches string for phrase; returns word number if found</i>
Wordindex(string, n)	<i>Returns the character position of the start of the n'th word</i>
Word(string, n)	<i>Returns the n'th word of string</i>

Word handling samples

```
string = 'Now is the time'  
Say 'Word 3='Word(string, 3)
```

```
Word 3=the
```


Word handling samples - Wordindex

```
string = 'Now is the time'  
say 'Third word starts at' Wordindex(string, 3)
```

```
Third word starts at 8
```

Word handling samples - Wordpos

```
string = 'Now is the time'  
say 'Position='Wordpos('the', string)
```

```
Position=3
```

Datatype

- Returns the datatype of a variable (NUM or CHAR)
- Returns true (1) or false (0) if the datatype matches second parameter

If Datatype(value, 'N') = 1 Then
 say "It's a number"

<u>A</u> lphanumeric	a-z A-Z 0-9
<u>B</u> inary	0 or 1
<u>L</u> owercase	a-z
<u>M</u> ixed	a-z A-Z
<u>N</u> umber	A valid number
<u>S</u> ymbol	Valid REXX variable
<u>U</u> ppercase	A-Z
<u>W</u> hole	Whole number
he <u>X</u> adecimal	Hex characters

Select

- Similar to a switch or case statement in other languages
- ‘Otherwise’ is required and ‘nop’ can be used
- Evaluates multiple expressions
- Really should be considered when nesting multiple “IF”s

Select

 When expression1 Then

 Instruction1

 When expression2 Then

 Instruction2

 When expression3 Then

 Instruction3

 Otherwise

 Instruction4

End

Select - Example

```
/* */  
arg age  
Select  
  When age < 50 Then  
    say 'Full warranty'  
  When age >= 50 Then  
    say 'Time to renew'  
  Otherwise  
    Nop  
End
```

```
examp6 38  
Full warranty
```

```
examp6 51  
Time to renew
```

RC (Return Code) and Exit

- RC is a special REXX variable that contains the return code from the last host command

```
'CP QUERY MAINT4'  
say 'Return code from query' rc
```

```
HCPCQV003E Invalid option - MAINT4  
Return code from query (3)
```

- The Exit command can be used to terminate a REXX program and optionally pass back a return code

```
Exit 4
```

- It is *good practice* to use Exit at the logical end of the program, even though REXX will exit when it reaches the end anyway

Signal

- Signal with a label name transfers control to the label
- Acts like GOTO in other languages

```
Signal MyRoutine
```

```
...
```

```
MyRoutine:
```

- Much better used to trap errors in a program

```
Signal ON|OFF condition
```

Conditions: ERROR	-non-zero rc from Host Command
HALT	-any external interrupt to halt
NOVALUE	-an un-initialized variable used
SYNTAX	-interpretation error detected
NOTREADY	-input/output not available

Signal

- Trapping errors can use special REXX variables
 - SIGL - Line number of the error
 - Sourceline - Function that returns the error line
 - Errortext - Function that returns the REXX error message

Signal - Example

```
/***/
Signal On Syntax
a = 1 + a

Syntax:
crc=crc
Say Errortext(crc)
Say 'On line' sigl':'
Say Sourceline(sigl)
Exit
```

```
Bad arithmetic conversion
On line 3:
a = 1 + a
```

Host commands

- A command that would typically be entered on the command line

`Address Command '...'`

`Address CMS '...'`

- Address Command is very specific, passing the parameter to CP/CMS without searching
- Address is important in multiple command environments
 - CP, CMS, XEDIT, CGI, etc

Host commands

- Recognized when a clause
 - Is not a comment
 - Is not a label
 - Is not an assignment
 - Does not begin with a REXX keyword (Do, If, Select, etc)

Tracing

- Produces diagnostic information about the program
- Various trace options

<u>E</u> rror	<i>Host commands with rc <> 0</i>
<u>C</u> ommand	<i>Host commands before execution</i>
<u>A</u> ll	<i>All clauses</i>
<u>R</u> esults	<i>All clauses; the result of any expression</i>
<u>I</u> ntermediate	<i>Intermediate results during expression evaluation</i>
<u>L</u> abels	<i>Passing through any labels</i>
<u>S</u> can without execution	<i>Trace without executing; Check for missing ends</i>
<u>N</u> ormal / <u>F</u> ailure	<i>Traces commands with negative return code</i>
<u>O</u> ff	<i>Turn trace off</i>

Tracing

- ‘S’ is good to check code structure
- ‘I’ is good for comprehensive tracing
 - Can be difficult to read with compute intensive routines
- ‘L’ is good to see which routines are called

Reading a trace – common tags

_	The source of a single clause/data is in the routine
+++	Trace message
>>>	Result of an expression

Tracing

- Intermediate trace tags

>V>	Contents of a variable
>L>	Literal; string or uninitialized variable
>F>	Result of a function call
>P>	Result of a prefix operation
>O>	Result of an operation on two terms
>C>	Name of a compound variable after substitution

Tracing - Example

```
/* */
trace i
arg age
Select
  When age < 50 Then
    say 'Full warranty'
  When age >= 50 Then
    say 'Time to renew'
  Otherwise
    Nop
End
```

```
examp6 44
3 *-* arg age
   >>> "44"
4 *-* Select
5 *-* When age < 50
   >V> "44"
   >L> "50"
   >O> "1"
   *-* Then
6 *-* say 'Full warranty'
   >L> "Full warranty"
Full warranty
```

Input/Output*

- Line input

Linein(name, line, count)

- name = Name of the stream
- line = Starting position
- count = Number of lines to read

- Omitting name causes Linein to read from the default input stream (eg: stdin)
- Linein() is the equivalent to a Pull
 - Reading input from the stack or console

*There are easier ways! Go to the CMS Pipelines session to learn more

Input/Output

- Line output

Lineout(name, string, line)

- name = Name of the stream
- string = Data to be written
- count = Number of lines to read

- Returns 0 if successful, 1 if not

- Lineout(name) is the equivalent to closing the stream (file)

Input/Output

- Lines()
- Returns the number of lines left in the stream

```
/* EXAMP11 EXEC */  
Do until Lines(examp11 exec) = 0  
    say linein(examp11 exec)  
End
```

```
examp11  
/* EXAMP11 EXEC */  
Do until Lines(examp11 exec) = 0  
    say linein(examp11 exec)  
End
```

Input/Output

- Charin()
 - Reads characters from an input stream
- Charout()
 - Write characters to an output stream
- Chars()
 - Number of characters remaining
- Rarely (if ever) used on z/VM
 - More relevant to other platforms (Linux, Windows, etc)

Value

- Value('count')
 - Returns the value of the variable 'count'
- Value('count', 4)
 - Returns current value of 'count' then sets 'count' to 4
- If the variable is not initialized, the variable name is returned in upper case

Translate

- Translate(string, tableo, tablei, pad)
- Translates characters in string from tablei to tableo
- tablei is a string of source characters
- tableo is a string of target characters

Examples

Translate('abcdef')	→ 'ABCDEF'
Translate('abbc', '&', 'b')	→ 'a&&c'
Translate('abcdef', '12', 'ec')	→ 'ab2d1f'
Translate('abcdef', '12', 'abcd', '.')	→ '12..ef'

Parse

- A very powerful mechanism to extract value(s) from a string

```
>>--PARSE-- .----- .-- .-ARG----- .->
      "-UPPER-"  | -EXTERNAL-----|
                  | -NUMERIC-----|
                  | -PULL-----|
                  | -SOURCE-----|
                  | -VALUE-- .----- .--WITH-|
                  |           "-expression-" |
                  | -VAR--name-----|
                  "-VERSION-----"

```



```
>-- .----- .-- ;-----<
      "-template list-"

```

Parse

- Assigns one or more values from a source string to variables based on a template list

Parse Value *expression* With template

- Parses the data that is the result of evaluating *expression*

Parse Var *variable* template

- Parses value of *variable* with template

Parse - Examples

```
/* EXAMP13 EXEC */  
Parse Value OneTwo() With val1 val2  
say val1 val2  
Exit  
  
OneTwo:  
Return 1 2
```

```
/* EXAMP14 EXEC */  
x = OneTwoThree()  
Parse Var x val1 . val2  
say val1 val2  
Exit  
  
OneTwoThree:  
Return 1 2 3
```

- Example 13 parses the result of a function
- Example 14 parses the value of a variable 'x'
uses '.' as a placeholder for one of the words of the value

Parse - Examples

Parse Value Time() With hh ':' mm ':' ss

- Breaks the current time into hours, minutes, seconds

```
/* EXAMP15 EXEC */ N 00000000 V31E 26092  
Parse Value Time() With hh ':' mm ':' ss  
say hh '-' mm '-' ss 41000000 V31E 26092
```

```
examp15  
12 - 37 - 17
```

Parse – Examples

- Positional parsing

Parse Value Time() With hh +2 . +1 mm +2 . +1 ss
10:15:30

Parse Value Time() With hh +2 . +1 mm +2 . +1 ss
10:15:30

Parse Value Time() With hh +2 . +1 mm +2 . +1 ss
10:15:30

...etc...

Parse - Examples

Parse Source template

- Parses information about the environment and the source of the program being executed

```
/* EXAMP16 EXEC */  
Parse Source vals  
say vals  
Parse Source . . fname ftype fmode .  
say fname ftype fmode
```

```
examp16  
CMS COMMAND EXAMP16 EXEC A1 examp16 CMS  
EXAMP16 EXEC A1
```

Parse - Examples

Parse External template

- Gets data from the terminal input buffer
- Leaves the program stack intact
 - If you need to have data from the user and don't want to deal with any data in the program stack
- Also allows a REXX program to retrieve data passed when a machine is XAUTOLOGged

Homework time!

All students have access to our z/VM system with their own CMS virtual machine

You have homework to do!

Homework WILL be checked before completion certificates are awarded

Create a REXX program to play the game Tic-Tac-Toe

Tic-Tac-Toe

A 3X3 grid of numbered squares

Player at the keyboard plays against the program

Players take turns

Display the current grid at the start and after each turn

Each player selects a square to play by number (1-9)

A previously selected square can not be played again

The player that puts their 'pieces' in three squares in any straight line is the winner

Send an email to homework@velocitysoftware.com when done

Send your homework program to the HOMEWORK user

'SENDFILE fileid HOMEWORK'

Questions!

Thank you for attending!

Training, Education and Information

<https://velocitysoftware.com/educate/>

Presentations and Handouts

<https://velocitysoftware.com/pdfs/>

