Chapter 2

REXX PITFALLS

SYS-ED/
Computer Education Techniques, Inc.
Objectives

You will learn:

- Commands and functions: distinction.
- NOP instruction.
- Nested comments.
- Uninitialized variables.
- Variable scoping.
- PARSE instruction.
- Command-line arguments – problems assessing.
- Compound variables – common problems.
- Strict versus non-strict comparison.
- Line continuation – common problems.
- Condition handling – common problems.
- Explicit concatenation.
- Uppercasing by ARG and PULL.
- Case sensitivity of labels.
- Null strings versus omitted strings.
1 Design Philosophy
versus Use in Practice

REXX was designed to be a user-friendly language. However, there are quirks and unexpected results which occur when coding in the REXX language.

The following issues will be most noticeable to IT personnel who have coded with other programming languages.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Client Application</th>
<th>Prioritization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized variables, quoting of literals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable scoping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARSE instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALL statement syntax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compound variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strict versus non-strict comparison</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Line continuation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When to use explicit concatenation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uppercasing by ARG and PULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null strings versus omitted strings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distinction of commands and functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case sensitivity of labels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition handling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested comments</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2 Uninitialized Variables

A common mistake is not to quote all literal strings. Omitting quotes around system commands, will typically result in syntax errors.

Examples:

/* the following actually causes division */
ipfc /inf foo.ipc
/* and this implies multiplication */
erase *.bak

/* the following are correct */
'ipfc /inf foo.ipc'
'erase *.bak'

REXX will perform substitution on unquoted strings that are valid symbols. Depending on the names that have been used for variables there can be a variety of unintended effects.

Example:

copy = 'erase'
...
copy "*..*"

A more general problem with not quoting literal strings is the use of uninitialized variables. These errors can go undetected except for the fact that the program does not behave as expected.
Example:

```
w = 42
call subroutine
...
subroutine: procedure expose x y z
    /* we forgot to expose w */
if w = 42 then
    say "W is the answer."
...
```

REXX provides a "default" value of uninitialized variables which is the (uppercase) name of the variable, the comparison in the above is false ("W" does not equal 42). These errors can be prevented by using this statement at the start of a REXXX program:

```
signal on novalue
```

Every attempted usage of an uninitialized variable will cause an error (which will be "label not found" unless a handler has been included for the NOVALUE condition. However, this will not work if there is a single use of an unquoted literal.

The recommended practice is not to use unquoted literals or uninitialized variables, and always start a program with SIGNAL ON NOVALUE in order to catch uninitialized variable errors.
3 Variable Scoping

Scoping problems with variables will typically result in subtle errors in REXX programs. A variable is visible from the time it is created until entering a subroutine that begins with PROCEDURE. Even then the variable can be added to scope of the subroutine by naming it after EXPOSE.

If there is a program that has many nested subroutines, the typical problem will be that a variable used in one of the higher routines must be exposed in all intermediate routines before it can be used in low-level routines.

Example:

```rexx
x = 50
call first
...
first: procedure
call second
...
second: procedure expose x
say "X-squared is" x**2
```

The variable X is not available, initialized, in the subroutine called SECOND because it was not exposed in FIRST. This problem will occur when certain data variables need to be available globally throughout the entire program. The use of PROCEDURE statements will promote “encapsulation” and prevent subroutines from having unintended side effects.
There are several techniques that can be used to provide global data easily.

**Example 1:**

```rexx
.glbl.screen_height = 25
.glbl.screen_width = 80
.glbl.attributes = 31
```

By placing all data into a single compound variable, it will only be necessary to ensure that the stem `GLBL.` is exposed everywhere.

**Example 2:**

```rexx
globals = 'height width attr'
height = 25
width = 80
attr = 31
...
subroutine: procedure expose (globals)
```

An alternative is to list the names of all required variables in a string and then do a special kind of expose. By placing the variable name in parentheses after EXPOSE, this informs REXX that the variable and all variable names contained in the value should be exposed.
4 PARSE instruction

The PARSE instruction needs to be purposefully applied in order to be effectively used.

4.1 Incorrect use of WITH

The keyword WITH is used only with PARSE VALUE. In all other forms of PARSE, WITH is not a keyword and will not raise any error condition; it will be interpreted as a variable name.

Example:

/* the following is correct */
parse value date('u') with month '/' day '/' year

/* the following is a very hard to find error */
x = "When in the course of human events"
parsed var x with a b c
say a b c       /* says "in the course" */
4.2 Unexpected Blanks in Parsed Results

The rules of PARSE provide that the last variable to be assigned immediately before a literal subpattern or before the end of the whole pattern will contain all remaining characters.

Commonly this will include leading or trailing blanks. However, REXX ignores blanks in comparisons and numbers, but not in other contexts. This will result in problems.

Example:

```rexx
call subroutine "one ( two 40 ) three"
...
subroutine:
parse arg a '(' b c ')' d /* a = "one"
   b = "two"
   c = "40"
   d = " three" */
if a == "one" then /* strict comparison fails */
   say "Not this"
if datatype(c, 'x') = 1 then /* not valid hex because */
   say "Nor this" /* of trailing blank */
if abbrev("three", d) then /* not abbreviation because */
   say "Nor this" /* of leading blank */
```

The STRIP() function can be used to address this problem. It will remove the leading and trailing blanks.

The special "." notation in a PARSE pattern can also be used for sending blanks to the bit-bucket.

Example:

```rexx
call subroutine "one ( two 40 ) three"
...
subroutine:
parse arg a .'( b c .' )' d /* a = "one"
   b = "two"
   c = "40"
   d = "three" */
```
4.3 Mismatched Arguments and PARSE Pattern

PARSE in a PARSE ARGS statement can be used to access the arguments of a subroutine.

Normally, arguments are passed to a subroutine as a list of values separated by commas. However, it is important to remember that the commas must also be used in the PARSE ARGS statement.

Example:

```
call subroutine 'carl', 'friedrich', 'gauss'
...
subroutine:
  /* this is probably wrong: */
  parse args first second third   /* first = 'carl'
     second = ''
     third = '' */
  /* this is probably what is intended: */
  parse args first, second, third /* first = 'carl'
     second = 'friedrich'
     third = 'gauss' */
```

The first PARSE ARGS statement doesn't work because it is only the first argument has been accessed and it consists of only a single word, since there are no commas separating subpatterns of the template.

The recommendation is to always use the same number of commas in the PARSE ARGS statement as are used in the corresponding procedure invocation.
5 Accessing Command-line Arguments

A common problem is in receiving command line arguments. When CMD.EXE invokes a REXX program, it places the entire string following the program name into a single argument. This is true even when the string contains embedded commas.

Example:

    parrot hello, world

    /* a general parrot program */
    parse args my_args
    say 'Polly says "'my_args',"'
    return

The output will be:

    Polly says "hello, world."

When receiving command line arguments, it is likely that a user will include excess blanks before or after the argument string - intentionally or otherwise. This can cause excess blanks to be included in the parsed variables, which may create program errors.
6 CALL Statement Syntax

REXX programmers frequently have trouble with the CALL instruction. This is because parentheses should not be used around the list of arguments.

However, parentheses are required when the procedure is invoked as a function.

The problem is further compounded by the fact that the following code snippet will work correctly:

```
call subroutine ('only one argument')
```

The reason that this works is that 'only one argument' is an expression consisting of a literal string, and parentheses can be placed around a REXX expression. This will work even if there were no space between 'subroutine' and '('.

This code snippet does not work:

```
call subroutine ('first argument', 'second argument')
```

The reason it doesn't work is that two literals separated by a comma do not constitute a valid expression, and an error message will be issued to this effect.
7 Compound Variables

REXX compound variables can lead to a variety of common problems.

7.1 Case Sensitivity in Compound Variable Tails

Whenever a reference is made to a compound variable in a program, REXX automatically interprets the symbol as if it were written in upper case.

Example:

Country.Tuesday = 'Belgium'

This code snippet assigns a variable whose name is COUNTRY.TUESDAY, provided TUESDAY is not itself the name of a variable. The stem, COUNTRY., is automatically taken as upper case, and the tail contains just one part. REXX looks for a simple variable called TUESDAY, which is also upper case, and if none has been initialized, the default initial value, which is TUESDAY, is substituted.

There is distinction between the name of a compound variable, and the symbol which is used to refer to it. This distinction often causes problems, particularly related to case.

Example:

day = 'Tuesday'
say "If it's" day", this must be" Country.day"."

Assuming that the preceding assignment is made, then what would be displayed is:

If it's Tuesday, this must be COUNTRY.Tuesday.

This occurs because the variables COUNTRY.TUESDAY and COUNTRY.Tuesday are distinct.

However, as far as REXX is concerned the symbols are not distinct.
7.2 Inability to Have 'Constant' Values in Tails

In order to obtain the same effect as the structures in C, PL/I, and other languages, it would be useful to have REXX create record-like data structures using compound variables.

Example:

```
person.age.name
person.ssn.name
person.salary.name
```

However, all parts of the tail of the compound variable are subject to substitution.

Example:

```
name = 'Kilgore Trout'
say "SSN of" name "is" person.ssn.name
```

This will provide a satisfactory result as long as SSN is not used as a REXX variable. This is because REXX will use the uninitialized value, which is "SSN". However, if SSN is ever used as a variable, intentionally or otherwise, this will not provide the required processing. Even if this doesn't happen, there will be an adverse impact on performance. This is because REXX has to do a full variable look-up in order to discover that SSN is uninitialized.

There are is no single satisfactory solution to this problem.

Example:

```
person.'ssn'.name
```

This will evaluate to the concatenation of the stem value `person`, the literal `ssn`, and the symbol `.name`. 
In general, do not use SSN as a variable in a program.

Or a scratch variable can be used for containing the exact value that is required.

Example:

```
x = 'SSN'
say "SSN of" name "is" person.x.name
```

It will also be necessary to use the right case in the literal. This is because case is significant in the evaluated form of a stem.

Example:

```
x1 = 'ssn'
x2 = 'SSN'
say person.x1.name "is not necessarily the same as",
     person.x2.name
```

A symbol can be used which is not evaluated as a variable. This works because REXX will always take 0ssn as a literal and not try to evaluate it, since variable names can't start with numbers. But it will not be aesthetically pleasing.
Example:

```rexx
name = 'Kilgore Trout'
say "SSN of" name "is" person.0ssn.name
```

Another approach is not to use "multidimensional" compound variables, and instead adopt a naming convention.

Example:

```rexx
person_age.name
person_ssn.name
person_salary.name
```

In terms of readability, performance, and relative immunity to the aforementioned issues. However it does make it more difficult to deal with the compound variable as a whole.
7.3 Inability to Have Expressions in Tails

REXX compound variables should not be considered equivalent to arrays in other languages.

REXX does not allow arbitrary expressions in "array" subscripts.

Example:

```
i = 10
j = 20
say "Value =" array.(i+j)
```

This code snippet does not display the value of array.30. It attempts to call a function called array, which will fail because the function does not exist. In REXX, a symbol, `array`, is immediately followed by a left parenthesis is considered to be a function reference.

The only way to use a "computed subscript" is to assign the value to a temporary variable:

Example:

```
x = i + j
say "Value =" array.x
```

A similar problem arises when one compound variable is used as a subscript in another. Assume that the stem `book` is used to contain the index (subscript) of a data item related to books. `Stem` itself will be indexed by the name of a book. This is known as an associative array, since data can be retrieved by "associations". It is commonly used in advanced REXX programming, and one of the most powerful features of the language.

For performance reasons, it is desirable to store a multi-column table of book-related information in a number of compound variables that have a numeric index instead of being indexed by a string. This will be faster and requires less storage space, since the index string doesn't need to be stored multiple times internally.
Example:

i = 1000
name = "Memoirs of a Lady of Pleasure"
book.name = i
author.i = "John Cleland"
date.i = "1790"
bkname.i = name

The reason that the actual book title has been used to index the book array is that the assumption is that there is some need to retrieve book information by the exact title. However, the overhead associated with using long strings as indices in every column of the table can be minimized by keeping a row number and using that as the index.

When it is necessary to retrieve information, such as the author of a given book, it can be done by coding the:

title = "Lolita"
index = book.title
say "The author of" title "is" author.index".

It would not have worked to do this:

title = "Lolita"
say "The author of" title "is" author.book.title".

This is because REXX attempts to substitute values for book and title separately and independently. Unless book has been assigned some value, it will be evaluated as BOOK and the resulting tail will be BOOK.Lolita
7.4 Inability to Deal with Cross Sections of a Compound Variable

Example:

This is a database for a garden club:

```plaintext
name = "Susy Flor"
address.name.0 = 2
address.name.1 = "1234 Asphodel Way"
address.name.2 = "Fleur, XX 99999"
specialty.name.0 = 3
specialty.name.1 = "Amaryllis"
specialty.name.2 = "Hyacinth"
specialty.name.3 = "Wisteria"
```

If an individual leaves the club, there will need to be a way to remove all related information.

Approach:

```plaintext
name = "Susy Flor"
drop address.name. specialty.name.
```

This Approach snippet will not work as intended. While it will not cause a REXX error, it will try to drop REXX variables called `ADDRESS.Susy Flor` and `SPECIALTY.Susy Flor`. This is because `address.name` and `specialty.name` are not valid REXX stems.

There is no simple way to do achieve the intended result in REXX. All that can be done is to write loops for dropping each variable individually.
8 Strict versus Non-strict Comparison

Comparison in REXX with the = operator is "non-strict". This means that REXX will attempt to determine whether both operands are numeric values, in which case a numeric comparison will be performed.

Even when a character-string comparison is done, leading or trailing blanks on both operands are ignored. This will be frequently helpful when dealing with user input, since extra blanks may well be present. Non-strict comparison rules may apply with any comparison operator, such as <, >, <=, \=, etc.

There is another type of comparison operation which is "strict". That is, the operation treats the data only as character strings rather than as numbers. As a result, strict comparisons will be faster. Furthermore, leading and trailing blanks are not ignored in a strict comparison. Strict comparison operators are written as <<, >>, <<=, \==, and so forth.

The choice as to the proper types of comparison to use in any given case can be problematic:

The non-strict comparison operations are used most commonly.

Example:

```rexx
a = '3'
b = '3.0'
c = '3e0'
d = ' 3 '
say (a = b),' ' (a = c),' ' (a = d) /* displays "1, 1, 1" */
```

All comparisons will yield a value of "1" (true), because all of the strings are equivalent forms of the number 3. This is probably the intended result.

If instead strict comparison (==) were used in the SAY statement, then the result would be "0, 0, 0", because all the strings would be distinct as character strings.

There are issues hidden in the convenience of the non-strict comparison operators. One serious issue would result from the fact that numbers can be represented in exponential notation with an embedded "e", such as 3e0.
However, non-numeric data may contain such strings naturally, such as with hexadecimal values.

Example:

```
say '3e0' < '300' /* gives "1", since 3 < 300 */
```

If the data were intended to represent either character strings or hex numbers, then the program would fail, because the correct result should be "0" ('3e0' is after '300' in the ASCII collating sequence).

Strict comparison (<<) should have been used.

Another problem when working with character strings is treating strings which may be interpreted as numbers. If these strings are longer than the current NUMERIC DIGITS setting (normally 9 digits), then the results will be unpredictable.

Example:

```
say '1234567890' = '1234567891' /* gives "1" */
```

This will provide the wrong answer for most purposes. This is because the strings are interpreted as numbers, and by the definition of REXX arithmetic comparison, they are equal. Only 9 significant digits are considered in the comparison.
9 Line Continuation

When a REXX clause is not complete on one line, it will be necessary to use a continuation character. A comma is the continuation character.

However, commas are also used to separate arguments in a procedure call and sub-templates in a parse pattern. It is easy to forget to add an extra comma when one of these statements is continued.

Example:

```
say max(3, 4, 5,
       6, 7, 8, 9)
```

This statement will produce an error 40; which is an incorrect call to a routine. Since the ending comma on the first line is taken to be a continuation character, the result will be the same as the code had been.

Example:

```
say max(3, 4, 5 6, 7, 8, 9)
```

Since the comma is replaced with one blank when the second line is concatenated to the first, and "5 6" isn't a valid number because of the embedded blank, the example should have been written:

Example:

```
say max(3, 4, 5,,
       6, 7, 8, 9)
```
10 Condition Handling

A REXX program can trap certain exceptional conditions by using a SIGNAL ON or CALL ON statement.

Example:

call on halt
...
/* control comes here when ctrl-break is hit */
halt:
say "Program interrupted. Do you want to continue?"
pull ans
if abbrev("YES", ans) then
    return
/* terminate the program */
exit

This code will trap pressing of the Ctrl-Break (or Ctrl-C) key by the user; this will interrupts the program.

In a more complex program, there will be problems. A choice can then be made to the user by offering the user the option of leaving the current operation or computation and returning to an initial prompt in the program.

Example:

signal on halt
/* this is the main prompt of the program */
restart:
say "Enter a command..."
pull command
...
halt:
say "Program interrupted. Do you want to restart the program?"
pull ans
if abbrev("YES", ans) then
    signal restart
/* terminate the program */
exit
It will not work right if the HALT condition occurs while a subprocedure is being executed. This is because as far as REXX is concerned, the program is still executing the subprocedure. The only way to get out of a subprocedure is a RETURN or EXIT statement; SIGNAL will not do it.

The program might appear to work correctly, but it will be subject to various kinds of errors. If the subprocedure started with a PROCEDURE statement, then important variables might not be exposed. Given the way that REXX is currently defined, there is no way to solve this kind of problem.

There is another sort of problem which can occur.

**Example:**

```REXX
signal on halt
```

As soon as the HALT condition is raised by the user pressing Ctrl-Break, further handling of this condition will be disabled. It will be necessary to explicitly execute SIGNAL ON again in order to re-enable the handler. This can be done by putting another immediately after the `halt: label`. 
11 Explicit Concatenation

The concatenation operation in REXX is normally implicit, but it can be requested explicitly with the || operator. There are times when || needs to be used explicitly. REXX code typically is fairly free-format; blanks or their absence is not important.

However, there are exceptions:

- If one or more blanks occur between symbols, literals, or parenthesized expressions, then "blank concatenation" is implied, rather than "abuttal concatenation".
- If a string or a literal is followed immediately (without intervening blanks) by a left parenthesis, then REXX treats it as a function reference.
- If a literal is followed by the letter 'X' (upper or lower case), then REXX treats it as a hex string (even if the string is not a valid hex number).
- If a literal is followed by the letter 'B' (upper or lower case), then REXX treats it as a bit string (even if the string is not a valid bit string).

When a concatenation operator is not included in certain expressions, the end result will be unintended function call references.

Example:

```rexx
say "Number of observations--"(alpha + beta)
```

In REXX any quoted string is a "token", which can refer to a function to be called if it is followed immediately by a left parenthesis. This is what happened with the code snippet.

Example:

```rexx
say "Number of observations--"||(alpha + beta)
```

The code snippet has been changed to reflect the proper/intended result.
When variables called X or B are used and the letters X or B occur by themselves immediately following a quoted string, they cause the string to be interpreted as a hex or binary literal - even if its syntax is not proper for such a literal. The result will be an error 15; which is an invalid hexadecimal or binary string.

Example:

```rexx
x = 100
y = 50
say "Sum="x + y
```

This should have been written as:

```rexx
say "Sum="||x + y
say "Sum=" x + y
```

Or avoid the use of variables named X or B altogether.
The ARG and PULL instructions are frequently required in REXX programs.

- ARG instruction is provided as an abbreviation of PARSE UPPER ARG.
- PULL is an abbreviation for PARSE UPPER PULL.

The specification of UPPER, will causes automatic uppercase conversion of strings and can be problematic. By making strings case insensitive, all strings can be treated as upper case and it will not be necessary to deal separately with equivalent lower or mixed case strings.

Automatic upper casing by ARG and PULL can be a source of unanticipated bugs.

ARG is typically used heavily and there may result in an inadvertent comparison of using lower or mixed case strings against variables set by ARG.

Passing binary data to a subroutine will also result in a problem. Binary data may be read from a file or produced by the D2C or X2C functions or be encoded using hex string literals. Any characters in that data that happen accidentally to be lower case alphabetic letters will be converted unexpectedly by ARG!

Example:

```
data = charin(file, 1, 50) /* read 50 bytes */
call subroutine data
...
subroutine: procedure
arg record /* whoops, random data mangling! */
```
13 Case Sensitivity of Labels

A typical REXX program will be mostly case insensitive. Keywords, variable names, and labels can be in upper, lower, or mixed case. REXX will treat these items as if they were always upper case.

The SIGNAL instruction can be used to implement a computed GOTO. Although SIGNAL also is defined to terminate any open DO groups, there will be problems with this usage. Nevertheless, it can be a useful technique which used properly can be more efficient than a SELECT statement in containing a large number of WHEN clauses.

Example:

```
signal value 'CASE'x
  case1: /* handler for the case x = 1 */
  ...
  case2: /* handler for the case x = 2 */
  ...
  case3: /* handler for the case x = 3 */
  ...
```

It will be important to watch for alphabetic case sensitivity. The value of the expression in the signal statement must match the alphabetic case of labels in the program exactly, even though REXX will always take the labels themselves to be upper case.

If signal value 'case'x had been used, an error 16 (label not found) would occur. This is because REXX would look for labels like case1, case2, etc., even though the actual labels in the program were CASE1, CASE2, etc. This is in spite of how they were written,
14 Null Strings versus Omitted Strings

In calls to REXX routines, including the main program, arguments can be omitted. The omission of an argument will not cause a REXX error; except for built-in functions, which often have required arguments.

There is a technique for using the ARG built-in function which will determine in a subroutine whether an argument has been omitted.

Example:

```rexx
make_example: procedure
  if arg(1, 'o') then do
    say "Required first argument of 'make_example' omitted."
    exit
  end
```

Unlike with built-in functions, there is not any way to raise error 40, which is an incorrect call to routine, automatically. Most REXX programmers, will use the short-cut of testing an argument for a null string. This is because REXX supplies a null string whenever an attempt is made to refer to a missing argument.

Example:

```rexx
make_example: procedure
  if arg(1) = '' then do
    say "Required first argument of 'make_example' omitted."
    exit
  end
```

This is not a fundamentally sound approach for the following reasons:

1. Ordinary comparison was used therefore if a string consisting of multiple blanks had been passed, it would still be treated as if it had been omitted.

2. Even when a strict comparison to the null string had been made, it would still complain when an explicit null string had been passed.

3. There may not be valid reasons to accept a null string as an argument, however there should not be a complete omission of the argument.
A different form of this problem can occur when an argument is blindly passed to a built-in function.

Example:

```rexx
subroutine: procedure
parse arg first, second
if max(first, second) > 100 then do
    say "Argument too large"
    exit
end
```

If either argument has been omitted, this program will fail with error 40 in the call to MAX. This is because PARSE ARG will set first or second to a null string if the corresponding argument is omitted.

A null string is not identical to an omitted string, and MAX will fail when it is passed a null string. However, it will not be fail by omitting the second argument.
15 Commands and Functions: Distinction

The term command is ambiguous in the REXX programming language.

There are approximately 25 native commands in REXX. However, the preferred term is instruction or keyword instruction.

A partial list of native commands in REXX include:

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>CALL</th>
<th>DO</th>
<th>EXIT</th>
<th>IF</th>
<th>PARSE,</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULL</td>
<td>PUSH</td>
<td>QUEUE</td>
<td>RETURN</td>
<td>SAY,</td>
<td>SELECT</td>
</tr>
</tbody>
</table>

When these keywords are used in a REXX program, they are never quoted.

There is another class of REXX statements that consist of commands to an external command processor. These commands start with a keyword and may be followed by one or more parameters.

In order to avoid common problems it is advisable to always enclose command keywords in single or double quotes.

Example:

'copy config.sys config.bak' /* entire command quoted */
parse arg from to
'copy' from to /* only command name quoted */

Certain system commands handled by CMD.EXE, such as CALL, IF, and EXIT, have the same names as REXX keyword instructions. These must appear inside quotes if they are not to be treated as REXX instructions.

Built-in functions provide for another aspect of REXX service. There are approximately 65 functions, including SUBSTR, POS, LINEIN, LINEOUT, MAX, and VALUE, etc. From the perspective of their utilization in REXX, they are functions. This means that they return a value.

Some of the functions are used primarily for the "side effects" they produce rather than for their value: LINEOUT, CHAROUT, STREAM, and VALUE. Other useful services are available through functions in external function packages such as REXXUTIL.
A common error is to invoke a function by writing it as a function call on a line by itself; this is what is done in the C language. What occurs is that the value returned by the function (often "0" or "1") is passed as a command to the default command environment, which will try to execute it. The result will be a SYS1041 error message stating that the command is not a recognized internal or external command, operable program, or batch file.

The proper way to invoke such a function is either by using it on the right hand side of an assignment statement or by invoking it with a REXX CALL instruction. The use of a CALL means parentheses around the argument list should be omitted.

**Example:**

```rexx
define value 'path', newpath, 'os2environment' /* wrong */
define x = value('path', newpath, 'os2environment') /* right */
define call value 'path', newpath, 'os2environment' /* right */
```
16 NOP instruction

The NOP instruction has several uses. It is required in the syntax of IF and SELECT statements.

Example:

```rexx
if some_condition() then nop; else say /* this is correct */
  "Condition was false."
if some_condition() then; else say /* this is NOT correct */
  "Condition was false."
```

Example:

```rexx
if x >= 100 then
  if y >= 200 then
    say "X is >= 100 and Y is >= 200."
  else
    nop
else
  say "X is < 100."
```

When there are nested IF statements, it will sometimes be necessary to have an ELSE clause which does nothing, in order that the conditions match:

When the first branch of an IF statement is null, the NOP must be used.

Example:

```rexx
nop /* about to invoke erase command */
'erase' name_of_something
```

NOP is frequently used in tracing. The REXX specification states that during interactive tracing REXX will stop only after a statement is executed. NOP is used to pause immediately before a statement.
Example:

do forever
   if something then do
     nop /* about to signal out of loop */
     signal something_handler
   end
...
end

REXX will not pause during interactive tracing for certain types of statements, such as CALL and SIGNAL. NOP can be used to force a pause.
REXX allows nesting of comments. It is useful because it allows code to be temporarily removed temporarily from a program by enclosing it in comment delimiters.

**Example:**

```rexx
say something
/* the following code is has been removed for debugging
/* this should never be 0 if x, y, z, n are integers >= 3 */
a = x**n + y**n - z**n
if a = 0 then
   'erase' myfile
*/
call subroutine
```

Comments can be nested to any arbitrary depth. There is a subtle programmatic issues which need to be recognized and addressed.

While REXX is scanning for comments within comments it looks only for "*/" and "/*". In particular, it will ignore the possibility that these character sequences may occur in quoted strings.

When a sequence of code is enclosed containing either "*/" or "/*" in a literal string, then a mismatched comment error will probably result.

**Example:**

```rexx
/* This isn't going to work...
say 'Watch out for the use of "/*/'.
*/
```

With this code snippet, REXX will assume that a second level of comment nesting has occurred, and that the entire remainder of the program will be treated as a comment.