Introduction to String Manipulation

by Howard Fosdick

Overview

String manipulation means parsing, splicing, and pasting together *character strings*, sets of consecutive characters. Rexx excels at string manipulation. This is important for a wider variety of reasons than may be apparent at first. Many programming problems are readily conceived of as operations on strings. For example, building commands to issue to the operating system is a really a string-concatenation exercise. Analyzing the feedback from those commands once they are issued means text analysis and pattern matching. Much of the data formatting and reporting that IT organizations perform requires string processing. Data validation and cleansing require text analysis.

In a broad sense, many programming problems are essentially exercises in "symbol manipulation." *String processing* is a means to achieve generic symbol manipulation.

List processing is another example. Entire programming languages (such as LISP) have been built on the paradigm of processing lists. A list can be considered simply a group of values strung together. Manipulating character strings thus becomes a vehicle for list processing.

The applications that these techniques underlie are endless. Everything from report writing, to printing mailing labels, to editing documents, to creating scripts for systems administration, to scripts that configure the environment, rely on string manipulation.

This chapter introduces Rexx's outermost operators, functions, and pattern-matching capabilities. We show you the features by which Rexx supports string processing so that you will combine them in new ways to address the programming problems you face.

Concatenation and Parsing

Concatenation is the joining together of strings into larger strings. *Bifurcation* refers to splitting a string into two parts. *Parsing* is the inspection of character strings, to analyze them, extract pieces, or break them into components. For example, parsing a U.S. telephone number could separate it

[This tutorial was excerpted from Chapter 28 of the *Rexx Programmer's Guide*, Wiley, © 2005]

into its constituent parts — a country code, an area code, the prefix and suffix. *Pattern matching* is the scanning of strings for certain patterns. Together, these operations constitute *string manipulation* or *text processing*. Figure 6-1 summarizes the major string operations.

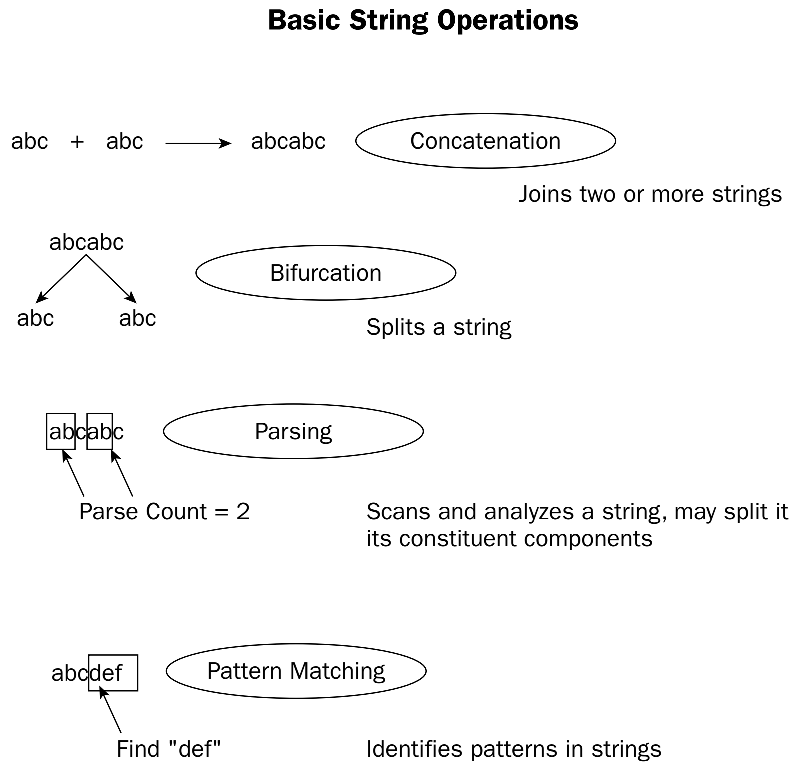

Figure 6-1

We've already seen that Rexx supports three ways of concatenating strings. These are:

- *Implicit concatenation* with one blank between the symbols
- *Abuttal*, in which immediately adjacent symbols are concatenated without an intervening blank
- *Explicit concatenation* via the concatenation operator, ||

The three styles of concatenation can be intermixed within statements. Concatenation may occur wherever expressions can be coded. Here are some sample statements run in sequence:

```
apple='-Apple'
say  'Candy' || ' ' || apple || ' ' || 'Rodeo'
                                 /* displays: 'Candy -Apple Rodeo'          */
say  'Candy'apple                /* displays: 'Candy-Apple'                 */
say  'Candy' apple               /* displays: 'Candy -Apple'                */
say  'Candy'apple apple 'Rodeo'  /* displays: 'Candy-Apple -Apple Rodeo     */
```

We've also seen several simple examples of string parsing. The `arg` instruction retrieves the arguments sent in to a program or internal function and places them into a list of variables. Its general format is:

```
        arg  [template]
```

The *template* is a list of symbols separated by blanks and/or patterns. The `pull` instruction operates in the same manner as `arg`, reading and parsing a string input by the user into a list of variables. The input string is parsed (separated) into the variables in the list, positionally from leftmost to rightmost, as separated by one or more spaces. The spaces delimiting the strings are stripped out, and the variables do not contain any leading or trailing blanks.

There are two special cases to consider when a script reads and parses input by the `arg` or `pull` instructions. The first is the situation in which more arguments are passed in to the routine than the routine expects. Look at this case:

```
user input:   one  2  three  '4'
program:      pull a b c

a contains: ONE
b contains: 2
c contains: THREE  '4'
```

The last (rightmost) variable `c` in the variable list contains all remaining (unparsed) information. The rule is: *If you code too few input variables to hold all those that are input, the final variable in the input list contains the extra information.* Remember that you could just ignore this extra information by coding a period:

```
program:     pull  a  b  c  .
```

Now the variables will contain:

```
a contains: ONE
b contains: 2
c contains: THREE
```

The `'4'` is simply eliminated from the input by the *placeholder variable*, the period at the end of the `pull` instruction input list or *template*.

The second situation to consider is if too few arguments are passed in to the receiving routine. Say that the script issues a `pull` instruction to read input from the user. If too few elements are input by the user, any variables in the list that cannot be assigned values are set to null:

```
user input: one  2
program:    pull a b c

a contains: ONE
b contains: 2
c contains: ''        /*  c is set to the null string  */
```

Variable `c` is set to the null string (represented by back-to-back quotation marks, `''`). This is different from saying that the variable is uninitialized, which would mean its value is its own name in uppercase. If the last variable were uninitialized, it would be set to '`C`'.

`pull` is short for the instruction:

```
        parse  upper  pull  [template]
```

The *template* is a list of symbols separated by blanks and/or patterns. `upper` means uppercase translation occurs. Its presence is optional on the `parse` instruction. To avoid uppercase translation, just leave the `upper` keyword out of the `parse` instruction.

Let's look at the `parse` instruction in more detail. This form of the instruction parses an expression:

```
        parse  [upper]  value  [expression]  with  [template]
```

The *expression* evaluates to some string that is parsed according to the *template*. The template provides for three basic kinds of parsing:

- ❑ By words (character strings delimited by blanks or spaces)
- ❑ By pattern (one character or a string other than blanks by which the expression string will be analyzed and separated)
- ❑ By numeric pattern (numbers that specify column starting positions for each substring within the expression)

Figure 6-2 below illustrates these three parsing methods.

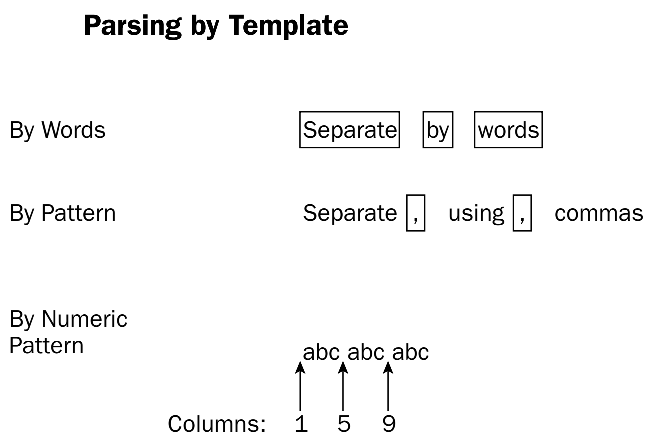

Figure 6-2

You are already familiar with parsing by words. This is where we use `parse` to separate a list of elements into individual components based on intervening blanks. Let's parse an international telephone number as an example.

```
phone = '011-311-458-3758'
parse  value  phone  with  a  b
```

This is a parse by *words* or blank separators. Since there are no blank separators anywhere within the input string, the results of the `parse` instruction are:

```
a = 011-311-458-3758
b = ''             /* b is assigned the null string. */
```

Obviously, the dash (–) here is the separator, not the blank. Let's try parsing by *pattern*, using the dash (–) as the separator or *delimiter*:

```
parse value phone with country_code  '-'  area_code  '-'  prefix  '-'  suffix
```

The results are:

```
country_code = 011
area_code = 311
prefix = 458
suffix = 3758
```

If there were more information in the input variable, regardless of whether or not it contained more dash delimiters, it all would have been placed into the last variable in the list, `suffix`. If there are too few strings in the input variable list, according to the parsing delimiter, then extra variables in the variable list are assigned null string(s).

The *pattern* can be supplied in a variable. This yields greater programmability and flexibility. In this case, enclose it in parentheses when specifying it in the template:

```
sep = '-'         /* the dash will be the delimiter ... */
parse value phone with country_code (sep) area_code (sep) prefix (sep) suffix
```

This `parse` instruction gives the same results as the previous one with the hardcoded delimiter dashes. The advantage to placing the separator pattern in a variable is that we can now parse a different, international designation for this phone number using the same `parse` instruction, just by changing the separator inside the pattern variable:

```
phone = '011.311.458.3758'
sep = '.'        /* The period is the Swiss delimiter for phone numbers  ... */
parse value phone with country_code (sep) area_code (sep) prefix (sep) suffix
```

The same `parse` instruction properly separates the constituent pieces of the phone number with this different delimiter. So, supplying the separator pattern in a variable gives scripts flexibility in parsing operations.

Now parse by *numbers*. These represent *column positions* in the input. Run:

```
phone = '011-311-458-3758'
parse  value  phone  with country_code 5 area_code  9 prefix 13 suffix
```

Here are the results from this statement:

```
country_code = 011-
area_code = 311-
prefix = 458-
suffix = 3758
```

Oops! You can see that parsing by numbers goes strictly by column positions. Delimiters don't count. Add these extra columns positions to eliminate the unwanted separators:

```
parse value phone with country_code  4  5  area_code  8  9 prefix  12  13  suffix
```

This gives the intended results because it parses out the unwanted separators by column positions:

```
country_code = 011
area_code = 311
prefix = 458
suffix = 3758
```

These are *absolute* column positions. Each refers to an absolute column position, counting from the beginning of the string.

Placing a plus (+) or minus (–) sign before any number makes its position *relative* to the previously specified number in the list (or 1 for the first number). You can mix absolute and relative positions together in the same template, and even use negative numbers (which move the relative position backwards to the left) but be careful. Unless you have a situation that really requires it, jamming all the parsing into one complex statement is rarely worth it. Just code a series of two or three simpler statements instead. Then others will be able to read and understand your code.

This example properly parses the phone number with both absolute and relative column numbers. The plus signs (+) indicate relative numbers. In this case, each advances the column position one character beyond the previous absolute column indicator:

```
parse value phone with country_code 4  +1 area_code 8  +1 prefix 12  +1 suffix
```

This statement produces the desired result:

```
country_code = 011
area_code = 311
prefix = 458
suffix = 3758
```

With this background, you can see that the `parse` instruction provides real string-processing power. This example assigns the entire telephone number in the variable `phone` to three new variables (kind of like a three-part assignment statement):

```
parse value phone with phone_1  1  phone_2  1  phone_3
```

Now the variables `phone_1`, `phone_2`, and `phone_3` all contain the same value as `phone`:

```
phone   = '011.311.458.3758'

phone_1 = '011.311.458.3758'
phone_2 = '011.311.458.3758'
phone_3 = '011.311.458.3758'
```

In all the examples thus far, the input string was not changed. But it can be if encoded as part of the variable list. Here's an example. Say that we have this variable:

```
employee_name = 'Deanna Troy'
```

This statement simply translates the employee's name into uppercase and places it back into the same variable:

```
parse  upper  value  employee_name  with  employee_name
```

This statement strips off the employee's first name and places it into the variable `first_name`. Then it puts the remainder of the name back into the `employee_name` variable:

```
parse  value  employee_name  with  first_name  employee_name
```

The `value` keyword refers to any expression. You may also see the keyword `var` encoded when referring specifically to a variable. In this case, you should not code the `with` keyword. This statement using `var` gives the exact same results as the previous example with `value` and `with`:

```
parse  var employee_name  first_name  employee_name
```

A Sample Program

With this introduction to parsing, here's a sample program to illustrate parsing techniques. This script preprocesses the "load file" used to load data into a relational database such as DB2, Oracle, SQL Server, or MySQL. The script performs some simple data verification on the input file prior to loading that data into the database. This "data-cleansing" script ensures the data we load into the database is clean before we run the database load utility. A script like this is useful because the data cleansing that database utilities typically perform is limited.

Here's how the data will look after it's loaded into the relational table:

EMP_NO	FNAME	LNAME	DEPT_NO
10001	George	Bakartt	307
10002	Bill	Wall	204
10003	Beverly	Crusher	305

Databases like DB2, Oracle, and SQL Server accept input data in several different file formats. Two of the most popular are *comma-delimited files* and *record-oriented* or *column-position files*. Here's an example of a *comma-delimited file*:

```
10001,"George","Bakartt","307"
10002,"Bill","Wall","204"
10003,"Beverly","Crusher","305"
1x004,"joe","Zip","305"
10005,"Sue","stans","3x5"
```

Commas separate the four input fields. In this example, all character strings are enclosed in double quotation marks. Under operating systems that employ a file type, the file type for comma-delimited ASCII files is typically `*.del`. This input file is named `database_input.del`.

Here is the other kind of file, a *record file*. Data fields start in specific columns. Fields are padded with blanks, as necessary, so that the next field starts in its required column. Where file types are used this file is typically of extension `*.asc`, so we've named this file `database_input.asc`:

```
10001George Bakartt307
10002Bill   Wall   204
10003BeverlyCrusher305
1x004joe    Zip    305
10005Sue    stans  3x5
```

The program reads either of these two input file types. It determines which kind of file it is processing by scanning the input text for commas. If the data contains commas, the program assumes it is dealing with a comma-delimited ASCII file.

Then the program performs some simple data verification. It ensures that the `EMP_NO` and `DEPT_NO` data items are numeric, and that the first and last names both begin with capital letters. The script writes any errors it finds to the display. Here's the program:

```
/*  DATABASE INPUT VERIFICATION:                                      */
/*                                                                    */
/*     Determines type of database input file (*.del or *.asc).       */
/*     Reads the input data as appropriate to that file type.         */
/*     Verifies EMP_NO and DEPT_NO are numeric, names are cap alpha.  */

arg input_file .                /* read input filename from user      */
c = ','                            /* variable C contains one comma   */

do while lines(input_file) > 0
   input_line = linein(input_file)    /* read a line from input file  */

   /* get EMP_NO, FNAME, LNAME, DEPT_NO from *.DEL or *.ASC file      */

   if pos(c,input_line) > 0 then do       /* File is delimited ASCII. */
      parse value input_line with emp_no (c) fname (c) lname (c) dept_no
      fname   = strip(fname,B,'"')
      lname   = strip(lname,B,'"')        /* remove quote " marks     */
      dept_no = strip(dept_no,,'"')
      end
      else do
         parse value input_line with emp_no 6 fname 13 lname 20 dept_no
         fname   = strip(fname)
         lname   = strip(lname)              /* remove trailing blanks  */
      end

      say 'Input line:' emp_no fname lname dept_no

      /* Ensure EMP_NO & DEPT_NO are numeric */

      if datatype(emp_no) \= 'NUM'  |  datatype(dept_no) \= 'NUM' then
         say 'EMP_NO or DEPT_NO are not numeric:' emp_no dept_no

      /* Ensure the two names start with a capital letter */

      if verify(substr(fname,1,1),'ABCDEFGHIJKLMNOPQRSTUVWXYZ') > 0 then
         say "First name doesn't start with a capital letter:" fname
      if verify(substr(lname,1,1),'ABCDEFGHIJKLMNOPQRSTUVWXYZ') > 0 then
         say "Last name doesn't start with a capital letter:" lname

   end
```

So that we can easily feed it either kind of file to process, the script accepts the filename as an input parameter. This technique of reading the name of the file to process from the command line is common. It offers more flexibility than "hardcoding" the filename into the script.

To start off, the script reads the first line of input data and determines whether it is processing a comma-delimited input file or a record-oriented file by this code:

```
if pos(c,input_line) > 0 then do      /* file is delimited ascii */
```

The `pos` built-in function returns the character position of the comma (represented by the variable `c`) within the target string. If the returned value is greater than `0`, a comma is present in the input line, and the program assumes that it is dealing with comma-delimited input. If the script finds no comma in the input line, it assumes that it is dealing with a record-oriented input file.

If the program determines that it is working with a comma-delimited input file, it issues this `parse` instruction to split the four fields from the input line into their respective variables:

```
parse value input_line with emp_no (c) fname (c) lname (c) dept_no
```

This `parse` statement strips data elements out of the input string according to comma delimiters. But there is a problem. The second, third, and fourth data elements were enclosed in double quotation marks in the input file. To remove these leading and trailing quotation marks, we use the built-in `strip` function:

```
fname   = strip(fname,B,'"')
lname   = strip(lname,B,'"')           /* remove quote " marks    */
dept_no = strip(dept_no,,'"')
```

The `B` operand stands for Both — strip out *both* leading and trailing double quotation marks. Other `strip` function options are `L` for *leading* only and `T` for *trailing* only. `Both` is the default, so as the third line in the previous example shows, we don't need to explicitly code it. Instead, we just show that parameter is missing by coding two commas back-to-back. The final parameter in the `strip` function encloses the character to remove within quotation marks. Here we enclosed the double quotation marks (`"`) within two single quotation marks, so that `strip` will remove double quotation marks from the variable's contents.

If the script does not find a comma in the input line, it assumes that it is dealing with a file whose data elements are located starting in specific columns. So, the script employs a *parse by number* statement, where the numbers specify column starting positions:

```
parse value input_line with emp_no 6 fname 13 lname 20 dept_no
```

If you program in languages like COBOL or Pascal, you might recognize this as what is often referred to as *record I/O*. Languages like C, C++, and C# call this an I/O *structure*, or *struct*. Chapter 5 showed that Rexx's stream I/O model is simple, yet you can see that it is powerful enough to easily perform record I/O by parsing the input in this manner. Part of the beauty of Rexx is that it is so easy to perform such operations, without needing special syntax or hard-to-code features in the language to accomplish them.

After the parsing by number, the record input may contain trailing blanks for the two names, so these statements remove them:

```
fname   = strip(fname)
lname   = strip(lname)              /* remove trailing blanks  */
```

Now that it has decoded the file and normalized the data elements, the program can get to work and verify the data contents. This statement uses the `datatype` built-in function to verify that the `EMP_NO` and `DEPT_NO` fields (the first and last data elements in each input record) are numeric. If `datatype` does not return the character string `NUM`, then one of these fields is not numeric and an error message is displayed:

```
if datatype(emp_no) \= 'NUM'  |  datatype(dept_no) \= 'NUM' then
   say 'EMP_NO or DEPT_NO are not numeric:' emp_no dept_no
```

The `logical or` (|) is used to test both data elements in one `if` instruction. If either is not numeric, the error message is displayed.

Finally, the script uses the `verify` built-in function to ensure that the two names both start with a capital letter. First, this nested use of the `substr` built-in function returns the first letter of the name:

```
substr(fname,1,1)
```

Then the `verify` function tests this letter to ensure that it's a member of the string consisting of all capital letters:

```
if verify(substr(fname,1,1),'ABCDEFGHIJKLMNOPQRSTUVWXYZ') > 0 then
   say "First name doesn't start with a capital letter:" fname
```

The *nesting* of the `substr` function means that we have coded one function (`substr`) within another (`verify`). Rexx resolves the innermost function first. The result of the innermost function is then plunked right into the code at the position formerly occupied by that function. So, the `substr` function returns the first letter of the variable `fname`, which then becomes the first parameter within the parentheses for the `verify` function.

Pretty nifty, eh? Rexx allows you to nest functions to an arbitrary depth. We do not recommend nesting beyond a single level or else the code can become too complicated. We'll provide an example of deeper nesting (and how it becomes complicated!) later in this chapter.

It's easy to code for *intermediate results* by breaking up the nesting into two (or more) statements. This example shows how to eliminate the nested function to simplify the code. It produces the exact same result as our nested example:

```
first_letter = substr(fname,1,1)
if verify(first_letter,'ABCDEFGHIJKLMNOPQRSTUVWXYZ') > 0 then
```

After the script runs, here is its output for the sample data we viewed earlier:

```
D:\Regina\hf>regina database_input.rexx database_input.asc
Input line: 10001 George Baklarz 307
Input line: 10002 Bill Wong 304
Input line: 10003 Beverly Crusher 305
Input line: 1x004 joe Zip 305
EMP_NO or DEPT_NO are not numeric: 1x004 305
First name doesn't start with a capital letter: joe
Input line: 10005 Sue stans 3x5
EMP_NO or DEPT_NO are not numeric: 10005 3x5
Last name doesn't start with a capital letter: stans
```

The last two lines of the input data contained several errors. Parsing techniques and string functions together enabled the program to identify these errors.

String Functions

The `parse` instruction provides syntactically simple, but operationally sophisticated parsing. You can resolve many string-processing problems with it. Rexx also includes over 30 string-manipulation functions, a few of which the sample script above illustrates.

This section describes more of the string functions. A later section in this chapter discusses the eight outermost functions that are *word-oriented*. The *word-oriented functions* process strings on the basis of words, where a *word* is defined as a character string delimited by blanks or spaces. For example, this string consists of a list of 16 words:

```
now is the time for all good men to come to the aid of their country
```

Before we proceed, here is a quick summary of Rexx's string functions (see Appendix C for full coding details of these and all other Rexx functions):

- ❑ `abbrev` — Tells if one string is equal to the first characters of another
- ❑ `center` — Centers a string within blanks or other *pad* characters

- ❑ `changestr` — Changes all occurrences of one string within another to a specified string
- ❑ `compare` — Tells if two strings are equal (like using the = operator)
- ❑ `copies` — Returns a string concatenated to itself n times
- ❑ `countstr` — Counts how many times one string appears within another
- ❑ `datatype` — Verifies string contents based on a variety of "data type" tests
- ❑ `delstr` — Deletes a substring from within a string
- ❑ `insert` — Inserts one string into another
- ❑ `lastpos` — Returns the last occurrence of one string within another
- ❑ `left` — Returns the first n characters of a string, or it can left-justify a string
- ❑ `length` — Returns the length of a string
- ❑ `overlay` — Overlays one string onto another starting at a specified position in the target
- ❑ `pos` — Returns the position of one string within another
- ❑ `reverse` — Reverses the characters of a string
- ❑ `right` — Returns the last n characters of a string, or it can right-justify a string
- ❑ `strip` — Strips leading and/or trailing blanks (or other characters) from a string
- ❑ `substr` — Returns a substring from within a string
- ❑ `translate` — Transforms characters of a string to another set of characters, as directed by two "translation strings"
- ❑ `verify` — verifies that all characters in a string are part of some defined set
- ❑ `xrange` — Returns a string of all valid character encodings

The `changestr` and `countstr` functions were added by the ANSI-1996 standard. Rexx implementations that meet the TRL-2 standard of 1990 but not the ANSI-1996 standard may not have these two functions. This is one of the few differences between the TRL-2 and ANSI-1996 standards (which are fully enumerated in Chapter 13). Regina Rexx fully meets the ANSI-1996 standard and includes these two functions.

Here's a simple program that demonstrates the use of the `abbrev`, `datatype`, `length`, `pos`, `translate`, and `verify` string functions. The script reads in four command-line arguments and applies data verification tests to them. The script displays any inaccurate parameters.

```
/*  VERIFY ARGUMENTS:                                               */
/*                                                                  */
/*     This program verifies 4 input arguments by several criteria.  */

parse arg  first  second  third  fourth  .   /* get the arguments    */

/* First parm must be a valid abbreviation for TESTSTRING            */

if abbrev('TESTSTRING',first,4) = 0 then
```

```
      say 'First parm must be a valid abbreviation for TESTSTRING:' first

   /* Second parm must consist only of digits and be under 5 bytes long */

   if datatype(second) \= 'NUM' then
      say 'Second parm must be numeric:' second
   if length(second) > 4 then
      say 'Second parm must be under 5 bytes in length:' second

   /* Third parm must occur as a substring somewhere in the first parm  */

   if pos(third,first) = 0 then
      say 'Third parm must occur within the first:' third first

   /* Fourth parm translated to uppercase must contain only letters ABC */

   if fourth = '' then
      say 'You must enter a fourth parameter, none was entered'
   uppercase = translate(fourth)     /* translate 4th parm to uppercase */
   if verify(uppercase,'ABC') > 0 then
      say 'Fourth parm in uppercase contains letters other than ABC:' fourth
```

Here's an example of running this program with parameters it considers correct:

```
   c:\Regina\pgms> regina  verify_arguments  TEST  1234  TEST  abc
```

Here's an example where incorrect parameters were input:

```
   c:\Regina\pgms>regina verify_arguments TEXT 12345 TEST abcdef
   First parm  must be a valid abbreviation for TESTSTRING: TEXT
   Second parm must be under 5 bytes in length: 12345
   Third parm must occur within the first: TEST TEXT
   Fourth parm in uppercase contains letters other than ABC: abcdef
```

Let's discuss the string functions this code illustrates.

The first parameter must be a valid abbreviation for a longer term. Where would you use this function? An example would be a program that processes the commands that a user enters on a command line. The system must determine that the abbreviation entered is both valid and that it uniquely specifies which command is intended. The `abbrev` function allows you to specify how many characters the user must enter that match the beginning of the target string. Here, the user must enter at least the four letters `TEST` for a valid match:

```
   if abbrev('TESTSTRING',first,4) = 0 then
      say 'First parm must be a valid abbreviation for TESTSTRING:' first
```

The second parameter the user enters must be numeric (it must be a valid Rexx number). The `datatype` function returns the string `NUM` if this is the case, otherwise it returns the string `CHAR`:

```
   if datatype(second) \= 'NUM' then
      say 'Second parm must be numeric:' second
```

datatype can also be used to check for many other conditions, for example, if a string is alphanumeric, binary, lowercase, mixed case, uppercase, a whole number, a hexadecimal number, or a valid symbol.

Using the length function allows the program to determine if the second parameter contains more than four characters:

```
if length(second) > 4 then
   say 'Second parm must be under 5 bytes in length:' second
```

The third parameter must be a substring of the first parameter. The pos function returns the starting position of a substring within a string. If the substring does not occur within the target string, it returns 0:

```
if pos(third,first) = 0 then
   say 'Third parm must occur within the first:' third first
```

This code ensures that the user entered a fourth parameter. If a fourth parameter was not entered, the argument will have been set to the null string (represented by the two immediately adjacent single quotation marks):

```
if fourth = '' then
   say 'You must enter a fourth parameter, none was entered'
```

Finally, when translated to uppercase, the fourth parameter must not contain any letters other than A, B, or C. Using the translate function with a single parameter translates the fourth argument to uppercase:

```
uppercase = translate(fourth)      /* translate 4th parm to uppercase */
```

Use the verify function to ensure that all characters in a string are members of some set of characters. This verify statement ensures that all the characters in the string named uppercase are members of its second parameter, hardcoded here as the literal string ABC. If this is not the case, the verify function returns the position of the first character violating the rule:

```
if verify(uppercase,'ABC') > 0 then
   say 'Fourth parm in uppercase contains letters other than ABC:' fourth
```

The Rexx string functions are pretty straightforward. This script shows how easy it is to use them to perform data verification and for basic string processing.

The Word-Oriented Functions

A *word* is a group of printable characters surrounded by blanks or spaces. A word is a blank-delimited string. Rexx offers a group of *word-oriented functions*:

- ❑ delword — Deletes the *n*th word(s) from a string
- ❑ space — Formats words in a string such that they are separated by one or more occurrences of a specified pad character

- ❑ `subword` — Returns a *phrase* (substring) of a string that starts with the *n*th word
- ❑ `word` — Returns the *n*th word in a string
- ❑ `wordindex` — Returns the character position of the *n*th word in a string
- ❑ `wordlength` — Returns the length of the *n*th word in a string
- ❑ `wordpos` — Returns the word position of the first word of a phrase (substring) within a string
- ❑ `words` — Returns the number of words in a string

These functions can be coupled with the outermost functions to address any number of programming problems in which symbols are considered as strings of words. One such area is *textual analysis* or *natural language processing*. An example of a classic text analysis problem is to confirm the identity of the great English playwright Shakespeare. Were all his works written by one person? Could they have been written by one his better-known contemporaries?

One way to answer these questions is to analyze Shakespeare's works and look for word-usage patterns. Humans tend to use words in consistent ways. (Some experts claim they can analyze word usage to the degree that individuals' *linguistic profiles* are unique as their fingerprints). Analyzing Shakespeare's texts and comparing them to those of contemporaries indicates whether Shakespeare's works were actually written by him or someone else.

Special-purpose languages such as SNOBOL are particularly adept at natural language processing. But SNOBOL is premodern; it lacks good control constructs and robust I/O. Better to use a more mainstream, portable, general-purpose language like Rexx that offers strong string manipulation in the context of good structure.

Text analysis is a complex topic outside the scope of this book. But we can present a simple program that suggests how Rexx can be applied to textual analysis. The script named Poetry Scanner reads modern poetry and counts the number of articles and prepositions in the input. It produces a primitive form of "sophistication rating" or *lexical density*. In our example, this rating comprises two ratios: the ratio of the number of longer words to the number of shorter words, and the ratio of prepositional words to the total number of words in the text.

To perform these operations, the script translates the input text to all uppercase and removes punctuation, because punctuation represents extraneous characters that are irrelevant to the analysis.

For this input poem:

```
"The night was the darkest,
for the byrds of love were flying. And lo!
I   saw   them   with the eyes of the eagle.
above
    the cows   flew   in the cloud pasture.
below
    the  earthworms  were  multiplying ...
god grant that they all find their ways home."
```

. . . the program produces this output:

```
THE NIGHT WAS THE DARKEST
FOR THE BYRDS OF LOVE WERE FLYING AND LO
I SAW THEM WITH THE EYES OF THE EAGLE
ABOVE
THE COWS FLEW IN THE CLOUD PASTURE
BELOW
THE EARTHWORMS WERE MULTIPLYING
GOD GRANT THAT THEY ALL FIND THEIR WAYS HOME

Ratio long/short words:  0.40625
Number of articles:    8
Number of prepositions: 5
Ratio of preps/total words: 0.111111111

Press ENTER key to exit...
```

Here is the program:

```
/*  POETRY SCANNER:                                                  */
/*                                                                   */
/*     This program scans text to perform primitive text analysis.   */

list_of_articles = 'A AN THE'
list_of_preps    = 'AT BY FOR FROM IN OF TO WITH'

big_words       = 0   ;   small_words  = 0
number_articles = 0   ;   number_preps = 0

do while lines('poetry.txt') > 0
   line_str = linein('poetry.txt')  /* read a line of poetry         */
   line_str = translate(line_str)   /* translate to uppercase        */
   line_str = translate(line_str,'      ','.,!:;"') /* remove punc.  */
   call lineout ,space(line_str)    /* display converted input line  */

   do j=1 to words(line_str)        /* do while a word to process    */
      if wordlength(line_str,j) >= 5 then
          big_words = big_words + 1             /* count big words   */
      else
          small_words = small_words + 1         /* count small words */
      word_to_analyze = word(line_str,j)        /* get the word      */
      if wordpos(word_to_analyze,list_of_articles) > 0 then
          number_articles = number_articles + 1 /* count the articles*/
      if wordpos(word_to_analyze,list_of_preps) > 0 then
          number_preps = number_preps + 1       /* count prep phrases*/
   end
end
say
say 'Ratio long/short words: ' (big_words/small_words)
say 'Number of articles:   ' number_articles
say 'Number of prepositions:' number_preps
say 'Ratio of preps/total words:' (number_preps/(big_words+small_words))
```

The program demonstrates several of the word-oriented functions, including `words`, `word`, `wordlength`, and `wordpos`. It also uses the `translate` function in two different contexts.

After it reads a line of input, the program shows how the `translate` function can be used with only the input string as a parameter to translate the contents of the string to all uppercase letters:

```
line_str = translate(line_str)           /* translate to uppercase       */
```

Then `translate` is used again, this time to replace various punctuation characters with blanks. In this call, the third parameter to `translate` contains the characters to translate, and the second parameter tells what characters to translate them to. This example translates a various punctuation characters into blanks:

```
line_str = translate(line_str,'      ','.,!:;"')         /* remove punc. */
```

The `do` loop processes the individual words in each input line. It executes while there is a word to process in the current input line:

```
do j=1 to words(line_str)               /* do while a word to process     */
```

The `words` function returns the number of blank-delimited words in the input line, `line_str`.

The `wordlength` function tells the length of the word. The script uses it to determine whether the word is longer than 4 bytes:

```
if wordlength(line_str,j) >= 5 then
   big_words = big_words + 1             /* count big words   */
```

The script needs to get an individual word in order to determine if that word is an article or preposition. To parse out one word from the input string, the script invokes the `word` function:

```
word_to_analyze = word(line_str,j)             /* get the word  */
```

To identify articles in the text, the program initializes a string containing the articles:

```
list_of_articles = 'A AN THE'
```

Then it uses the `wordpos` function to see if the word being inspected occurs in this list of articles. `wordpos` returns the starting position of the word in a string if it occurs in the string. If it returns `0`, we know that the word is not an article:

```
if wordpos(word_to_analyze,list_of_articles) > 0 then
   number_articles = number_articles + 1           /* count the articles*/
```

What this line of code really does is *list processing*. It determines if a given element occurs in a list. String processing is easily used to emulate other kinds of processing techniques and various data structures, such as the *list*. As mentioned in the chapter introduction, string manipulation is powerful because it is a generic tool that can easily be used to implement other processing paradigms.

The program ends with several `say` instructions that show how output can be dynamically concatenated from the results of expressions. The last line of the program calculates a ratio and displays it with an appropriate label:

```
say 'Ratio of preps/total words:' (number_preps/(big_words+small_words))
```

Rexx evaluates the expression in parentheses prior to executing the `say` instruction and displaying the output line. Remember that in evaluating expressions, Rexx always works from the innermost set of parentheses on out. The script uses the parentheses to ensure that this expression is resolved first:

```
(big_words+small_words)
```

The result of this expression feeds into the division:

```
(number_preps/(big_words+small_words))
```

To summarize, this simple program illustrates a number of the word and string functions. More importantly, it demonstrates that these features can be combined to create powerful string-processing scripts. Rexx offers excellent string-processing facilities.

The Bit String Functions and Conversions

The TRL-2 standard added support for *bit strings*, strings that represent binary values. Bit strings are composed solely of 0s and 1s. They are represented as a string of 0s and 1s immediately followed by the letter `b` or `B`:

```
'11110000'b        /* represents one character (or "byte") as a bit string */
```

This encoding parallels that used to represent *hexadecimal* (or *hex*) strings. Hex is the base-16 arithmetic system by which computer bits are represented. Each character or byte is represented by two hex digits. Hex strings are composed of the digits `0` thru `9` and letters `A` thru `F`, immediately followed by the letter `x` or `X`:

```
'0D0A'x            /* the two byte end-of-line indicator in Windows and DOS */
```

Binary strings find several uses. For example, use them to specify characters explicitly, bit by bit. This helps you store and manipulate unprintable characters, for example. The relationship of bit strings to characters is described by the table called a *character map*. Sometimes this is referred to as the *character encoding scheme*.

Want to see your system's entire character map? Just enter the `xrange` function:

```
 say  xrange()      /* displays the character map */
```

Or display some portion of the character map by specifying a range of starting and ending points. The range can be expressed in binary, hex, or character. You'll see the entire map, just as shown earlier, if you enter the entire range of the map explicitly:

```
say xrange('00'x,'FF'x)                   /* displays the character map */
```

This statement also displays the entire character range:

```
say xrange('00000000'b,'11111111'b)       /* displays the character map */
```

Display the same character map in hex (base-16) by using the `c2x` (character-to-hex) conversion function:

```
say  c2x(xrange())                 /* displays the character map in hex */
```

Want to see it as a bit string? You'll have to do two conversions: character to hex, then hex to binary. Nest the character-to-hex (`c2x`) function within the hex-to-binary (`x2b`) function to do this. Remember, Rexx always evaluates the expression nested in the innermost parentheses first and works its way outward from there. In this example, Rexx first performs the `xrange` function; then it executes `c2x`, and finally it runs `x2b`, giving us the binary map in the end:

```
say  x2b(c2x(xrange()))           /* displays the character map in binary */
```

Bit strings have many applications. For example, database management systems manipulate *bit map indexes* to provide quick access to data having a low variety of possible values (*low cardinality*) by ANDing bit strings representing the data values. Another use for bit strings is in the technique called *key folding*. This develops a key for direct (random) data access based on character string key fields. A logical or bit operation is applied to the character field(s) to develop a key that is evenly distributed across direct access slots or positions in the database or on disk. A similar technique called *character folding* is used to map similar characters to a common target, for example, to eliminate certain distinctions between strings. This would be useful when you want similar strings to be compared as equal.

Rexx provides three *binary string functions* that perform logical operations on binary strings:

- `bitand` — Returns the string result of two strings logically AND'd together, bit by bit
- `bitor` — Returns the string result of two strings logically OR'd together, bit by bit
- `bitxor` — Returns the string result of two strings logically EXCLUSIVE OR'd, bit by bit

Here are examples that apply these binary operations on bit strings. The binary string functions return their results in the form of a character string (comprising one character, since 8 bits make a character and the input strings we supply are one character long). Therefore, we use the character-to-hex (`c2x`) and hex-to-binary (`x2b`) functions to interpret the result back to a displayable bit string:

```
say  x2b(c2x(bitand('11110000'b,'11001100'b)))      /* displays: 11000000 */
say  x2b(c2x(bitor('11110000'b,'11001100'b)))       /* displays: 11111100 */
say  x2b(c2x(bitxor('11110000'b,'11001100'b)))      /* displays: 00111100 */
```

The `bitand` operation sets bits to TRUE (1) in the result, only if they are TRUE in *both* strings. `bitor` sets bits to TRUE (1) if they are TRUE in *either* string. The `bitxor` function sets bits to TRUE only if they are TRUE *in exactly one* input string or the other.

The next chapter covers data conversions in further detail and includes an sample program that demonstrates folding a two-part character key. It illustrates the `bitand` function and the `c2x` (character-to-hexadecimal) and `x2b` (hexadecimal-to-binary) conversion functions.

Summary

This chapter introduces string processing. It describes the basic techniques for concatenation and parsing in Rexx and lists the many built-in functions for string and word processing. The sample programs demonstrate some of these techniques and functions.

The techniques we explored included concatenation, or the joining together of strings, and parsing, the analysis and splitting of strings into their constituent substrings. We looked at a sample script that performed input data validation and saw how string analysis and parsing applied to this problem. Then we looked at string functions, including those that analyze *words*, or discrete groups of letters surrounded by spaces or blanks. Finally, we discussed bit strings. These can be used in a wide variety of applications, such as database bit indexes and key folding. We discussed the major bit manipulation functions and how bit strings are converted to and from other forms by using conversion functions.

Chapter 8 illustrates more string manipulation. It includes a script that can tell whether parentheses are balanced (for example, as they might be coded within a Rexx statement). There is also a function called `Reverse`, which reverses the characters in an input string, just like the Rexx built-in `reverse` function. This new `Reverse` script does its work in an interesting way — it calls itself as its own subroutine. Stay tuned!

Test Your Understanding

1. What is *string processing*, and why are outermost features important in a scripting language?
2. What are the three methods of string concatenation? How is each different?
3. What are the three methods of parsing with the `parse` instruction, and how does each operate?
4. Which built-in function would you use for each of the following tasks:
   - ❑ Checking that all characters in one string occur as members in another
   - ❑ Verifying the data type of a user-input data item
   - ❑ Finding the position of a substring with a string
   - ❑ Removing all occurrences of a specified character from a string
   - ❑ Right- and left- justifying a string for printing in a report
   - ❑ Removing leading and/or trailing pad characters from a string
5. What is the difference between the `wordindex` and `wordpos` functions?
6. How are printable characters, hex characters, and bit strings related? What are some of the conversion functions used to convert values between them?
7. What are some of the uses of bit strings in applications?