Table of Contents

Brief Introduction to REXX

The Features

REXX is a versatile command programming language that was developed by IBM. It provides a common programming structure that has a free format and is easy to use. This makes REXX particularly good for beginners.

The basic design of REXX also makes it a very powerful language. It has a large range of built-in functions, extensive parsing capabilities, and the ability to run under any MVS address space.

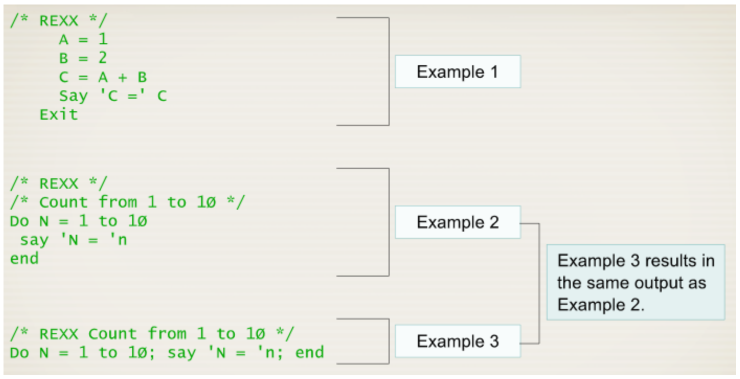
Some advantages include:

- Free formatting
- Very readable
- Easy to learn and use
- Has multiple, powerful, built-in functions
- Extensive string manipulation facilities
- Runs in all MVS address spaces and on many platforms

<u>Ease of Use</u>

Unlike other languages, such as C++ and assembly, REXX instructions are based on English and decimal arithmetic.

This enables programs to be easily interpreted and written, even by novices.



```
/* REXX */
     A = 1
     B = 2
     C = A + B
     Say 'C =' C
   Exit
```

Example 1

```
/* REXX */
/* Count from 1 to 1Ø */
Do N = 1 to 1Ø
 say 'N = 'n
end
```

Example 2

```
/* REXX Count from 1 to 1Ø */
Do N = 1 to 1Ø; say 'N = 'n; end
```

Example 3

Example 3 results in the same output as Example 2.

<u>Built-in Functions</u>

The REXX language provides a host of built-in functions that enable you to easily perform many common actions ranging from arithmetic conversions to text formatting.

- **String Functions** - these are used to perform common functions on data strings, such as finding characters in a string and adding, changing, or deleting characters from a string.
- **Word Functions** - these are used to perform various functions on words, which are blank delimited character strings, in a string or record.
- **Justification Functions** - these functions enable text in a character string to be centered or justified to the left or right of a record.
- **Numeric Functions** - these perform common actions on numeric strings of data, such as rounding, formatting, and random number generating.
- **Character Conversion Functions** - these enable character strings to be converted between hex, binary, decimal and character forms.
- **Environment Functions** - these are used to determine the current system settings.

Debugging Aids

In addition to basic error messaging, REXX provides numerous tracing options and interactive debugging facilities. These include interactive tracing, syntax checking and condition traps.

Interpreted Language

REXX is an interpreted language that does not require compilation. This means that the source code of the program can be executed directly, and each source instruction is checked and "interpreted" into machine code before being executed.

The main disadvantage of interpreted languages is that they often use up more CPU time, which makes them more expensive to run. They do, however, provide the benefits of increased flexibility and shorter development time. Although REXX does not require compilation, REXX routines can be compiled, which reduces the CPU resources required to run them, if necessary.

Compiled Language

A compiled language should be considered for applications that run every day and use a lot of machine time. But if the application changes a lot, issues a large volume of commands, or has to be developed quickly, REXX is probably the best solution.

In any event, once the application is written in REXX, it can always be compiled at a later stage.

Parsing Facility

REXX includes extensive parsing capabilities for character string manipulation. "Parsing" is the act of separating computer input into its constituent parts for subsequent processing actions.

The REXX parsing facility enables a pattern to be defined and used to separate characters, numbers, and mixed input into separate variables in a single instruction.

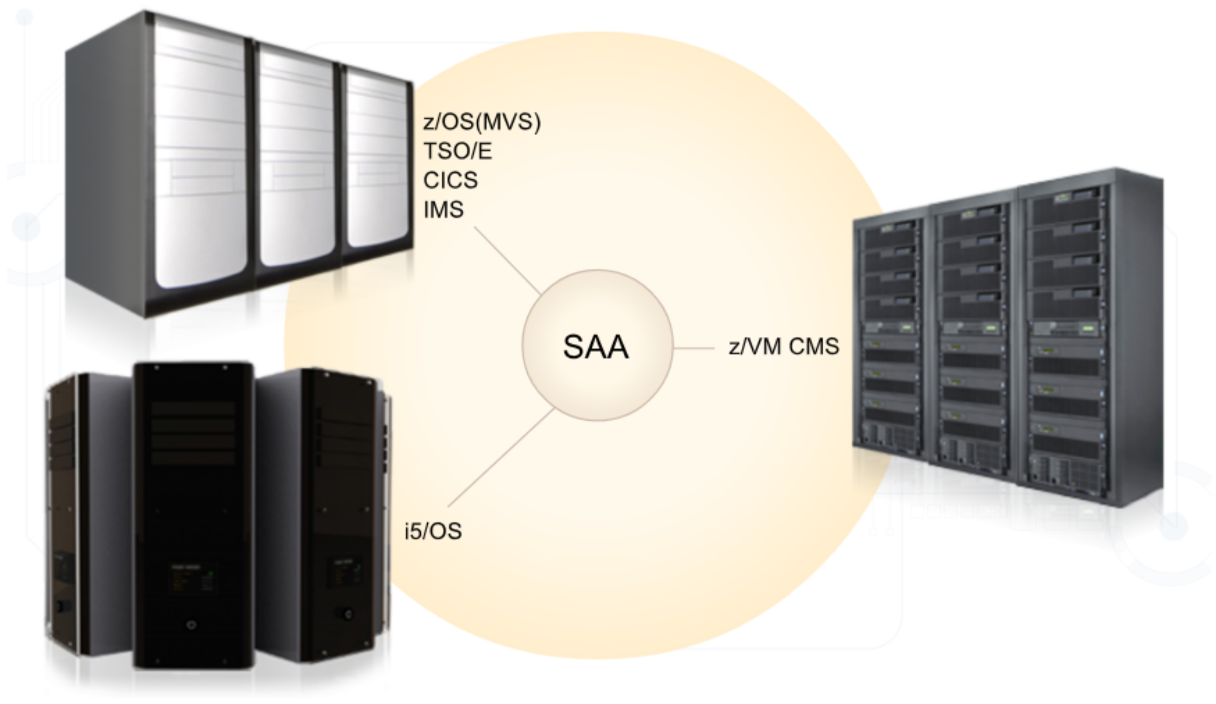
Components of REXX TSO/E

The following are the five basic components of REXX TSO/E:

- **The Commands** - these perform actions that are specific to the REXX environment, such as trapping screen messages.
- **TSO/E External Functions** - these commands, which can only be executed by REXX running under TSO/E, enable specific tasks such as data set and file management.
- **Built-in Functions** - these functions and procedures quickly perform common actions on character strings and system information.
- **External Data Queue** - this component, which is also called the data stack, enables data to be stored externally to the main REXX storage so it can be shared by other REXX routines running in the same address space.
- **Keyword Instructions** - the four types of keyword instructions are conversations with REXX, variable management, logic flow and execution control.

Systems Application Architecture (SAA)

When dealing with REXX programs, particularly in the IBM environment, you may encounter Systems Application Architecture (SAA).



SAA was an attempt to provide consistent interfaces across products. Many enterprise REXX systems were written to conform to SAA, which provided:

- A common programming interface
- Common communications support
- Common user access

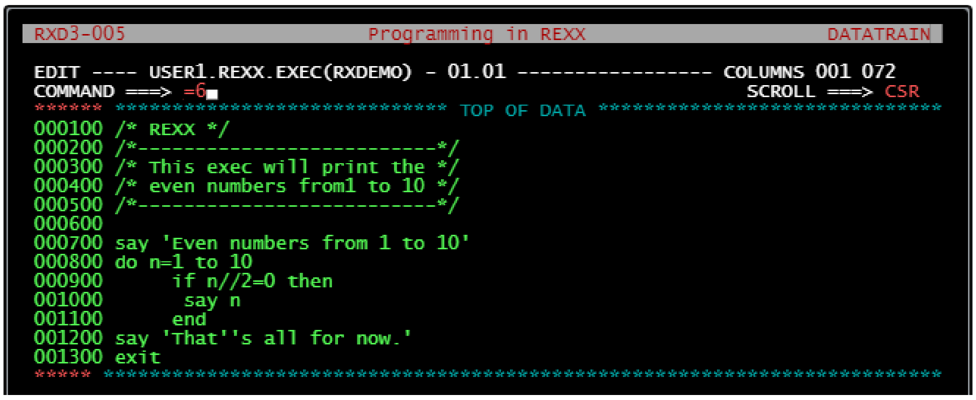
Other Platforms

REXX now runs on various IBM and non-IBM platforms, including z/OS, z/VM, i5/OS, Microsoft Windows, and Linux as an integrated part of the operating system or as an add-on product.

REXX acts on all these platforms as both a powerful scripting language to drive operating system functions and as a complete programming language.

Executing REXX from TSO/ISPF

This is a simple REXX program, which is also called an EXEC. It is defined in a z/OS environment under TSO/E and the ISPF editor.

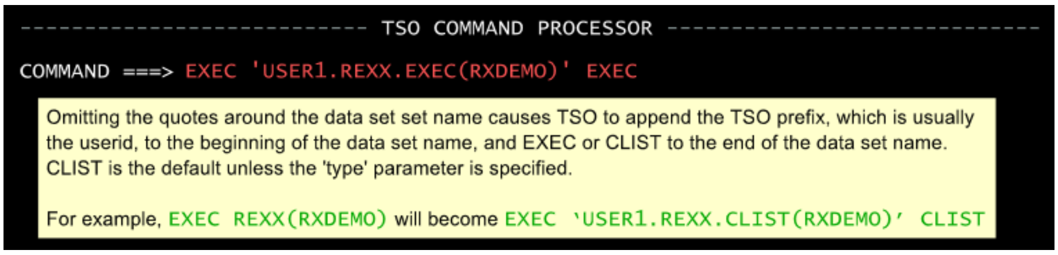


```
RXD3-005                  Programming in REXX                  DATATRAIN

EDIT ---- USER1.REXX.EXEC(RXDEMO) - 01.01 ---------------- COLUMNS 001 072
COMMAND ===> =6                                               SCROLL ===> CSR
****** ****************************** TOP OF DATA ******************************
/* REXX */
/*-------------------------*/
/* This exec will print the */
/* even numbers from1 to 10 */
/*-------------------------*/

say 'Even numbers from 1 to 10'
do n=1 to 10
     if n//2=0 then
      say n
     end
say 'That''s all for now.'
exit
****** ************************************************************************
```

EXECs are usually executed by a TSO command from the ISPF command line or from the TSO command processor.

Explicit Execution

REXX execs can be run explicitly by using the EXEC TSO command with a fully qualified data set name and member. Use the EXEC parameter after the data set name to distinguish the REXX exec from a CLIST.



```
-------------------------- TSO COMMAND PROCESSOR ----------------------------

COMMAND ===> EXEC 'USER1.REXX.EXEC(RXDEMO)' EXEC
```

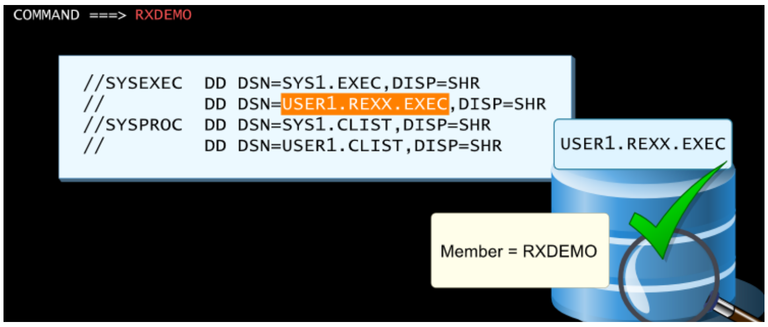
Omitting the quotes around the data set set name causes TSO to append the TSO prefix, which is usually the userid, to the beginning of the data set name, and EXEC or CLIST to the end of the data set name. CLIST is the default unless the 'type' parameter is specified.

For example, `EXEC REXX(RXDEMO)` will become `EXEC 'USER1.REXX.CLIST(RXDEMO)' CLIST`

Implicit Execution

Alternatively, a REXX exec can be executed implicitly by using the member name of the REXX exec, but the data set containing the REXX program in that member must be allocated to the DD name concatenation SYSEXEC, which is only used for EXECS, or SYSPROC, which is used for both CLISTS and EXECS. Note that REXX programs in data sets defined to SYSPROC must start with a comment containing the word "REXX".

When the REXX member name is executed, TSO searches for a program or command of that name, and then for the SYSEXEC and SYSPROC DD name concatenation libraries.



```
COMMAND ===> RXDEMO

//SYSEXEC  DD DSN=SYS1.EXEC,DISP=SHR
//         DD DSN=USER1.REXX.EXEC,DISP=SHR
//SYSPROC  DD DSN=SYS1.CLIST,DISP=SHR
//         DD DSN=USER1.CLIST,DISP=SHR
```

Extended Implicit Execution

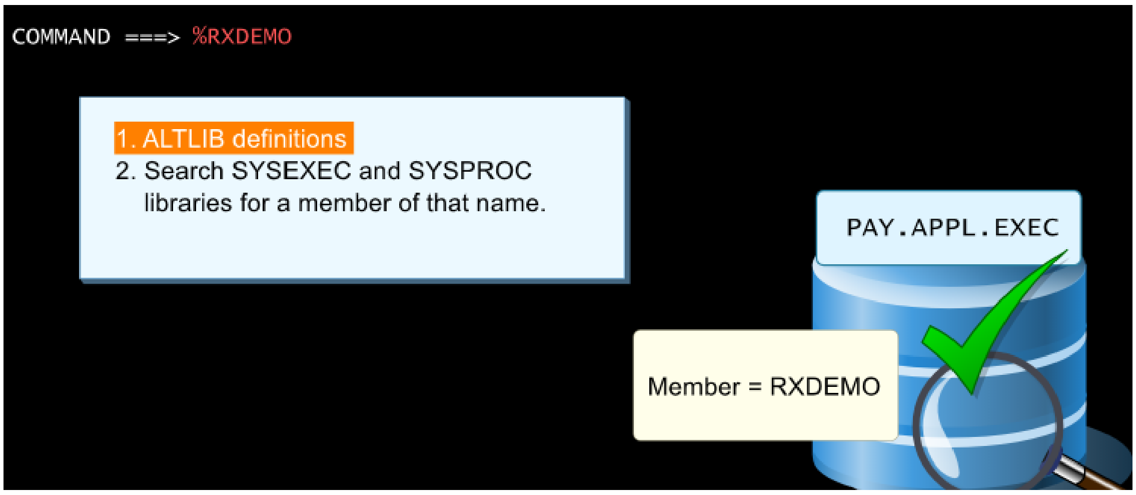
Occasionally, the name of the member in which the REXX program resides is the same as a TSO command or program, which causes confusion when you attempt to run it.

Preceding the member name with a percent sign (%) causes TSO to look only in the SYSEXEC and SYSPROC DD definition concatenations for the member containing the REXX program. This is called extended implicit execution.

ATLIB Definitions

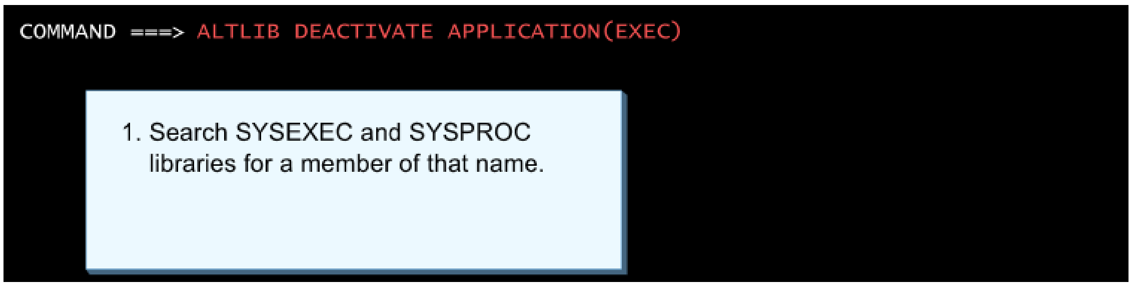
An alternate library can be added to the search order prior to the SYSEXEC and SYSPROC definitions by using the ALTLIB command.

Defining an ALTLIB enables REXX program libraries to be easily added to the search order for specific applications and then removed after the application has completed.



```
COMMAND ===> %RXDEMO
```

1. ALTLIB definitions
2. Search SYSEXEC and SYSPROC libraries for a member of that name.

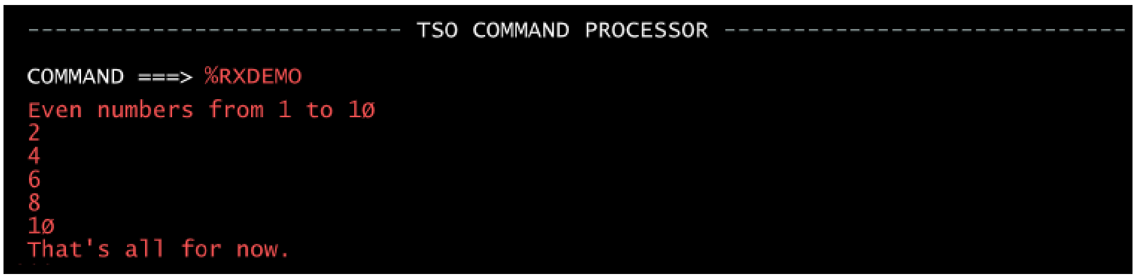


```
COMMAND ===> ALTLIB DEACTIVATE APPLICATION(EXEC)
```

1. Search SYSEXEC and SYSPROC libraries for a member of that name.

Executing REXX

The RXDEMO REXX is executed and its results are displayed.



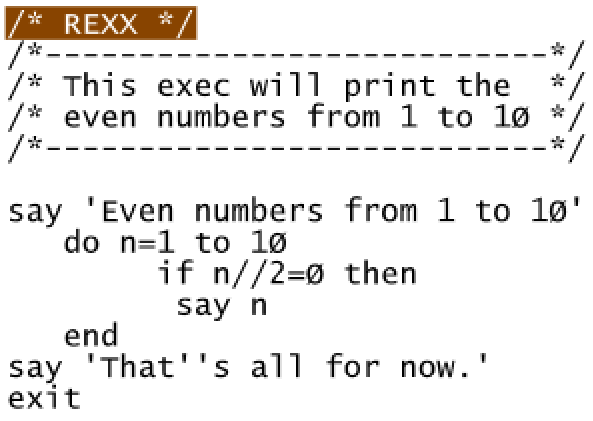
```
-------------------------- TSO COMMAND PROCESSOR ----------------------------

COMMAND ===> %RXDEMO
Even numbers from 1 to 10
2
4
6
8
10
That's all for now.
```

Processing REXX

This section will cover a step-by-step look into the REXX code that causes each line displayed in the example above to occur.

When the command **%RXDEMO** command is entered, TSO searches the REXX and CLIST libraries for the first occurrence of a member named RXDEMO.

When this is found, TSO opens the member and begins to interpret the program contained in the member.



```
/* REXX */
/*---------------------------*/
/* This exec will print the  */
/* even numbers from 1 to 10 */
/*---------------------------*/

say 'Even numbers from 1 to 10'
   do n=1 to 10
        if n//2=0 then
         say n
   end
say 'That''s all for now.'
exit
```

Identifying REXX Code

The first line identifies the EXEC to the systems interpreter. Because this line has a comment containing the word "REXX", the system invokes the REXX interpreter rather than the CLIST interpreter.

SAY Keyword Instruction

The comment lines enclosed by /* and */ are ignored and the interpreter goes to the first executable statement.

The SAY keyword instruction causes the text following it to be displayed on the screen.

DO Keyword Instruction

The next statement to be executed is the entry to a DO loop. The do keyword instruction defines the beginning of a set of statements through to the end keyword instruction, which will loop under defined conditions.

In this instance, the do keyword sets the value to 1 to be stored in the n variable. Each time the loop executes, this n variable will be incremented by 1 until it reaches the value of 11. In the if keyword, the remainder (the // operator - two slashes together) of n divided by 2 is checked to see if it is 0. Since this remainder is 1, nothing happens. Processing continues until the end keyword instruction then loops back to the do statement.

REXX Loops

The DO loop increments the value of n by 1 and tests to see if it is greater than 10. Since n is 2, the loop continues.

This time, n//2 returns 0 so the if expression evaluates to true and the then statement executes. The say statement tells REXX to write the value of n onto the terminal display.

Processing continues until n contains the value of 10. The program will now loop back once more to the do statement.

When n is incremented this time, it has the value 11. Because this is greater than 10, the loop is terminated and processing continues from the next statement after the loop.

The say keyword instruction causes the text in quotes to be written to the terminal. The two single quotes coded together ('') are translated to a single quote (').

Exiting REXX Programs

Finally, the exit statement tells REXX to terminate the exec. Control is then returned to TSO.

REXX Terms and Operators

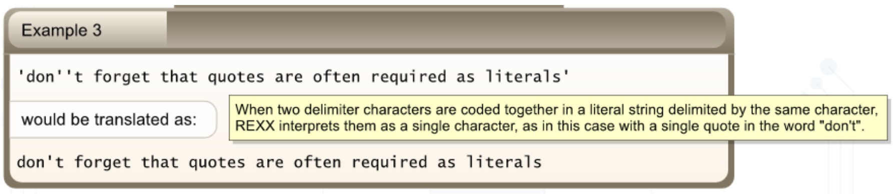
REXX Comments

A REXX comment is a sequence of characters delimited by **/* and */**. Comments can, in turn, have other comments nested within them. They can be placed anywhere in a REXX program and can continue over multiple lines. The REXX interpreter ignores comments.

Literal Strings

When a REXX program is executed, the interpreter "sees" everything as a symbol or variable. Any symbol or variable that has not been assigned a value is assigned a default value of the variable name with all alphabetic characters converted to uppercase.

If letters or character strings are to be treated as literals rather than symbols, they must be identified as literals by enclosing the relevant character string in single (') or double (") quotation marks. When the interpreter encounters one of these characters, it reads all the characters up to, but not including, the next quotation mark of the same type and interprets them as literals, exactly as typed.



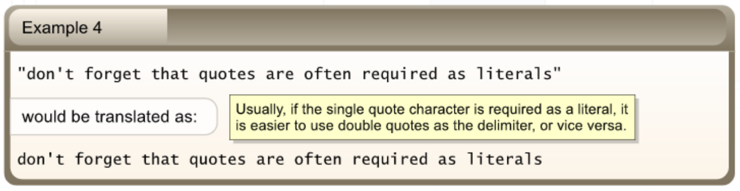
Example 3

```
'don''t forget that quotes are often required as literals'
```

would be translated as:

When two delimiter characters are coded together in a literal string delimited by the same character, REXX interprets them as a single character, as in this case with a single quote in the word "don't".

```
don't forget that quotes are often required as literals
```



Example 4

```
"don't forget that quotes are often required as literals"
```

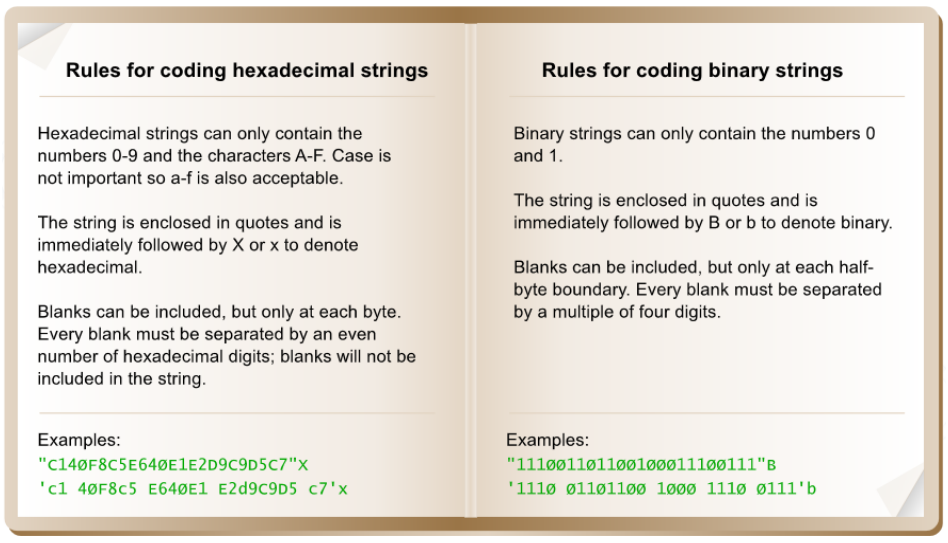
would be translated as:

Usually, if the single quote character is required as a literal, it is easier to use double quotes as the delimiter, or vice versa.

```
don't forget that quotes are often required as literals
```

Hexadecimal and Binary Strings

Hexadecimal and binary strings can also be defined. With some exceptions, these are coded like literal strings. Listed below are the rules for both:



Rules for coding hexadecimal strings

Hexadecimal strings can only contain the numbers 0-9 and the characters A-F. Case is not important so a-f is also acceptable.

The string is enclosed in quotes and is immediately followed by X or x to denote hexadecimal.

Blanks can be included, but only at each byte. Every blank must be separated by an even number of hexadecimal digits; blanks will not be included in the string.

Examples:

```
"C140F8C5E640E1E2D9C9D5C7"X
'c1 40F8c5 E640E1 E2d9C9D5 c7'x
```

Rules for coding binary strings

Binary strings can only contain the numbers 0 and 1.

The string is enclosed in quotes and is immediately followed by B or b to denote binary.

Blanks can be included, but only at each half-byte boundary. Every blank must be separated by a multiple of four digits.

Examples:

```
"1110011011001000111100111"B
'1110 01101100 1000 1110 0111'b
```

REXX Symbols

A REXX symbol consists of a group of characters. The characters can be any valid alphanumeric or any @, #, $, ¢, !, ?, . or _ characters.

Examples of symbols	Examples of labels
Client_address	Section_2:
Student.name:	Start_of_procedure:
@ISPNAME	Student.name:
RECORD.1	$PAYSECTION:

A symbol must begin with an alpha character or any of the special characters, except the period. Symbols can be variables, labels, constants, or REXX keywords. A label is a symbol with a colon (:) on the end.

REXX Numbers

A constant is a symbol that starts with a numeric character or a period. Constants are also referred to as numerics. Numerics can be expressed as whole numbers, decimals, and exponents.

```
7              /* equals 7                         */
-5             /* equals -5                        */
89.45256       /* equals 89.45256                  */
.005           /* equals 0.005                     */
31e3           /* equals 31,000                    */
47e12          /* equals 47,000,000,000,000        */
-.34e2         /* equals -34                       */
-.34e-2        /* equals -0.0034                   */
+3             /* equals 3                         */
```

Listed here are several examples of numerics and the value that the REXX interpreter will assign them.

Assignment Variables

A REXX assignment statement must contain a valid REXX symbol as the target or variable name of the assignment. The right side of an assignment can contain another variable, constant, compound variable, or expression composed of a combination of these.

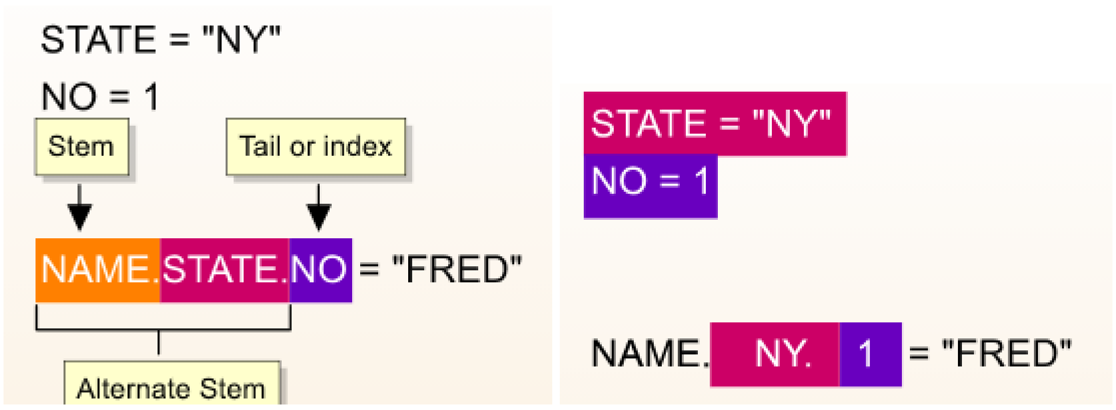
```
a = 7           /*a is assigned the value 7          */
b = a + 5       /*b is assigned the value 12  (7 + 5)   */
x = 'hello'
y = 'world'

xy2 = x      y  /*xy2 is assigned the string hello world*/
xy2 = x y       /*xy2 is assigned the string hello world*/
xy2 = xy        /*xy2 is assigned the string XY         */
msg1 = ''       /* msg1 assigned to a null string       */
msg1 =          /* msg1 assigned to a blank             */
msg.4 = 'Error' /* msg.4 is assigned the string Error   */
```

When assigning a variable to multiple values, unless they are defined as literals, REXX will regard multiple spaces as a single space. Shown here are some examples of assignment statements, and the use of abuttal characters and methods.

Compound Variables

A compound variable is the REXX equivalent of an array. It consists of a stem value with a period as the last character, which is followed by at least one other character called the tail. When the tail begins with a numeric, it is often referred to as the index. As with REXX variables, compound variables must not begin with a numeric character.

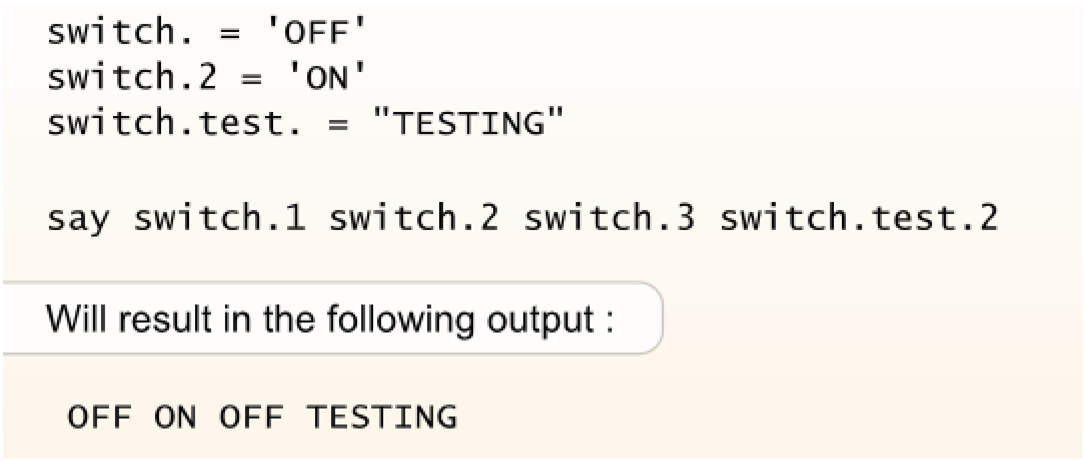
Compound variables can contain multiple stems and tails. A unique feature of the compound variable is that all qualifiers, except the primary stem, are treated as variables and will have their values substituted if assigned. The qualifiers are the characters between each period. A compound variable can be a maximum of 250 characters in length before and after variable substitution.



Stem Variables

A stem also has a useful property and this is that it always ends in a period.

When a stem variable is used as the target of an assignment, all compound variables that do or could start with that stem are set to the assigned value.



```
switch. = 'OFF'
switch.2 = 'ON'
switch.test. = "TESTING"

say switch.1 switch.2 switch.3 switch.test.2
```

Will result in the following output :

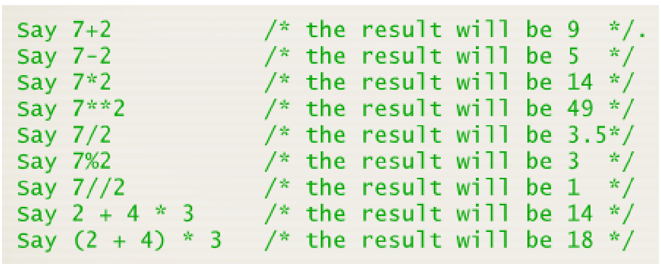
```
 OFF ON OFF TESTING
```

REXX Operators

An operator is a symbol that defines an operation. REXX operators are divided into the following as described below.

Arithmetic Operators

Arithmetic operators perform arithmetic functions. By default, numeric operations are performed to nine significant places.
The REXX interpreter can use the standard operators of +, -, *, and /, as well as the following special operators:



```
Say 7+2            /* the result will be 9  */.
Say 7-2            /* the result will be 5  */
Say 7*2            /* the result will be 14 */
Say 7**2           /* the result will be 49 */
Say 7/2            /* the result will be 3.5*/
Say 7%2            /* the result will be 3  */
Say 7//2           /* the result will be 1  */
Say 2 + 4 * 3      /* the result will be 14 */
Say (2 + 4) * 3    /* the result will be 18 */
```

- % indicates the integer component of a division operation
- // indicates the remainder of a division operation
- ** indicates exponentiation or "to the power of"; the power value must be a whole number

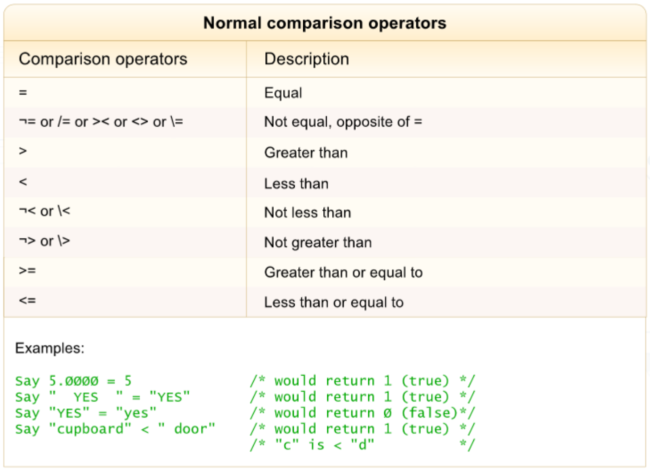
REXX arithmetic follows the standard mathematical precedence of **, *, //, %, /, +, -, and left to right, but parentheses (()) can be used to alter the order of precedence. Spaces coded between numerics and operators are ignored.

Normal Comparison Operators

Numerous comparison operators are available in REXX. Most of them are much the same as other languages.

REXX also accepts a number of forms for the logical NOT operator. Note the following:

- When using REXX on a mainframe, the most common NOT operator is the symbol Shift-6 on a 3270 series terminal.
- As most PCs do not have this character, the backslash (\ ) symbol can be used.
- When the expression is unambiguous, the forward slash (/) character can also be used.
- When comparing numerics, the dual characters of <> or >< can be used.



Normal comparison operators

Comparison operators	Description
=	Equal
¬= or /= or >< or <> or \=	Not equal, opposite of =
>	Greater than
<	Less than
¬< or \<	Not less than
¬> or \>	Not greater than
>=	Greater than or equal to
<=	Less than or equal to

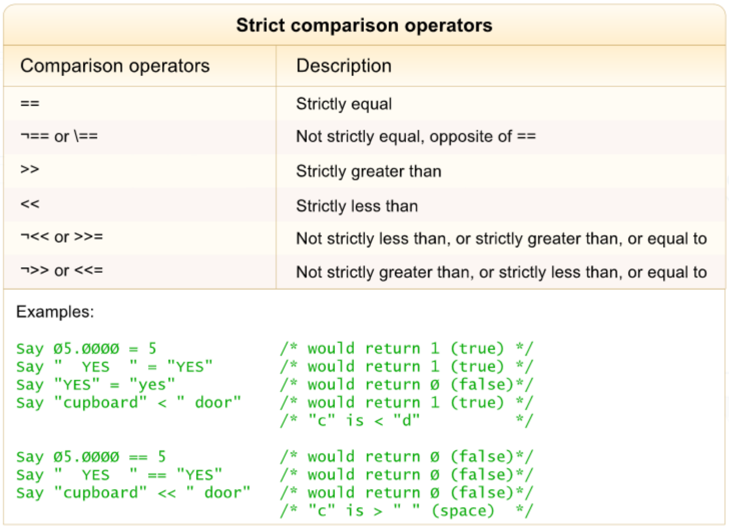
Examples:

```
Say 5.0000 = 5            /* would return 1 (true) */
Say "  YES  " = "YES"     /* would return 1 (true) */
Say "YES" = "yes"         /* would return 0 (false)*/
Say "cupboard" < " door"  /* would return 1 (true) */
                          /* "c" is < "d"          */
```

Comparisons return the value 1 when true and 0 when false. Listed above are the normal comparison characters and their descriptions.

Strict Comparison Operators

When the strings being compared are both numeric, REXX performs a numeric comparison. Other strings are compared on a character by character basis. String comparisons are case sensitive and the comparison order depends on the character set in use on the platform where the REXX program is running. This may be ASCII or EBCDIC.



Strict comparison operators

Comparison operators	Description
==	Strictly equal
¬== or \==	Not strictly equal, opposite of ==
>>	Strictly greater than
<<	Strictly less than
¬<< or >>=	Not strictly less than, or strictly greater than, or equal to
¬>> or <<=	Not strictly greater than, or strictly less than, or equal to

Examples:

```
Say Ø5.ØØØØ = 5             /* would return 1 (true) */
Say "  YES   " = "YES"      /* would return 1 (true) */
Say "YES" = "yes"           /* would return Ø (false)*/
Say "cupboard" < " door"    /* would return 1 (true) */
                            /* "c" is < "d"          */

Say Ø5.ØØØØ == 5            /* would return Ø (false)*/
Say "  YES   " == "YES"     /* would return Ø (false)*/
Say "cupboard" << " door"   /* would return Ø (false)*/
                            /* "c" is > " " (space)  */
```

Normal comparisons usually ignore leading and trailing blanks in string comparisons, and leading and trailing zeros in numeric comparisons. However, if strict comparison operands are used, that is, double comparison operators, the entire string is compared byte for byte.

Listed above are the strict comparison operators and some examples of normal and strict comparisons.

REXX Expressions

A REXX expression consists of a combination of operators, symbols, and parentheses.

```
a + b                       /* an arithmetic expression*/
'Hello' name                /* a string operation expression*/
z > 4                       /* a logical expression*/
```

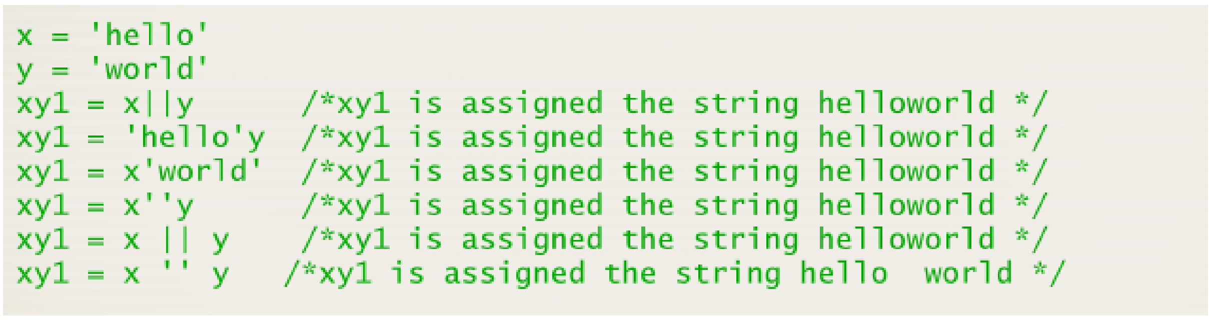
Logical Operators

Logical operators enable two expressions to be joined by a logical "and", "or", or "exclusive or" . These are generally used in conjunction with logical expressions to determine a multiple condition test for a true/false value. The three logical operators used by REXX are:

- & Logical "and" means if both expressions are true, the result is true; otherwise, the result is false. These are processed first, but
- parentheses can be used to alter the order of precedence.
- | Logical "or" means if either or both expressions are true, the result is true; otherwise, the result is false.
- && Logical "exclusive or" means if either, but not both, expressions are true, the result is true; otherwise the result is false.

Abuttal Characters

When an expression in an assignment statement is required to join two values together without any spaces, double vertical bars (||) can be used as a concatenation (abuttal) characters. Any spaces coded between literals or symbols on either side of abuttal characters, will be removed.



```
x = 'hello'
y = 'world'
xy1 = x||y       /*xy1 is assigned the string helloworld */
xy1 = 'hello'y   /*xy1 is assigned the string helloworld */
xy1 = x'world'   /*xy1 is assigned the string helloworld */
xy1 = x''y       /*xy1 is assigned the string helloworld */
xy1 = x || y     /*xy1 is assigned the string helloworld */
xy1 = x '' y    /*xy1 is assigned the string hello  world */
```

A double ("") string can also be used, but only between symbols that are abutted to them. Alternatively, literals can be abutted to variables by not coding any spaces between them.

Special Characters

The colon, semicolon, and comma are special characters that have specific meanings when used in REXX code:

- **The Colon (:)** - When a valid symbol is coded as the first word on a statement, immediately followed by a colon, it is considered to be a label. Labels are used to define procedures, as the targets for branching instructions (GOTO) and for documentation purposes, to identify a piece or section of code.
    - Example: read_record: or procedure1:
- **The Semicolon (;)** - This is the statement delimiter, that is, it marks the end of the current statement. REXX also interprets the end of line as the end of the current statement; therefore, it is only necessary to use the semicolon when placing two or more statements together on the same line.
    - Example: say ‘enter number?’; pull num; say num
- **The Comma (,)** - This is used to continue the current statement on to the next line. This is necessary because of what was covered above. The REXX interpreter detects a comma as the last character on a line, which is not part of a literal or comment, it assumes the next line is a continuation preceded by a space.