
pro 1000

1990 to present

Spelling and capitalization

Index

Looping

Conditional loops

Repetitive loops

Conditionals

Testing for multiple conditions

Compound variables

PARSE

NUMERIC

Error handling and exceptions


0

References

Further reading

External links

--

<u>Paradigm</u>	multiparadigm: procedural, structured
<u>Designed by</u>	<u>Mike Cowlishaw</u>
<u>Developer</u>	Mike Cowlishaw, <u>IBM</u>
<u>First appeared</u>	1979
<u>Stable release</u>	<u>ANSI X3.274</u> / 1996
<u>Typing discipline</u>	Dynamic
<u>Filename extensions</u>	.cmd, .bat, .exec, .rexx, .rex, EXEC
<u>Major implementations</u>	
VM/SP R3, TSO/E V2, SAAREXX, <u>ARexx</u> , BREXX, Regina, ^[1] Personal REXX, REXX/imc	
<u>Dialects</u>	
<u>NetRexx</u> , <u>Object REXX</u> , now <u>ooREXX</u> , <u>KEXX</u>	
<u>Influenced by</u>	
<u>PL/I</u> , <u>ALGOL</u> , <u>EXEC</u> , <u>EXEC 2</u>	
<u>Influenced</u>	
<u>NetRexx</u> , <u>Object REXX</u>	
 Rexx Programming at Wikibooks	

Rexx has the following characteristics and features:

- Simple syntax
- The ability to route commands to multiple environments
- The ability to support functions, procedures and commands associated with a specific invoking environment.
- A built-in stack, with the ability to interoperate with the host stack if there is one.
- Small instruction set containing just two dozen instructions
- Freeform syntax
- Case-insensitive tokens, including variable names
- Character string basis
- Dynamic data typing, no declarations
- No reserved keywords, except in local context
- No include file facilities
- Arbitrary numerical precision
- Decimal arithmetic, floating-point
- A rich selection of built-in functions, especially string and word processing
- Automatic storage management
- Crash protection
- Content addressable data structures
- Associative arrays
- Straightforward access to system commands and facilities
- Simple error-handling, and built-in tracing and debugger
- Few artificial limitations
- Simplified I/O facilities
- Unconventional operators
- Only partly supports Unix style command line parameters, except specific implementations
- Provides no basic terminal control as part of the language, except specific implementations
- Provides no generic way to include functions and subroutines from external libraries, except specific implementations

Rexx has just twenty-three, largely self-evident, instructions (such as `call`, `parse`, and `select`) with minimal punctuation and formatting requirements. It is essentially an almost free-form language with only one data-type, the character string; this philosophy means that all data are visible (symbolic) and debugging and tracing are simplified.

Rexx's syntax looks similar to PL/I, but has fewer notations; this makes it harder to parse (by program) but easier to use, except for cases where PL/I habits may lead to surprises. One of the REXX design goals was the principle of least astonishment.^[8]

History

pre-1990

Rexx was designed and first implemented, in assembly language, as an 'own-time' project between 20 March 1979 and mid-1982 by Mike Cowlshaw of IBM, originally as a scripting programming language to replace the languages EXEC and EXEC 2.^[2] It was designed to be a macro or scripting language for any system. As such, Rexx is considered a precursor to Tcl and Python. Rexx was also intended by its creator to be a simplified and easier to learn version of the PL/I programming language. However, some differences from PL/I may trip up the unwary.

It was first described in public at the SHARE 56 conference in Houston, Texas, in 1981,^[9] where customer reaction, championed by Ted Johnston of SLAC, led to it being shipped as an IBM product in 1982.

Over the years IBM included Rexx in almost all of its operating systems (VM/CMS, MVS TSO/E, IBM i, VSE/ESA, AIX, PC DOS, and OS/2), and has made versions available for Novell NetWare, Windows, Java, and Linux.

The first non-IBM version was written for PC DOS by Charles Daney in 1984/5^[3] and marketed by the Mansfield Software Group (founded by Kevin J. Kearney in 1986).^[2] The first compiler version appeared in 1987, written for CMS by Lundin and Woodruff.^[10] Other versions have also been developed for Atari, AmigaOS, Unix (many variants), Solaris, DEC, Windows, Windows CE, Pocket PC, DOS, Palm OS, QNX, OS/2, Linux, BeOS, EPOC32/Symbian, AtheOS, OpenVMS,^{[11]:p.305} Apple Macintosh, and Mac OS X.^[12]

The Amiga version of Rexx, called ARexx, was included with AmigaOS 2 onwards and was popular for scripting as well as application control. Many Amiga applications have an "ARexx port" built into them which allows control of the application from Rexx. One single Rexx script could even switch between different Rexx ports in order to control several running applications.

1990 to present

In 1990, Cathie Dager of SLAC organized the first independent Rexx symposium, which led to the forming of the REXX Language Association. Symposia are held annually.

Several freeware versions of Rexx are available. In 1992, the two most widely used open-source ports appeared: Ian Collier's REXX/imc for Unix and Anders Christensen's Regina^[1] (later adopted by Mark Hessling) for Windows and Unix. BREXX (<http://ftp.gwdg.de/pub/language/s/rexx/brexx/html/rx.html>) is well known for WinCE and Pocket PC platforms, and has been "back-ported" to VM/370 and MVS.

OS/2 has a visual development system from Watcom VX-REXX. Another dialect was VisPro REXX from Hockware.

Portable Rexx by Kilowatt and *Personal Rexx* by Quercus are two Rexx interpreters designed for DOS and can be run under Windows as well using a command prompt. Since the mid-1990s, two newer variants of Rexx have appeared:

- NetRexx: compiles to Java byte-code via Java source code; this has no reserved keywords at all, and uses the Java object model, and is therefore not generally upwards-compatible with 'classic' Rexx.
- Object REXX: an object-oriented generally upwards-compatible version of Rexx.

In 1996 American National Standards Institute (ANSI) published a standard for Rexx: ANSI X3.274–1996 "Information Technology – Programming Language REXX".^[13] More than two dozen books on Rexx have been published since 1985.

Rexx marked its 25th anniversary on 20 March 2004, which was celebrated at the REXX Language Association's 15th International REXX Symposium in Böblingen, Germany, in May 2004.

On October 12, 2004, IBM announced their plan to release their Object REXX implementation's sources under the Common Public License. Recent releases of Object REXX contain an ActiveX Windows Scripting Host (WSH) scripting engine implementing this version of the Rexx language.

On February 22, 2005, the first public release of Open Object Rexx (ooRexx) was announced. This product contains a WSH scripting engine which allows for programming of the Windows operating system and applications with Rexx in the same fashion in which Visual Basic and JScript are implemented by the default WSH installation and Perl, Tcl, Python third-party scripting engines.

As of January 2017 REXX was listed in the TIOBE index as one of the fifty languages in its top 100 not belonging to the top 50.^[14]

In 2019, the 30th Rexx Language Association Symposium marked the 40th anniversary of Rexx. The symposium was held in Hursley, England, where Rexx was first designed and implemented.^[15]

Toolkits

Rexx/Tk, a toolkit for graphics to be used in Rexx programmes in the same fashion as Tcl/Tk is widely available.

A Rexx IDE, RxxxEd, has been developed for Windows.^[11] RxSock for network communication as well as other add-ons to and implementations of Regina Rexx have been developed, and a Rexx interpreter for the Windows command line is supplied in most Resource Kits for various versions of Windows and works under all of them as well as DOS.

Spelling and capitalization

Originally the language was called *Rex* (*Reformed Executor*); the extra "X" was added to avoid collisions with other products' names. REX was originally all uppercase because the mainframe code was uppercase oriented. The style in those days was to have all-caps names, partly because almost all code was still all-caps then. For the product it became REXX, and both editions of Mike Cowlishaw's book use all-caps. The expansion to *REstructured eXtended eXecutor* was used for the system product in 1984.^[8]

Syntax

Looping

The loop control structure in Rexx begins with a **DO** and ends with an **END** but comes in several varieties. NetRexx uses the keyword **LOOP** instead of **DO** for looping, while ooRexx treats **LOOP** and **DO** as equivalent when looping.

Conditional loops

Rexx supports a variety of traditional structured-programming loops while testing a condition either before (**do while**) or after (**do until**) the list of instructions are executed:

```
do while [condition]
[instructions]
end
```

```
do until [condition]
[instructions]
end
```

Repetitive loops

Like most languages, Rexx can loop while incrementing an index variable and stop when a limit is reached:

```
do index = start [to limit] [by increment] [for count]
[instructions]
end
```

The increment may be omitted and defaults to 1. The limit can also be omitted, which makes the loop continue forever.

Rexx permits counted loops, where an expression is computed at the start of the loop and the instructions within the loop are executed that many times:

```
do expression
[instructions]
end
```

Rexx can even loop until the program is terminated:

```
do forever
[instructions]
end
```

A program can break out of the current loop with the `leave` instruction, which is the normal way to exit a `do forever` loop, or can short-circuit it with the `iterate` instruction.

Combined loops

Like PL/I, Rexx allows both conditional and repetitive elements to be combined in the same loop:^[16]

```
do index = start [to limit] [by increment] [for count] [while condition]
[instructions]
end
```

```
do expression [until condition]
[instructions]
end
```

Conditionals

Testing conditions with IF:

```
if [condition] then
do
[instructions]
end
else
do
[instructions]
end
```

The ELSE clause is optional.

For single instructions, DO and END can also be omitted:

```
if [condition] then
[instruction]
else
[instruction]
```

Indentation is optional, but it helps improve the readability.

Testing for multiple conditions

SELECT is Rexx's CASE structure, like many other constructs derived from PL/I. Like some implementations of CASE constructs in other dynamic languages, Rexx's WHEN clauses specify full conditions, which need not be related to each other. In that, they are more like cascaded sets of IF - THEN - ELSEIF - THEN - . . . - ELSE code than they are like the C or Java `switch` statement.

```
select
when [condition] then
[instruction] or NOP
when [condition] then
do
[instructions] or NOP
end
otherwise
[instructions] or NOP
end
```

The NOP instruction performs "no operation", and is used when the programmer wishes to do nothing in a place where one or more instructions would be required.

The OTHERWISE clause is optional. If omitted and no WHEN conditions are met, then the SYNTAX condition is raised.

Simple variables

Variables in Rexx are typeless, and initially are evaluated as their names, in upper case. Thus a variable's type can vary with its use in the program:

```
say hello /* => HELLO */
hello = 25
say hello /* => 25 */
hello = "say 5 + 3"
say hello /* => say 5 + 3 */
interpret hello /* => 8 */
drop hello
say hello /* => HELLO */
```

Compound variables

Unlike many other programming languages, classic Rexx has no direct support for arrays of variables addressed by a numerical index. Instead it provides *compound variables*.^[17] A compound variable consists of a stem followed by a tail. A . (dot) is used to join the stem to the tail. If the tails used are numeric, it is easy to produce the same effect as an array.

```
do i = 1 to 10
stem.i = 10 - i
end
```

Afterwards the following variables with the following values exist: `stem.1 = 9`, `stem.2 = 8`, `stem.3 = 7`...

Unlike arrays, the index for a stem variable is not required to have an integer value. For example, the following code is valid:

```
i = 'Monday'
stem.i = 2
```

In Rexx it is also possible to set a default value for a stem.

```
stem. = 'Unknown'
stem.1 = 'USA'
stem.44 = 'UK'
stem.33 = 'France'
```

After these assignments the term `stem.3` would produce 'Unknown'.

The whole stem can also be erased with the DROP statement.

```
drop stem.
```

This also has the effect of removing any default value set previously.

By convention (and not as part of the language) the compound `stem.0` is often used to keep track of how many items are in a stem, for example a procedure to add a word to a list might be coded like this:

```
add_word: procedure expose dictionary.
parse arg w
```

```
n = dictionary.0 + 1
dictionary.n = w
dictionary.0 = n
return
```

It is also possible to have multiple elements in the tail of a compound variable. For example:

```
m = 'July'
d = 15
y = 2005
day.y.m.d = 'Friday'
```

Multiple numerical tail elements can be used to provide the effect of a multi-dimensional array.

Features similar to Rexx compound variables are found in many other languages (including [associative arrays](#) in AWK, [hashes](#) in Perl and [Hashtables](#) in Java). Most of these languages provide an instruction to iterate over all the keys (or *tails* in Rexx terms) of such a construct, but this is lacking in classic Rexx. Instead it is necessary to keep auxiliary lists of tail values as appropriate. For example, in a program to count words the following procedure might be used to record each occurrence of a word.

```
add_word: procedure expose count. word_list
parse arg w .
count.w = count.w + 1 /* assume count. has been set to 0 */
if count.w = 1 then word_list = word_list w
return
```

and then later:

```
do i = 1 to words(word_list)
w = word(word_list,i)
say w count.w
end
```

At the cost of some clarity it is possible to combine these techniques into a single stem:

```
add_word: procedure expose dictionary.
parse arg w .
dictionary.w = dictionary.w + 1
if dictionary.w = 1 /* assume dictionary. = 0 */
then do
n = dictionary.0+1
dictionary.n = w
dictionary.0 = n
end
return
```

and later:

```
do i = 1 to dictionary.0
w = dictionary.i
say i w dictionary.w
end
```

Rexx provides no safety net here, so if one of the words happens to be a whole number less than `dictionary.0` this technique will fail mysteriously.

Recent implementations of Rexx, including IBM's Object REXX and the open source implementations like ooRexx include a new [language construct](#) to simplify iteration over the value of a stem, or over another collection object such as an array, table or list.

```
do i over stem.
say i '-->' stem.i
end
```

Keyword instructions

PARSE

The PARSE instruction is particularly powerful; it combines some useful string-handling functions. Its syntax is:

```
parse [upper] origin [template]
```

where *origin* specifies the source:

- `arg` (arguments, at top level tail of command line)
- `linein` (standard input, e.g. keyboard)
- `pull` (Rexx data queue or standard input)
- `source` (info on how program was executed)
- `value` (an expression) with: the keyword `with` is required to indicate where the expression ends
- `var` (a variable)
- `version` (version/release number)

and *template* can be:

- list of variables
- column number delimiters
- literal delimiters

`upper` is optional; if specified, data will be converted to upper case before parsing.

Examples:

Using a list of variables as template

```
myVar = "John Smith"
parse var myVar firstName lastName
say "First name is:" firstName
say "Last name is:" lastName
```

displays the following:

```
First name is: John
Last name is: Smith
```

Using a delimiter as template:

```
myVar = "Smith, John"
parse var myVar LastName ", " FirstName
say "First name is:" firstName
say "Last name is:" lastName
```

also displays the following:

```
First name is: John
Last name is: Smith
```

Using column number delimiters:

```
myVar = "(202) 123-1234"
parse var MyVar 2 AreaCode 5 7 SubNumber
say "Area code is:" AreaCode
say "Subscriber number is:" SubNumber
```

displays the following:

```
Area code is: 202
Subscriber number is: 123-1234
```

A template can use a combination of variables, literal delimiters, and column number delimiters.

INTERPRET

The `INTERPRET` instruction evaluates its argument and treats its value as a Rexx statement. Sometimes `INTERPRET` is the clearest way to perform a task, but it is often used where clearer code is possible using, e.g., `value()`.

Other uses of `INTERPRET` are Rexx's (decimal) arbitrary precision arithmetic (including fuzzy comparisons), use of the `PARSE` statement with programmatic templates, stemmed arrays, and sparse arrays.

```
/* demonstrate INTERPRET with square(4) => 16 */
X = 'square'
interpret 'say' X || '(4) ; exit'
SQUARE: return arg(1)**2
```

This displays 16 and exits. Because variable contents in Rexx are strings, including rational numbers with exponents and even entire programs, Rexx offers to interpret strings as evaluated expressions.

This feature could be used to pass functions as *function parameters*, such as passing SIN or COS to a procedure to calculate integrals.

Rexx offers only basic math functions like ABS, DIGITS, MAX, MIN, SIGN, RANDOM, and a complete set of hex plus binary conversions with bit operations. More complex functions like SIN were implemented from scratch or obtained from third party external libraries. Some external libraries, typically those implemented in traditional languages, did not support extended precision.

Later versions (non-classic) support CALL variable constructs. Together with the built-in function VALUE, CALL can be used in place of many cases of INTERPRET. This is a classic program:

```
/* terminated by input "exit" or similar */
do forever ; interpret linein() ; end
```

A slightly more sophisticated "Rexx calculator":

```
x = 'input BYE to quit'
do until x = 'BYE' ; interpret 'say' x ; pull x ; end
```

PULL is shorthand for parse upper pull. Likewise, ARG is shorthand for parse upper arg.

The power of the INTERPRET instruction had other uses. The Valour software package relied upon Rexx's interpretive ability to implement an OOP environment. Another use was found in an unreleased Westinghouse product called *Time Machine* that was able to fully recover following a fatal error.

NUMERIC

```
say digits() fuzz() form() /* => 9 0 SCIENTIFIC */
say 999999999+1 /* => 1.000000000E+9 */
numeric digits 10 /* only limited by available memory */
say 999999999+1 /* => 1000000000 */

say 0.999999999=1 /* => 0 (false) */
numeric fuzz 3
say 0.99999999=1 /* => 1 (true) */
say 0.99999999==1 /* => 0 (false) */

say 100*123456789 /* => 1.23456789E+10 */
numeric form engineering
say 100*123456789 /* => 12.34567890E+9 */

say 53 // 7 /* => 4 (rest of division)*/
```

	Calculates $\sqrt{2}$	Calculates e
code	<pre>numeric digits 50 n=2 r=1 do forever /* Newton's method */ rr=(n/r+r)/2 if r=rr then leave r=rr end say "sqrt" n ' = ' r</pre>	<pre>numeric digits 50 e=2.5 f=0.5 do n=3 f=f/n ee=e+f if e=ee then leave e=ee end say "e =" e</pre>
output	sqrt 2 = 1.414213562373095048801688724209698078569671875377	e = 2.7182818284590452353602874713526624977572470936998

SIGNAL

The SIGNAL instruction is intended for abnormal changes in the flow of control (see the next section). However, it can be misused and treated like the GOTO statement found in other languages (although it is not strictly equivalent, because it terminates loops and other constructs). This can produce difficult-to-read code.

Error handling and exceptions

It is possible in Rexx to intercept and deal with errors and other exceptions, using the SIGNAL instruction. There are seven system conditions: ERROR, FAILURE, HALT, NOVALUE, NOTREADY, LOSTDIGITS and SYNTAX. Handling of each can be switched on and off in the source code as desired.

The following program will run until terminated by the user:


```

signal on halt;
do a = 1
  say a
  do 100000 /* a delay */
  end
end
halt:
say "The program was stopped by the user"
exit

```

A **signal on** novalue statement intercepts uses of undefined variables, which would otherwise get their own (upper case) name as their value. Regardless of the state of the NOVALUE condition, the status of a variable can always be checked with the built-in function SYMBOL returning VAR for defined variables.

The VALUE function can be used to get the value of variables without triggering a NOVALUE condition, but its main purpose is to read and set environment variables, similar to POSIX `getenv` and `putenv`.

Conditions

ERROR

Positive RC from a system command

FAILURE

Negative RC for a system command (e.g. command doesn't exist)

HALT

Abnormal termination

NOVALUE

An unset variable was referenced

NOTREADY

Input or output error (e.g. read attempts beyond end of file)

SYNTAX

Invalid program syntax, or some other error condition

LOSTDIGITS

Significant digits are lost (ANSI REXX, not in TRL second edition)

When a condition is handled by **SIGNAL ON**, the SIGL and RC system variables can be analyzed to understand the situation. RC contains the REXX error code and SIGL contains the line number where the error arose.

Beginning with REXX version 4 conditions can get names, and there's also a **CALL ON** construct. That's handy if external functions do not necessarily exist:

```

ChangeCodePage: procedure /* protect SIGNAL settings */
signal on syntax name ChangeCodePage.Trap
return SysQueryProcessCodePage()
ChangeCodePage.Trap: return 1004 /* windows-1252 on OS/2 */

```

See also

- ISPF
- XEDIT
- Comparison of computer shells
- Comparison of programming languages

References

1. Mark Hessling (October 25, 2012). "Regina REXX Interpreter" (<http://regina-rexx.sourceforge.net>). SourceForge project regina-rexx. Retrieved February 10, 2014.
2. M. F. Cowlishaw. "IBM REXX Brief History" (http://www.rexxla.org/links/IBM_historical_pages/rexxhist.html). IBM. Retrieved August 15, 2006.
3. Melinda Varian. "REXX Symposium, May 1995" (<http://www.rexxla.org/events/1995/report.html>).
4. "Catalog of All Documents (filter=rexx)" (http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/FINDBOOK?filter=rexx). IBM library server. 2005. Retrieved February 10, 2014.
5. "Does ArcaOS include REXX support?" (<https://www.arcanos.com/faqwd/does-arcaos-include-rexx-support>). Retrieved September 3, 2020.
6. *IBM Virtual Machine Facility /370: EXEC User's Guide* (http://bitsavers.org/pdf/ibm/370/VM_370/Release_2/GC20-1812-1_VM370_EXEC_Users_Guide_Rel_2_Apr75.pdf) (PDF) (Second ed.). International Business Machines Corporation. April 1975. GC20-1812-1.
7. *EXEC 2 Reference* (http://bitsavers.org/pdf/ibm/370/VM_SP/Release_2_Jun82/SC24-5219-1_VM_SP_EXEC_2_Rel_2_Reference_Apr82.pdf) (PDF) (Second ed.). International Business Machines Corporation. April 1982. p. 92. SC24-5219-1.

8. M. F. Cowlshaw (1984). "The design of the REXX language" (<https://www.cs.tufts.edu/~nr/cs257/archive/mike-cowlshaw/rexx.pdf>) (PDF). *IBM Systems Journal*, VOL 23. NO 4, 1984 (PDF). *IBM Research*. 23 (4): 333. doi:10.1147/sj.234.0326 (<http://doi.org/10.1147/sj.234.0326>). Retrieved January 23, 2014. "Could there be a high astonishment factor associated with the new feature? If a feature is accidentally misapplied by the user and causes what appears to him to be an unpredictable result, that feature has a high astonishment factor and is therefore undesirable. If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature."
9. M. F. Cowlshaw (February 18, 1981). "REXX -- A Command Programming Language" (<http://speleotrove.com/rexxhist/share56.txt>). SHARE. Retrieved August 15, 2006.
10. Lundin, Leigh; Woodruff, Mark (April 23, 1987). "T/REXX, a REXX compiler for CMS" (<http://www.copyrightsearch.org/trex-a-rexx-compiler-for-cms/TXu000295377>). U.S. Copyright Office. Washington, DC: Independent Intelligence Incorporated (TXu000295377).
11. Howard Fosdick (2005). *Rexx Programmer's Reference*. Wiley Publishing. p. 390. ISBN 0-7645-7996-7.
12. "Rexx Implementations" (https://web.archive.org/web/20060924072512/http://www.rexxla.org/About_Rexx/mfc/rexxplat.html). RexxLA. Archived from the original (http://www.rexxla.org/About_Rexx/mfc/rexxplat.html) on September 24, 2006. Retrieved August 15, 2006.
13. While ANSI INCITS 274-1996/AMD1-2000 (R2001) and ANSI INCITS 274-1996 (R2007) are chargeable, a free draft can be downloaded: "American National Standard for Information Systems – Programming Language REXX" (<http://www.rexxla.org/rexxlang/standards/j18pub.pdf>) (PDF). X3J18-199X.
14. "The Next 50 Programming Languages" (<http://www.tiobe.com/tiobe-index/>). *TIOBE index*. tiobe.com. 2017. Archived (<https://web.archive.org/web/20170119054400/http://www.tiobe.com/tiobe-index/>) from the original on January 19, 2017. Retrieved January 10, 2017.
15. "RexxLA - Symposium Schedule" (<https://www.rexxla.org/events/schedule.rsp?year=2019>).
16. M. F. Cowlshaw (1990). *The Rexx Language - A Practical Approach to Programming* (2nd ed.). Prentice Hall. ISBN 0-13-780651-5.
17. "Six Rules of Thumb for Rexx" (<http://www.uic.edu/depts/accc/software/regina/rexxrule.html#p02h24>).

Further reading

- Callaway, Merrill. *The ARexx Cookbook: A Tutorial Guide to the ARexx Language on the Commodore Amiga Personal Computer*. Whitestone, 1992. ISBN 978-0963277305.
- Callaway, Merrill. *The Rexx Cookbook: A Tutorial Guide to the Rexx Language in OS/2 & Warp on the IBM Personal Computer*. Whitestone, 1995. ISBN 0-9632773-4-0.
- Cowlshaw, Michael. *The Rexx Language: A Practical Approach to Programming*. Prentice Hall, 1990. ISBN 0-13-780651-5.
- Cowlshaw, Michael. *The NetRexx Language*. Prentice Hall, 1997. ISBN 0-13-806332-X.
- Daney, Charles. *Programming in REXX*. McGraw-Hill, TX, 1990. ISBN 0-07-015305-1.
- Ender, Tom. *Object-Oriented Programming With Rexx*. John Wiley & Sons, 1997. ISBN 0-471-11844-3.
- Fosdick, Howard. *Rexx Programmer's Reference*. Wiley/Wrox, 2005. ISBN 0-7645-7996-7.
- Gargiulo, Gabriel. *REXX with OS/2, TSO, & CMS Features*. MVS Training, 1999 (third edition 2004). ISBN 1-892559-03-X.
- Goldberg, Gabriel and Smith, Philip H. *The Rexx Handbook*. McGraw-Hill, TX, 1992. ISBN 0-07-023682-8.
- Goran, Richard K. *REXX Reference Summary Handbook*. CFS Nevada, Inc., 1997. ISBN 0-9639854-3-4.
- IBM Redbooks. *Implementing Rexx Support in Sdsf*. Vervante, 2007. ISBN 0-7384-8914-X.
- Kiesel, Peter C. *Rexx: Advanced Techniques for Programmers*. McGraw-Hill, TX, 1992. ISBN 0-07-034600-3.
- Marco, Lou *ISPF/REXX Development for Experienced Programmers*. CBM Books, 1995. ISBN 1-878956-50-7
- O'Hara, Robert P. and Gomberg, David Roos. *Modern Programming Using Rexx*. Prentice Hall, 1988. ISBN 0-13-597329-5.
- Rudd, Anthony S. 'Practical Usage of TSO REXX'. CreateSpace, 2012. ISBN 978-1475097559.
- Schindler, William. *Down to Earth Rexx*. Perfect Niche Software, 2000. ISBN 0-9677590-0-5.

External links

- Mike Cowlshaw's home page (<http://speleotrove.com/mfc/mfc.html>)
- REXX language page (<https://www.ibm.com/us-en/marketplace/compiler-and-library-for-rexx-on-ibm-z>) at IBM
- REXX Language Association (<https://www.rexxla.org/>)
- Rexx programming language (<https://www.openhub.net/languages/rexx>) at Open Hub

Retrieved from "<https://en.wikipedia.org/w/index.php?title=Rexx&oldid=1052792380>"

This page was last edited on 31 October 2021, at 03:04 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.