2

# Language Basics

## Overview

This chapter describes the basic elements of Rexx. It discusses the simple components that make up the language. These include script structure, elements of the language, operators, variables, and the like. As a starting point, we explore a simple sample script. We'll walk through this script and explain what each statement means. Then we'll describe the language components individually, each in its own section. We'll discuss Rexx variables, character strings, numbers, operators, and comparisons.

By the end of this chapter, you'll know about the basic components of the Rexx language. You'll be fully capable of writing simple scripts and will be ready to learn about the language features explored more fully in subsequent chapters. The chapters that follow present other aspects of the language, based on sample programs that show its additional features. For example, topics covered in subsequent chapters include directing the logical flow of a script, arrays and tables, input and output, string manipulation, subroutines and functions, and the like. But now, let's dive into our first sample script.

## A First Program

Had enough of your job? Maybe it's time to join the lucky developers who create computer games for a living! The complete Rexx program that follows is called the Number Game. It generates a random number between 1 and 10 and asks the user to guess it (well, okay, the playability is a *bit* weak. . . .) The program reads the number the user guesses and states whether the guess is correct.

```
/* The NUMBER GAME - User tries to guess a number between 1 and 10  */

/* Generate a random number between 1 and 10                        */

the_number = random(1,10)

say "I'm thinking of number between 1 and 10. What is it?"
```

```
pull the_guess

if the_number = the_guess then
   say 'You guessed it!'
else
   say 'Sorry, my number was: ' the_number

say 'Bye!'
```

Here are two sample runs of the program:

```
C:\Regina\pgms>number_game.rexx
I'm thinking of number between 1 and 10. What is it?
4
Sorry, my number was: 6
Bye!

C:\Regina\pgms>number_game.rexx
I'm thinking of number between 1 and 10. What is it?
8
You guessed it!
Bye!
```

This program illustrates several Rexx features. It shows that you document scripts by writing whatever description you like between the symbols `/*` and `*/`. Rexx ignores whatever appears between these *comment delimiters*. Comments can be isolated on their own lines, as in the sample program, or they can appear as *trailing comments* after the statement on a line:

```
the_number = random(1,10)  /* Generate a random number between 1 and 10 */
```

Comments can even stretch across multiple lines in *box style*, as long as they start with `/*` and end with `*/`:

```
/*************************************************************************
*  The NUMBER GAME - User tries to guess a number between 1 and 10      *
*  Generate a random number between 1 and 10                            *
*************************************************************************/
```

Rexx is *case-insensitive*. Code can be entered in lowercase, uppercase, or mixed case; Rexx doesn't care. The `if` statement could have been written like this if we felt it were clearer:

```
IF the_number = the_guess THEN
   SAY 'You guessed it!'
ELSE
   SAY 'Sorry, my number was: ' the_number
```

The variable named `the_number` could have been coded as `THE_NUMBER` or `The_Number`. Since Rexx ignores case it considers all these as references to the same variable. The one place where case *does* matter is within *literals* or hardcoded character strings:

```
       say 'Bye!'      outputs:      Bye!
```

while

```
          say 'BYE!'      displays:     BYE!
```

*Character strings* are any set of characters occurring between a matched set of either single quotation marks (`'`) or double quotation marks (`"`).

What if you want to encode a quote within a literal? In other words, what do you do when you need to encode a single or double quote as part of the character string itself? To put a single quotation mark within the literal, enclose the literal with double quotation marks:

```
say "I'm thinking of number between 1 and 10. What is it?"
```

To encode double quotation marks within the string, enclose the literal with single quotation marks:

```
say 'I am "thinking" of number between 1 and 10. What is it?'
```

Rexx is a *free-format language*. The spacing is up to you. Insert (or delete) blank lines for readability, and leave as much or as little space between instructions and their operands as you like. Rexx leaves the coding style up to you as much as a programming language possibly can.

For example, here's yet another way to encode the `if` statement:

```
IF the_number = the_guess THEN  SAY 'You guessed it!'
                          ELSE  SAY 'Sorry, my number was: ' the_number
```

About the only situation in which spacing is *not* the programmer's option is when encoding a Rexx *function*. A function is a built-in routine Rexx provides as part of the language; you also may write your own functions. This program invokes the built-in function `random` to generate a random number between 1 and 10 (inclusive). The parenthesis containing the function argument(s) must immediately follow the function name without any intervening space. If the function has no arguments, code it like this:

```
the_number = random()
```

Rexx requires that the parentheses occur *immediately after* the function name to recognize the function properly.

The sample script shows that one does not need to *declare* or predefine variables in Rexx. This differs from languages like C++, Java, COBOL, or Pascal. Rexx variables are established at the time of their first use. The variable `the_number` is defined during the assignment statement in the example. Space for the variable `the_guess` is allocated when the program executes the `pull` instruction to read the user's input:

```
pull  the_guess
```

In this example, the `pull` instruction reads the characters that the user types on the keyboard, until he or she presses the <ENTER> key, into one or more variables and automatically translates them to uppercase. Here the item the user enters is assigned to the newly created variable `the_guess`.

All variables in Rexx are variable-length character strings. Rexx automatically handles string length adjustments. It also manages numeric or data type conversions. For example, even though the variables `the_number` and `the_guess` are character strings, if we assume that both contain strings that represent numbers, one could perform arithmetic or other numeric operations on them:

```
their_sum  =  the_number  +  the_guess
```

Rexx automatically handles all the issues surrounding variable declarations, data types, data conversions, and variable length character strings that programmers must manually manage in traditional compiled languages. These features are among those that make it such a productive, high-level language.

## Language Elements

Rexx consists of only two dozen *instructions*, augmented by the power of some 70 *built-in functions*. Figure 2-1 below pictorially represents the key components of Rexx. It shows that the instructions and functions together compose the core of the language, which is then surrounded and augmented by other features. A lot of what the first section of this book is about is introducing the various Rexx instructions and functions.

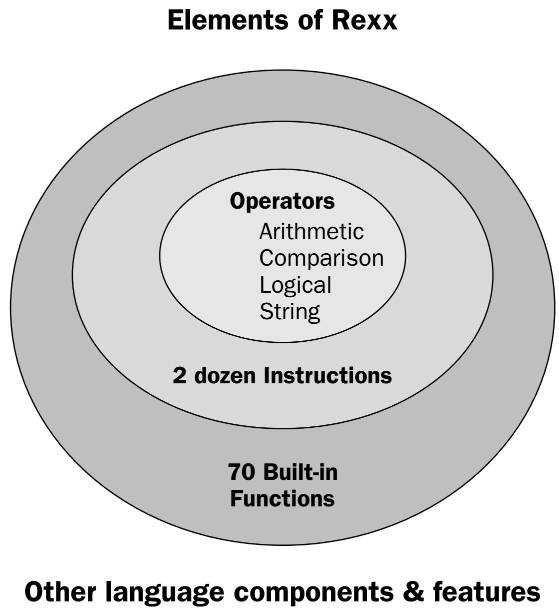

Figure 2-1

Of course, this book also provides a language reference section in the appendices, covering these and other aspects of the language. For example, Appendix B is a reference to all standard Rexx instructions, while Appendix C provides the reference to standard functions.

The first sample program illustrated the use of the instructions `say`, `pull`, and `if`. Rexx instructions are typically followed by one or more *operands*, or elements upon which they operate. For example, `say` is followed by one or more elements it writes to the display screen. The `pull` instruction is followed by a list of the data elements it reads.

The sample script illustrated one function, `random`. Functions are always immediately followed by parentheses, usually containing function *arguments*, or inputs to the function. If there are no arguments, the function must be immediately followed by empty parentheses `()` . Rexx functions always return a single result, which is then substituted into the expression directly in place of the function call. For example, the random number returned by the `random` function is actually substituted into the statement that follows, on the right-hand side of the equals sign, then assigned to the variable `the_number`:

```
the_number = random(1,10)
```

*Variables* are named storage locations that can contain values. They do not need to be declared or defined in advance, but are rather created when they are first referenced. You can declare or define all variables used in a program at the beginning of the script, but Rexx does not require this. Some programmers like to declare all variables at the top of their programs, for clarity, but Rexx leaves the decision whether or not to do this up to you.

All variables in Rexx are internally stored as variable-length strings. The interpreter manages their lengths and data types. Rexx variables are "typeless" in that their contents define their usage. If strings contain digits, you can apply numeric operations to them. If they do not contain strings representing numeric values, numeric operations don't make sense and will fail if attempted. Rexx is simpler than other programming languages in that developers do not have to concern themselves with data types.

*Variable names* are sometimes referred to as *symbols*. They may be composed of letters, digits, and characters such as `. ! ? _` . A variable name you create must not begin with a digit or period. A *simple variable name* does not include a period. A variable name that includes a period is called a *compound variable* and represents an *array* or *table.* Arrays will be covered in Chapter 4. They consist of groups of similar data elements, typically processed as a group.

If all Rexx variables are *typeless*, how does one create a numeric value? Just place a string representing a valid number into a Rexx variable. Here are *assignment statements* that achieve this:

```
whole_number_example            =  15
decimal_example                 =  14.2
negative_number                 = -21.2
exponential_notation_example    =  14E+12
```

A *number* in Rexx is simply a string of one or more digits with one optional decimal point anywhere in the string. Numbers may optionally be preceded by their sign, indicating a postive or a negative number. Numbers may be represented very flexibly by almost any common notation. Exponential numbers may be represented in either engineering or scientific notation (the default is scientific). The following table shows examples of numbers in Rexx.

| Number Type | Also Known As | Examples |
|---|---|---|
| Whole | Integer | '3'   '+6'   '9835297590239032' |
| Decimal | Fixed point | '0.3'  '17.36425' |
| Exponential | Real  --or-- | '1.235E+11'   (*scientific*, one digit left of decimal point) |
| | Floating point | '171.123E+11' (*engineering*, 1 to 3 digits left of decimal) |

Variables are assigned values through either assignment statements or input instructions. The assignment statement uses the equals sign (=) to assign a value to a variable, as shown earlier. The input instructions are the `pull` or `parse` instructions, which read input values, and the `arg` and `parse arg` instructions, which read command line parameters or input arguments to a script.

If a variable has not yet been assigned a value, it is referred to as *uninitialized*. The value of an uninitialized variable is the name of the variable itself in uppercase letters. This `if` statement uses this fact to determine if the variable `no_value_yet` is uninitialized:

```
if  no_value_yet = 'NO_VALUE_YET'  then
    say 'The variable is not yet initialized.'
```

*Character strings* or *literals* are any set of characters enclosed in single or double quotation marks ( `'` or `"` ).

If you need to include either the single or double quote within the literal, simply enclose that literal with the other *string delimiter*. Or you can encode two single or double quotation marks back to back, and Rexx understands that this means that one quote is to be contained within the literal (it knows the doubled quote does not terminate the literal). Here are a few examples:

```
literal= 'Literals contain whatever characters you like: !@#$%^&*()-=+~.<>?/_'
need_a_quote_mark_in_the_string = "Here's my statement."
same_as_the_previous_example    = 'Here''s my statement.'
this_is_the_null_string = ''  /*two quotes back to back are a "null string" */
```

In addition to supporting any typical numeric or string representation, Rexx also supports *hexadecimal* or base 16 numbers. *Hex strings* contain the upper- or lowercase letters `A` through `F` and the digits `0` through `9`, and are followed by an upper- or lowercase `X`:

```
twenty_six_in_hexidecimal = '1a'x  /*  1A is the number 26 in base sixteen      */
hex_string = "3E 11 4A"X           /*  Assigns a hex string value to hex_string */
```

Rexx also supports *binary*, or base two strings. Binary strings consist only of `0`s and `1`s. They are denoted by their following upper- or lowercase `B`:

```
example_binary_string = '10001011'b
another_binary_string = '1011'B
```

Rexx has a full complement of functions to convert between regular character strings and hex and binary strings. Do not be concerned if you are not familiar with the uses of these kinds of strings in programming languages. We mention them only for programmers who require them. Future chapters will explain their use more fully and provide illustrative examples.

## Operators

Every programming language has *operators*, symbols that indicate arithmetic operations or dictate that comparisons must be performed. Operators are used in calculations and in assigning values to variables, for example. Rexx supports a full set of operators for the following.

- ❑ Arithmetic
- ❑ Comparison
- ❑ Logical operators
- ❑ Character string concatenation

The arithmetic operators are listed in the following table:

| Arithmetic Operator | Use |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Integer division — returns the integer part of the result from division |
| // | Remainder division — returns the remainder from division |
| ** | Raise to a whole number power |
| + (as a prefix ) | Indicates a positive number |
| – (as a prefix) | Indicates a negative number |

All arithmetic operators work as one would assume from basic high-school algebra, or from programming in most other common programming languages. Here are a few examples using the less obvious operators:

```
say (5 % 2)   /* Returns the integer part of division result. Displays: 2   */
say (5 // 2)  /* Returns the remainder from division.      Displays: 1      */
say (5 ** 2)  /* Raises the number to the whole power.     Displays: 25     */
```

Remember that because all Rexx variables are strings, arithmetic operators should only be applied to variables that evaluate to valid numbers. Apply them only to strings containing digits, with their optional decimal points and leading signs, or to numbers in exponential forms.

Numeric operations are a major topic in Rexx (as in any programming language). The underlying principle is this — *the Rexx standard ensures that the same calculation will yield the same results even when run under different Rexx implementations or on different computers*. Rexx provides an exceptional level of machine- and implementation-independence compared with many other programming languages.

If you are familiar with other programming languages, you might wonder how Rexx achieves this benefit. Internally, Rexx employs decimal arithmetic. It does suffer from the approximations caused by languages that rely on floating point calculations or binary arithmetic.

The only arithmetic errors Rexx gives are *overflow* (or *underflow*). These result from insufficient storage to hold exceptionally large results.

To control the number of significant digits in arithmetic results, use the `numeric` instruction. Sometimes the number of significant digits is referred to as the *precision* of the result. Numeric precision defaults to nine digits. This sample statement illustrates the default precision because it displays nine digits to the right of the decimal place in its result:

```
say  2 / 3             /* displays  0.666666667  by default      */
```

This example shows how to change the precision in a calculation. Set the numeric precision to 12 digits by the `numeric` instruction, and you get this result:

```
numeric digits  12      /*  set numeric precision to 12 digits    */
say  2 / 3              /*  displays:  0.666666666667             */
```

Rexx preserves trailing zeroes coming out of arithmetic operations:

```
say  8.80 - 8           /*  displays: 0.80  */
```

If a result is zero, Rexx always displays a single-digit `0`:

```
say  8.80 - 8.80        /*  displays: 0     */
```

Chapter 7 explores computation further. It tells you everything you need to know about how to express numbers in Rexx, conversion between numeric and other formats, and how to obtain and display numeric results. We'll defer further discussion on numbers and calculations to Chapter 7.

*Comparison operators* provide for numeric and string comparisons. These are the operators you use to determine the equality or inequality of data elements. Use them to determine if one data item is greater than another or if two variables contain equal values.

Since every Rexx variable contains a character string, you might wonder how Rexx decides to perform a character or numeric comparison. The key rule is*: if both terms involved in a comparison are numeric, then the comparison is numeric*. For a numeric comparison, any leading zeroes are ignored and the numeric values are compared. This is just as one would expect.

If either term in a comparison is other than numeric, then a *string comparison* occurs. The rule for string comparison is that leading and trailing blanks are ignored, and if one string is shorter than the other, it is padded with trailing blanks. Then a character-by-character comparison occurs. String comparison is case-sensitive. The character string `ABC` is not equal to the string `Abc`. Again, this is what one would normally assume.

Rexx features a typical set of comparison operators, as shown in the following table:

| Comparison Operator | Meaning |
|---|---|
| `=` | Equal |
| `\=` `¬=` | Not equal |
| `>` | Greater than |
| `<` | Less than |
| `>=` `\<` `¬<` | Greater than or equal to, not less than |
| `<=` `\>` `¬>` | Less than or equal to, not greater than |
| `><` `<>` | Greater than or less than (same as not equal) |

The "not" symbol for operators is typically written as a backslash, as in "not equal:" `\=` But sometimes you'll see it written as ¬ as in "not equal:" `¬=` Both codings are equivalent in Rexx. The first representation is very common, while the second is almost exclusively associated with mainframe scripting. *Since most keyboards outside of mainframe environments do not include the symbol ¬ we recommend always using the backslash*. This is universal and your code will run on any platform. The backslash is the ANSI-standard Rexx symbol. You can also code "not equal to" as: `<>` or `><` .

In Rexx comparisons, if a comparison evaluates to `TRUE`, it returns 1. A `FALSE` comparison evaluates to 0. Here are some sample numeric and character string comparisons and their results:

```
'37'  = '37'    /*  TRUE  - a numeric comparison */
'0037'= '37'    /*  TRUE  - numeric comparisons disregard leading zeroes */
'37'  = '37   ' /*  TRUE  - blanks disregarded   */
'ABC' = 'Abc'   /*  FALSE - string comparisons are case-sensitive        */
'ABC' = '   ABC  ' /* TRUE- preceding & trailing blanks are irrelevant   */
''    = '     '    /* TRUE- null string is blank-padded for comparison   */
```

Rexx also provides for *strict comparisons* of character strings. *In strict comparisons, two strings must be identical to be considered equal*—leading and trailing blanks count and no padding occurs to the shorter string. Strict comparisons only make sense in string comparisons, not numeric comparisons. Strict comparison operators are easily identified because they contain doubled operators, as shown in the following chart:

| Strict Comparison Operator | Meaning |
|---|---|
| `==` | Strictly equal |
| `\==` `¬==` | Strictly not equal |
| `>>` | Strictly greater than |
| `<<` | Strictly less than |
| `>>=` `\<<` `¬<<` | Strictly greater than or equal to, strictly not less than |
| `<<=` `\>>` `¬>>` | Strictly less than or equal to, strictly not greater than |

Here are sample strict string comparisons:

```
'37'  == '37   '  /*  FALSE - strict comparisons include blanks         */
'ABC' >> 'AB'     /*  TRUE - also TRUE as a nonstrict comparison        */
'ABC' == '   ABC  '   /* FALSE - blanks count in strict comparison      */
''    == '   '        /* FALSE - blanks count in strict comparison      */
```

*Logical operators* are sometimes called *Boolean operators* because they apply *Boolean logic* to the operands. Rexx's logical operators are the same as the logical operators of many other programming languages. This table lists the logical operators:

| Logical Operator | Meaning | Use |
|---|---|---|
| `&` | Logical AND | TRUE if both terms are true |
| `\|` | Logical OR | TRUE if either term is true |
| `&&` | Logical EXCLUSIVE OR | TRUE if either (but not both) terms are true |
| `¬` or `\` (as a prefix) | Logical NOT | Changes TRUE to FALSE and vice versa |

Boolean logic is useful in `if` statements with multiple comparisons. These are also referred to as *compound comparisons*. Here are some examples:

```
if  ('A' = var1)  &  ('B' = var2) then
   say 'Displays only if BOTH comparisons are TRUE'

if  ('A' = var1)  |  ('B' = var2) then
   say 'Displays if EITHER comparison is TRUE'

if  ('A' = var1) &&  ('B' = var2) then
   say 'Displays if EXACTLY ONE comparison is TRUE'

if \('A' = var1)  then say 'Displays if A is NOT equal to var1'
```

*Concatenation* is the process of pasting two or more character strings together. Strings are appended one to the end of the other. *Explicitly* concatenate strings by coding the *concatenation operator* `||` . Rexx also automatically concatenates strings when they appear together in the same statement. Look at these instructions executed in sequence:

```
my_var = 'Yogi Bear'
say 'Hi there,'  ||  ' '  ||  my_var     /*  displays:  'Hi there, Yogi Bear'   */
say 'Hi there,'my_var                    /*  displays:  'Hi there,Yogi Bear'
                                                      no space after the comma */
say 'Hi there,'  my_var                  /*  displays:  'Hi there, Yogi Bear'
                                                     one space after the comma */
```

The second `say` instruction shows *concatenation through abuttal*. A literal string and a variable appear immediately adjacent to one another, so Rexx concatenates them without any intervening blank.

Contrast this to the last `say` instruction, where Rexx concatenates the literal and variable contents, but with one blank between them. If there are one or more spaces between the two elements listed as operands to the `say` instruction, Rexx places exactly one blank between them after concatenation.

Given these three methods of concatenating strings, individual programmers have their own preferences. Using the concatenation operator makes the process more explicit, but it also results in longer statements to build the result string.

Rexx has four kinds of operators: arithmetic, comparison, logical, and concatenation. And there are several operators in each group. If you build a statement with multiple operators, how does Rexx decide which operations to execute first? The order can be important. For example:

4 times 3, then subtract 2 from the result is 10

Perform those same operations with the same numbers in a different order, and you get a different result:

3 subtract 2, then multiple that times 4 yields the result of 4

Both these computations involve the same two operations with the same three numbers but the operations occur in different orders. They yield different results.

Clearly, programmers need to know in what order a series of operations will be executed. This issue is often referred to as the operator *order of precedence*. The order of precedence is a rule that defines which operations are executed in what order.

Some programming languages have intricate or odd orders of precedence. Rexx makes it easy. Its order of precedence is the same as in conventional algebra and the majority of programming languages. (The only minor exception is that the prefix *minus operator* always has higher priority than the exponential operator).

From highest precedence on down, this lists Rexx's order of precedence:

| | |
|---|---|
| ❑ Prefix operators | `+ - \` |
| ❑ Power operator | `**` |
| ❑ Addition and subtraction | `+ -` |
| ❑ Concatenation | by intervening blanks `\|\|` by abuttal |
| ❑ Comparison operators | `= == > < >= <=` ...and the others |
| ❑ Logical AND | `&` |
| ❑ Logical OR | `\|` |
| ❑ EXCLUSIVE OR | `&&` |

If the order of precedence is important to some logic in your program, an easy way to ensure that operations occur in the manner in which you expect is to simply enclose the operations to perform first in parentheses. When Rexx encounters parentheses, it evaluates the entire expression when that term is required. So, you can use parentheses to guarantee any order of evaluation you require. The more deeply nested a set of parentheses is, the higher its order of precedence. The basic rule is this: *when Rexx encounters expressions nested within parentheses, it works from the innermost to the outermost*.

To return to the earlier example, one can easily ensure the proper order of operations by enclosing the highest order operations in parentheses:

```
say  (4 * 3)  - 2        /* displays: 10 */
```

To alter the order in which operations occur, just reposition the parentheses:

```
say  4 * (3 - 2)         /* displays: 4  */
```

## Summary

This chapter briefly summarizes the basic elements of Rexx. We've kept the discussion high level and have avoided strict "textbook definitions." We discussed variable names and how to form them, and the difference between simple variable names and the compound variable names that are used to represent tables or arrays. We discussed the difference between strings and numbers and how to assign both to variables.

We also listed and discussed the operators used to represent arithmetic, comparison, logical, and string operations. We gave a few simple examples of how the operators are used; you'll see many more, real-world examples in the sample scripts in the upcoming chapters.

The upcoming chapters round out your knowledge of the language and focus in more detail on its capabilities. They also provide many more programming examples. Their sample scripts use the language elements this chapter introduces in many different contexts, so you'll get a much better feel for how they are used in actual programming.

## Test Your Understanding

1. How are comments encoded in Rexx? Can they span more than one line?
2. How does Rexx recognize a function call in your code?
3. Must variables be declared in Rexx as in languages like C++, Pascal, or Java? How are variables established, and how can they be tested to see if they have been defined?
4. What are the two instructions for basic screen input and output?
5. What is the difference between a *comparison* and *strict comparison*? When do you use one versus the other? Does one apply strict comparisons to numeric values?
6. How do you define a numeric variable in Rexx?