Daney, Charles - Common REXX Pitfalls / How do I... ? (Fragments)

REXX is billed as a user-friendly language that is as easy as possible to read and write. Part of the foundation for this claim is that REXX syntax was carefully designed to "do the right" thing in the most common usage situations - the "principle of least surprise". Unfortunately, this objective isn't achieved 100% - sometimes surprising things happen in REXX, particularly from the point of view of people who have used other programming languages. Sometimes these surprises are actually the by-product of decisions that were intended to make REXX easier in some way or other.

Whatever the reason, we list here some of the language pitfalls that have been revealed by the experience of thousands of users over the years - things that show up again and again in customer support calls and messages in forums where REXX is discussed.

Many questions about OS/2 REXX that occur in customer support calls and online REXX forum discussions actually relate to features and capabilities of the operating system rather than of REXX itself. The following represents some of the questions that arise most often. See: How do I...?

Common REXX Pitfalls

- The NOP instruction
- Uninitialized variables, quoting of literals
- Variable scoping
- The PARSE instruction
- CALL statement syntax
- Nested comments
- Compound variables
- Strict vs. non-strict comparison
- Line continuation
- Condition handling
- When to use explicit concatenation
- Uppercasing by ARG and PULL
- Case sensitivity of labels
- Null strings vs. omitted strings
- Distinction of commands and functions

How do I...?

- Make a string all upper case or lower case
- Trap ctrl-break and ctrl-c
- Pass an array to a subroutine
- Return an array from a subroutine
- Format numbers neatly for output
- Write a line without a carriage return

- Testing for keyboard input
- Store data in a program

The NOP instruction

Although the NOP instruction does nothing, it has several uses.

First, it is required in the syntax of IF and SELECT statements. For instance, when the first branch of an IF statement is null, you must use NOP:

```
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."
```

Of course, this is not a natural example, since you would ordinarily make the only branch of an IF immediately follow THEN. However, when you have nested IF statements, you sometimes need to have an ELSE clause which does nothing, just so that the conditions match:

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

Another case in which NOP is useful is in tracing. The specification of REXX says that during interactive tracing REXX will stop only after a statement is executed. You can use NOP to pause just before a statement:

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

Also, REXX will not pause at all during interactive tracing for certain types of statements, such as CALL and SIGNAL. You can use NOP to force a pause:

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

(Back to title page)

Uninitialized variables, quoting of literals

By far the most frequent mistake that both beginning and experienced REXX users make is neglecting to quote *all* literal strings. It is tempting to leave off quotes around system commands, but this easily leads to mysterious syntax errors, e. g.:

```
/* 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'
```

Even worse, REXX will perform substitution on unquoted strings that are valid symbols. Depending on the names you use for variables you can produce some very unintended effects:

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

There is a more general problem with not quoting literal strings even when the direct results are harmless. One of the most frequent types of programming mistakes that both beginners and experience REXX users make is the use of uninitialized variables. Such errors can go completely undetected except for the fact that the program does not behave as expected:

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

Since 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). There is a very easy way to prevent errors of this kind: *always* put the statement

```
signal on novalue
```

at the start of any REXX program. Then every attempted usage of an uninitialzed variable will cause an error (which will be "label not found" unless you actually include a handler for the NOVALUE condition). Of course, this won't work if you have even one "innocuous" use of an unquoted literal.

So the best recommendation is: *Never* use unquoted literals or uninitialized variables, and always start a program with SIGNAL ON NOVALUE in order to catch uninitialized variable errors.

(Back to title page)

Variable scoping

Scoping problems with variables tend to cause many subtle errors in REXX programs. "Scope" refers to the part of a program in which a given variable is "visible". Normally, a variable is visible from the point 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 you have a program that has many nested subroutines, the typical problem is 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:

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

In the above example, the variable X is not available (or rather, is not initialized) in the subroutine called SECOND because it was not exposed in FIRST.

This problem most often arises when certain data variables need to be available globally throughout the whole program. However, the use of PROCEDURE statements is also a good thing, since it promotes "encapsulation" and prevents subroutines from having unintended side effects.

There are several techniques that can be used to provide "global" data easily. One is to place all such data into a single compound variable:

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

Then you need only be sure the stem GLBL. is exposed everywhere. An alternative is to list the names of all required variables in a string and then do a special kind of expose:

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

Placing the variable name in parentheses after EXPOSE tells REXX that the variable itself, and all variable names contained in the value should be exposed.

(Back to title page)

The PARSE instruction

The PARSE instruction is a very powerful REXX feature, but it can take quite a bit of experience to use it effectively.

It's also very easy to create subtle errors which can be very hard to find if you don't know some of the details of how PARSE operates.

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, since it will be interpreted as a variable name:

```
/* 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"
parse var x with a b c
say a b c       /* says "in the course" */
```

2. Unexpected blanks in parsed results

The rules of PARSE provide that the last variable to be assigned just before a literal subpattern or before the end of the whole pattern will contain all remaining characters. Frequently this includes some leading or trailing blanks. This can present subtle problems, since REXX ignores such blanks in ordinary comparisons and in numbers, but not in other contexts:

```
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 most general way to deal with this is to use the STRIP() function where extraneous leading or trailing blanks could be a problem, since STRIP() removes the leading and trailing blanks.

In some cases you can use the special "." notation in a PARSE pattern to send blanks to the bit-bucket:

```
call subroutine "one ( two 40 ) three"
...
subroutine:
parse arg a . '(' b c . ')' d . /* a = "one"
                                   b = "two"
                                   c = "40"
```

```
                                   d = "three" */
```

3. Mismatched arguments and PARSE pattern

One of the most common uses of PARSE is in a PARSE ARGS statement that is used to access the arguments of a subroutine. Normally, arguments are passed to a subroutine as a list of values separated by commas. It is easy to forget that the commas must *also* be used in the PARSE ARGS statement:

```
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 reason that the first PARSE ARGS statement doesn't work is that only the first argument (which consists of only a single word) has been accessed, since there are no commas separating subpatterns of the template.

In general, it would be a good rule of thumb to always use the same number of commas in the PARSE ARGS statement as are used in the corresponding procedure invocation.

4. Problems accessing command-line arguments

A very common problem that is the inverse of the one just discussed occurs 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 if the string contains embedded commas.

For instance, if you enter

```
parrot hello, world
```

on the command line, then the parrot.cmd program should be something like this:

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

The output of this will be

```
Polly says "hello, world."
```

There's one other thing to be careful of when receiving command line arguments. That is, it's quite possible that a user will include excess blanks before or after the argument string (intentionally or otherwise). As discussed above, this can cause excess blanks to be included in the parsed variables, which may create some very subtle program errors.

(Back to title page)

CALL statement syntax

Beginning REXX users frequently have trouble with the CALL instruction. The problem is that parentheses should *not* be used around the list of arguments. It is easy to forget this, because parentheses *are* required when the procedure is invoked as a function. The problem is further compounded by the fact that the following will actually 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 one may always place parentheses around a REXX expression. (This would also work even if there were no space between 'subroutine' and '('.)

However, this 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.

(Back to title page)

Nested comments

REXX allows nesting of comments. This is a very convenient feature, which is not present in similar languages, like C. The main reason it is useful is that it allows you to easily "remove" code temporarily from your program by enclosing it in comment delimiters:

```
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 danger in this, however. 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. If you enclose a sequence of code containing either /* or */ in a literal string, then a mismatched comment error will probably result:

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

What will happen here is that REXX will assume that a second level of comment nesting has occurred, and probably the entire remainder of the program will be treated as a comment. Surprise!

(Back to title page)

Compound variables

There are certain subtleties in the use of the REXX compound variables. These lead to a variety of common problems.

1. Case sensitivity in compound variable tails

Whenever you refer to a compound variable in a program, REXX automatically interprets the symbol as if it were written in upper case. Therefore,

```
Country.Tuesday = 'Belgium'
```

actually assigns a variable whose name is COUNTRY.TUESDAY, provided TUESDAY is not itself the name of a variable. What's actually happpening is that the stem, COUNTRY., is automatically taken as upper case, and the tail contains just one part. REXX looks for a simple variable called TUESDAY (also upper case), and if none has been initialized, the default initial value, which is TUESDAY, is substituted.

There is, however, an important 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. For instance, if you had the following:

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

Then assuming the preceding assignment, what would be displayed is

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

That is because the variables COUNTRY.TUESDAY and COUNTRY.Tuesday are distinct (though the symbols are not, as far as REXX is concerned).

2. Inability to have 'constant' values in tails

One would often like to create record-like data structures using compound variables, in order to obtain the same effect as one has with structures in C, PL/I, and other languages. For instance, it would be nice to represent personnel records using variables like

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

Unfortunately, all parts of the tail of the compound variable are subject to substitution. For instance, consider

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

This will work OK as long as SSN is not used as a REXX variable, because REXX will use the uninitialized value, which is "SSN". However, if SSN is ever used as a variable, intentionally or otherwise, this will probably break. Even if it this doesn't happen, this usage has an adverse performance impact, since REXX has to do a full variable look-up in order to discover that SSN is uninitialized.

There aren't any fully satisfactory solutions to this problem, either. You cannot simply write

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

since that evaluates to the concatenation of the stem value **person.**, the literal **'ssn'**, and the symbol **.name**.

One thing you can do is either to be very careful not to use **SSN** as a variable in your program, but the it's obviously very easy to forget about this restriction. Another thing you could do is to use a scratch variable to contain the exact value that you need:

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

Obviously, this is a lot of extra work, and you also have to be careful about being consistent to always use the right case in the literal:

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

because case is significant in the evaluated form of a stem.

Another possibility is to use a symbol which can't possibly be evaluated as a variable:

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

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 is obviously not very aesthetically pleasing.

One remaining possiblity is to not try to use "multidimensional" compound variables, and instead adopt a naming convention like this:

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

There's a lot to be said for this approach in terms of readability, performance, and relative immunity to the problems we've been discussing. But it does make it more difficult to deal with the compound variable as a whole, for instance if you need to DROP the whole data structure. In this case, you would have to drop each distinct stem, instead of just

```
drop person.
```

3. Inability to have expressions in tails

It is very tempting to think of REXX compound variables as if they were just like arrays in other languages. Unfortunately, this is not quite possible. One reason is that REXX does not allow arbitrary expressions in "array" subscripts. For instance,

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

Does not display the value of **array.30**. Instead, it tries to call a function called **array.**, which will probably fail because the function does not exist. The reason is that in REXX, a symbol (which **array.** is) immediately followed by a left parenthesis is considered to be a function reference.

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

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

A similar problem arises when you want to use one compound variable as a "subscript" in another. Suppose, for instance, that you use the stem **book.** 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 it is 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 is faster and requires less storage space, since the index string doesn't need to be stored multiple times internally.) A typical record of book information might be set up like this:

```
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 (we assume) there is some need to retrieve book information by the exact title. However, we avoid the overhead of using long strings as indices in every column of the table by keeping a row number and using that as the index.

When the time comes to retrieve some information, such as the author of a given book, we do it like this:

```
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"."
```

The reason is that REXX tries 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
```

4. Inability to deal with cross sections of a compound variable

Consider a possible database for a garden club:

```
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 should be a way to remove all related information. One is tempted to do this:

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

But this won't work as intended. While it will not cause a REXX error, all it will do is 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 isn't any simple way to do the intended thing in REXX. All you can do is write loops to drop each variable individually.

(Back to title page)

Strict vs. 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 done. In addition, even if a character-string comparison is done, leading or trailing blanks on both operands are ignored. This is frequently helpful when dealing with user input, since extra blanks may well be present. (See The PARSE instruction.) Such "non-strict" comparison rules in fact 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 possibly as numbers. Because of this, strict comparisons can be a little faster. Furthermore, leading and trailing blanks are not ignored in a strict comparison. Strict comparison operators are written as <<, >>, <<=, \==, and so forth.

The choice of the proper types of comparison to use in any given case can be somewhat confusing. The non-strict comparison operations are used most commonly, e. g.:

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

Here, all comparisons 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 are distinct as character strings.

However, there are pitfalls hidden in the convenience of the non-strict comparison operators. One of the most insidious is a result of the fact that numbers can be represented in exponential notation with an embedded "e", for instance **3e0**. But sometimes, non-numeric data may contains such strings naturally, such as with hexadecimal values. Consider:

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

If the data in this example were intended to represent either character strings or hex numbers, then the program would fail, because the correct result in that case should be "0" ('3e0' is after '300' in the ASCII collating sequence). Strict comparison (<<) should probably have been used here.

There is yet another problem when you are working with character strings which you want to treat as strings but which may be interpreted as numbers. Perhaps they are database keys. If these strings are longer than the current NUMERIC DIGITS setting (normally 9 digits), then you can get very surprising results:

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

This gives what is probably the wrong answer for most purposes, because the strings are interpreted as numbers, and by the definition of REXX arithmetic comparison, they are equal, since only 9 significant digits are considered in the comparison.

(Back to title page)

Line continuation

When a REXX clause is not complete on one line, it is necessary to indicate this with a comma, which is the continuation character. Normally this is a convenience, since most statements do not need to be continued, and therefore the end of the line can be taken as the end of the statement, which makes it unnecessary to include a semicolon after each statement.

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

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

This statement will produce an error 40 (incorrect call to routine), since the ending comma on the first line is taken to be a continuation character. The result is the same as if you had coded

```
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. This example should have been written

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

(Back to title page)

Condition handling

A REXX program can trap certain exceptional conditions by using a SIGNAL ON or CALL ON statement. For instance, you can trap pressing of the Ctrl-Break (or Ctrl-C) key by the user, which interrupts the program:

```
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
```

However, there can be problems with this in a more complex program. Perhaps one of the choices you want to offer the user is the option of leaving the current operation or computation and returning to an initial prompt in the program. What you want to do may be something like this:

```
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
```

The problem here is that this most likely will not work right if the HALT condition occurs while a subprocedure is being executed, 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) statment - SIGNAL doesn't do it. The program might appear to work correctly, but it will be subject to various kinds of errors. For instance, important variables might not be exposed if the subprocedure started with a PROCEDURE statement. Unfortunately, there is no way to solve this kind of problem as REXX is currently defined.

There is another sort of problem which can occur, but which is very easy to fix. In the example above, as soon as the HALT condition is raised by the user pressing Ctrl-Break, further handling of this condition will be disabled. You will need to explicitly execute SIGNAL ON again in order to re-enable the handler. In the example above, this can be done by putting another

```
signal on halt
```

immediately after the **halt:** label.

(Back to title page)

When to use 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. Normally REXX code is fairly free-format. That is, blanks or their absence is not important. However, there are a few important 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).

You can wind up with unintended function call references if you forget to write a concatenation operator in certain expressions, e. g.:

```
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 (without intervening blanks) by a left parenthesis. That is what would happen here, even though the string in question doesn't look at all like a function name. (And the corresponding function almost certainly will not be found.) This should have been written

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

An even more insidious problem can occur if you use variables called **X** or **B** When the letters X or B occur by themselves immediately follwing 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. So you will get error 15 (invalid hexadecimal or binary string) instead of what you wanted from

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

This should have been written as either of the following:

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

Or perhaps it would be better if you just avoid the use of variables named X or B altogether.

(Back to title page)

Uppercasing by ARG and PULL

The ARG instruction is provided as an abbreviation of PARSE UPPER ARG. Likewise PULL is an abbreviation for PARSE UPPER PULL. Although both of these instructions are frequently required in REXX programs, the specification of UPPER, which causes automatic uppercase conversion of strings is somewhat unfortunate.

The reason this is done is to make string handling somewhat case insensitive. That is, if all strings can be treated as upper case, it is not necessary to deal separately with equivalent lower or mixed case strings. This can be a great convenience. But there are several drawbacks as well. In the first place, most PC users are more accustomed to dealing with strings in lower case (less frequent need to use the shift key).

But more significantly, the automatic upper casing by ARG and PULL can be unexpected and a source of subtle bugs. ARG in particular may be used heavily, so it's more often a problem. You may simply forget that it does upper casing and try to do comparisons using lower or mixed case strings against variables set by ARG.

An even more serious problem may occur anytime you pass binary data to a subroutine. Binary data may be read from a file, for example, or be 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!

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

(Back to title page)

Case sensitivity of labels

Normally a REXX program itself is mostly case insensitive. That is, you can usually write keywords, variable names, and labels in upper, lower, or mixed case, and it doesn't matter. REXX generally treats such things as if they were always upper case.

The SIGNAL instruction can be used to implement a kind of computed GOTO, though there are problems with this usage, since SIGNAL also is defined to terminate any open DO groups. Nevertheless, it can be a useful technique. It can, for example, be much more efficient than a SELECT statement that contains a large number of WHEN clauses.

```
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 */
...
```

But watch out for alphabetic case sensitivity! The value of the expression in the signal statement much match the alphabetic case of labels in the program exactly, even though REXX will always take the labels themselves to be upper case. If in this example we had used

```
signal value 'case'x
```

then an error 16 (label not found) would occur, since REXX would look for labels like **case1**, **case2**, etc., even though the actual labels in the program are **CASE1**, **CASE2**, etc. (in spite of how they are written!).

(Back to title page)

Null strings vs. omitted strings

In calls to REXX routines, including the main program, arguments can always be omitted as far as REXX is concerned. That is, the omission of an argument will not cause a REXX error (except for built-in functions, which often have required arguments).

There is an official technique using the ARG built-in function for determining in a subroutine whether an argument has been omitted:

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

(Note that there isn't any way to raise error 40 (incorrect call to routine) automatically as is done for built-in functions.) Most REXX programmers, however, tend to use the short-cut of testing an argument for a null string, since REXX supplies a null string whenever one attempts to refer to a missing argument:

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

This is not really a very good approach. In the first place, this example used ordinary comparison, so if a string consisting of multiple blanks had been passed (which might be meaningful), it would still be treated as if it had been

omitted. (See Strict vs. non-strict comparison.) But even if a strict comparison to the null string had been done, it would still complain if an explicit null string had been passed. Yet there might be valid reasons to accept a null string as an argument, but not the complete omission of the argument.

A different form of this problem can occur if you just blindly pass an argument to a built-in function:

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

If this program is run it will fail with error 40 in the call to MAX in case either argument has been omitted. This is because PARSE ARG will set **first** or **second** to a null string if the corresponding argument is omitted. But a null string is not the same thing as an omitted string, and MAX will fail if you pass it a null string, though not if you simply omit the second argument.

(Back to title page)

Distinction of commands and functions

The term "command" when used in a discussion of REXX is somewhat ambiguous. There are about 25 native "commands" in REXX, but the preferred term is "instruction" or "keyword instruction". Among these are ADDRESS, CALL, DO, EXIT, IF, PARSE, PULL, PUSH, QUEUE, RETURN, SAY, and SELECT. When such keywords are used in a REXX program, they are never quoted.

There is another class of REXX statements that consists of commands to an external command processor. Usually in OS/2 this is CMD.EXE, or else a REXX-enabled application. Such commands usually start with a keyword and this may be followed by one or more parameters. In order to avoid certian common problems (see Quoting literals) it is advisable to always enclose command keywords (and perhaps the entire command) in single or double quotes:

```
'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.

There is another kind of REXX service that is provided by "built-in functions". There are about 65 such functions, which include SUBSTR, POS, LINEIN, LINEOUT, MAX, VALUE, etc. All of these are technically functions in the REXX sense, which means that they return a value. However, some of them are frequently used for the "side effects" they produce rather than for their value - LINEOUT, CHAROUT, STREAM, and VALUE, for instance. Many other userful services are available through functions in external function packages such as REXXUTIL.

A very common error is to invoke such a function by writing it as a function call on a line by itself, as one does in a language like C. What actually happens then 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. Usually 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 else invoking it with a REXX CALL instruction. Note that using CALL means parentheses around the argument list should be omitted. (See CALL statement syntax.)

```
value('path', newpath, 'os2environment')        /* wrong */
x = value('path', newpath, 'os2environment')    /* right */
call value 'path', newpath, 'os2environment'    /* right */
```

(Back to title page)

Make a string all upper case or lower case

Uppercasing a string is easy. In fact, it can be done in a couple of different ways:

```
string = 'the owl and the pussycat'
up_string = translate(string)
parse upper var string up_string
```

The TRANSLATE function is actually capable of performing many interesting substitutions on a string, but the default when it is used with only one argument is to convert all alphabetic characters to upper case.

Lowercasing a string is a little harder. There is no analogous LOWER modifier on the PARSE instruction. In order to use TRANSLATE you must specify the upper case and lower case characters explicitly:

```
upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lower = 'abcdefghijklmnopqrstuvwxyz'
low_string = translate(string, lower, upper)
```

Note that the characters to be replaced appear as the third argument, and the characters they are replaced with appear as the second argument.

A slight simplification can be achieved by using the XRANGE function to create strings of upper and lower case letters:

```
upper = xrange('A', 'Z')
lower = xrange('a', 'z')
low_string = translate(string, lower, upper)
```

However, this is not recommended because it is not portable. Even though it works with ASCII data, since the internal codes for upper case and lower case letters are contiguous, it does not work with other encodings such as EBCDIC.

(Back to title page)

Trap ctrl-break and ctrl-c

Normally the execution of a REXX program can be halted by pressing ctrl-break or ctrl-c, just as for most other non-PM programs. This can be a problem, however, if the program has a need to finish what it is doing or otherwise clean up after itself before it terminates.

It is very easy to protect a REXX program against inadvertent or premature termination. In REXX terminology, the action of pressing ctrl-break is said to raise a condition called HALT. The default behavior of a REXX program when HALT is raised is to terminate, but it is possible to alter this default by providing a handler for the HALT condition and enabling it with a SIGNAL ON or CALL ON instruction.

In the simplest case, you may just want to perform some normal clean-up activities, such as erasing work files:

```
signal on halt
/* normal program logic */
...
exit
/* halt condition handler */
halt:
call sysfiledelete tempfile1
call sysfiledelete tempfile2
say 'Bye-bye!'
```

But in other cases, you may want to prevent accidental (or intentional) termination of the program. In this case you would probably enable the HALT condition handler with CALL ON instead of SIGNAL ON. The reason is that this allows the condition handler to return to the exact point in the program at which the condition was raised. The handler might do nothing else besides this, or it might perhaps give a useful message about the current state of the program:

```
call on halt
do case_number = 1 to case_total
    ...
end
...
/* halt condition handler */
halt:
say 'Program is not interruptible at this point.'
```

```
say 'Now processing case' case_number 'of' case_total'.'
return
```

One problem you should be aware of is that there is a bug in CMD.EXE which prevents REXX from being notified if ctrl-break or ctrl-c is used during the execution of a system command. CMD.EXE will immediately terminate the whole REXX script if it detects a ctrl-break while processing a system command.

In the case of .EXE files run from a REXX program, what happens after pressing ctrl-break depends on how the .EXE file itself handles the signal. It is possible for the .EXE file to handle such interrupts itself, in which case REXX may never know about it and the REXX program will continue to run. Or, the .EXE file may let CMD.EXE handle the signal, in which case both the .EXE and the REXX script will be ended.

(Back to title page)

Pass an array to a subroutine

Passing an array (or a general compound variable) to a subroutine or function is something one frequently wants to do in order to operate on the array as a whole, e. g. to sort it, or to easily process all elements of a large collection. Unfortunately, it is not so easy to pass an array to a subroutine in REXX as it is in other languages.

What you can do is to pass the "name" of the array to the subprocedure. That is, the stem of a compound variable. The first problem that arises is how to refer to the compound variable within the subprocedure. It is a problem because REXX does not perform substitution for the stem part of a compound variable name. Therefore, the following doesn't work:

```
call mysub 'data', count
...
mysub:
parse arg array, n
do i = 1 to n
    if array.i < 0 then
        array.i = 0
    ...
    end
```

Here REXX will refer to variables named ARRAY.1, ARRAY.2, etc. instead of DATA.1, DATA.2, etc. as intended. What you need to do is use the VALUE function to access the arrays. VALUE can be used both to read and write elements of the array:

```
call mysub 'data', count
...
mysub:
parse arg array, n
do i = 1 to n
    if value(array'.i') < 0 then
        call value array'.i', 0
    ...
    end
```

Another problem arises if you begin the subroutine with the PROCEDURE instruction, which is normally good programming practice. This ensures that the subroutine can't inadvertently modify variables it isn't supposed to. Unfortunately, PROCEDURE also protects the data in the array which the subroutine is trying to access. This compound variable should be exposed - but since stem name is itself a parameter to the routine, there isn't any way to do this.

The only thing that can be done here is a kludge, based on the fact that you can expose a list of variables in a PROCEDURE statement by keeping the list in another variable. If you know that there are only a limited number of array names that will be used by the subroutine, you can put then all in one variable:

```
names = 'x1. x2. x3. x4. x5. x6. x7. x8. x9. x10.'
call mysub 'x3', count
...
mysub: procedure expose (names)
```

Here the stem names have all been listed in the NAMES variable (each ending with a period to indicate it is a stem). The NAMES variable and everyting listed within it is exposed by PROCEDURE EXPOSE, since it is enclosed in parentheses.

If you don't know in advance which array elements (or tails of a general compound variable) might be used, there isn't much choice but to pass each stem name in a list contained in the value of another variable which is reserved for the purpose:

```
stemnames = 'data. list. things.'
call mysub stemname, count
...
mysub: procedure expose (stemnames)
```

Even this roundabout method breaks down if you need to pass an array to an external procedure (in a separate file). In that case, it just isn't possible to do it, since external procedures implicitly begin with a PROCEDURE statement in which nothing is exposed, and it is not legal to use an explicit PROCEDURE EXPOSE. In this case, you must adopt some other technique, such as passing the data in a file, through the external data queue, in a list contained in a single variable, etc.

As long as you are working with arrays that are represented as compound variables each of these lastter techniques requires you to access each element of the compound variable and make a copy of it, which means really abandoning the natural usage of REXX compound variables. Further, this means you can only work with compound variables that are integrally subscripted (as opposed to arbitrary string "subscripts") - unless you go to even more trouble.

For instance, you might work with a list of things contained in a separate string variable, delimited by blanks, e. g.

```
names = 'Allison Becky Colleen'
likes = 'chocolate flowers poetry'
```

"Arrays" of this sort are, of course, easy to pass to a subprocedure. And there is the additional advantage that there are a number of REXX built-in functions that are specialized to work with such blank-delimited lists of words (WORD, WORDPOS, SUBWORD, etc.). Such lists can be quite long, since there is no particular upper limit to the length of a REXX string, but long lists can be very inefficient (i. e. slow) to work with. Also, of course, the individual items in the list can't contain embedded blanks.

(Back to title page)

Return an array from a subroutine

The problem of returning an array from a subroutine is really just the same as the problem of passing one in. Indeed, if you pass the name of a stem to the subroutine and expose it as described elsewhere (how to pass an array to a subroutine) then whatever changes you make to the compound variable are still in effect after the subroutine returns. You might choose to pass the name of a compound variable which will be used only for output in order to get the effect of returning a collection of values.

Another technique you can use, especially when an external procedure is involved, so that you can't expose variables, is to pass data through the external data queue. You can even adapt this technique to return data for a general compound variable (i. e. one that doesn't have a simple positive integral tail).

Suppose that one variable called TAILS. holds the list of tails of some other compound variable, call it VAR. Suppose also that TAILS.0 contains the number of such tails. The following code might be used at the end of a subroutine to place values on the external data queue:

```
do i = tails.0 to 1 by -1
    temp = tails.i
    push temp var.temp
    end
push tails.0
```

In this example, we have intentionally placed things on the stack in reverse order ("LIFO") with PUSH so that if the stack already contains data, the existing data will not be disturbed. The last thing placed on the stack is the number of data elements just placed on the stack. Each data item on the stack contains both the tail and the corresponding value of the compound variable.

After the subroutine returns, this data can be retrieved from the stack with something like this:

```
pull taillist.0
do i = 1 to taillist.0
    parse pull tail value.tail
    taillist.i = tail
    end
```

```
taillist.0 = count
```

Each item on the stack contains both the tail name and the corresponding data. These can be retrieved simultaneously with the one PARSE PULL statement - provided no tails contain embedded blanks. We have also assigned the tail values to a new array (TAILLIST.) in order to be able to keep track of them and (perhaps) iterate through them later.

(Back to title page)

Format numbers neatly for output

REXX makes it easy to display output without need for concern about data types and the complex formatting statements of languages like FORTRAN and C. However, when one wants to display data in a structured form like a table, there isn't much alternative to supplying some formatting information.

If all you want to do is display data in a table, the RIGHT and LEFT built-in functions can be used, according to whether you want data right or left justfied within a column. It doesn't matter whether the data is all numeric, all character strings, or some combination. For instance, if you want to produce a table of countries, capital cities, and populations, you might use code like this:

```
say left('Country', 15)||left('Capital', 15)||left('Population', 10)
do i = 1 to n
    say left(country.i, 15)||left(capital.i, 15)||right(population.i, 10)
    end
```

The output might look like this:

```
Country        Capital        Population
Austria        Vienna            7500000
Denmark        Copenhagen        5118000
Switzerland    Bern              6289000
United Kingdom London           55883100
```

In this example we used an explicit concatenation operator. If you use implicit concatenation, don't forget to take into account the single blank that is inserted.

When you are producing formatted output involving numbers, the number of decimal places is frequently of concern. In addition, you may want control over whether or not exponential notation is used.

The TRUNC built-in function is the simplest way to specify how many decimal places should be displayed. The first argument of TRUNC is a number, and the second is the number of decimal places. Using a value of 0 for decimal places (the default) is also the easiest way to obtain the integral part of any number. As the name implies, TRUNC simply truncates a number. It does not perform rounding. If you request more decimal places than are present in the number, the remainder are set to 0.

If you want even more control over the display format of numbers, you can use the FORMAT built-in function. This allows you to specify the number of digits before, as well as after, the decimal point. For instance, a table of powers might be done like this:

```
x = -3.162
do i = 1 to 8
    say format(x**i, 5, 4)
    end
```

And the results would look like this:

```
   -3.1620
    9.9982
  -31.6144
   99.9649
 -316.0890
  999.4733
-3160.3346
 9992.9779
```

If you are working with numbers in exponential form, there are additional optional arguments of FORMAT that allow you to specify the number of digits to be used in the exponential part, and the number of total places required in the number in order to trigger its representation in the exponential format.

(Back to title page)

Write a line without a carriage return

The REXX SAY instruction always adds a carriage return and a linefeed at the end of any output you specify. (This is equivalent to the use of the LINEOUT built-in function when the first argument is omitted.) The effect is to move the cursor to the beginning of a new line on the screen.

If you want to avoid this new line effect, simply use the CHAROUT built-in function. You might want to do this to display several items using different REXX instructions, but have the output appear on the same line. Or you might want to display a prompt and have the typed input be on the same line. This could be done with:

```
call charout , 'Enter the number of your selection: '
parse pull select
```

(Back to title page)

Testing for keyboard input

There is a function called SysGetKey in IBM's REXXUTIL function package that allows for reading a single key at a time. However, if no key has been pressed, the function will wait indefinitely. If you need to be able to test whether a key has actually been pressed you can use the standard CHARS function to test whether a key is waiting to be read.

Here is how you might use this to wait up to 10 seconds for a reply before proceeding:

```
say 'Press any key to continue'
do 10
    if chars() \= 0 then do     /* no argument means standard input */
        key = sysgetkey()
        leave
        end
    call delay 1                /* pause 1 second */
    end
```

(Back to title page)

Store data in a program

Most programming languages provide explicit mechanisms for including large amounts of data within a program - either character strings or numbers. This is typically done by special syntax for initializing arrays. REXX does not have anything comparable to this.

The normal way of handling this is simply to use a number of assignment statements:

```
msg.1 = 'Try again.'
msg.2 = 'Sorry, wrong answer.'
msg.3 = 'Input not understood.'
```

This isn't necessarily as bad as it looks, since there is little overhead in performing assignments like this. You can put all such statements in a subroutine at the end of the source file to get them out of the way, and just call this subroutine when the program starts.

There is somewhat more of a problem when you want to use compound variables with non-numeric subscripts, since you can't use literals to specify a compound variable tail. What you wind up with is something like this:

```
x = 'Allison'
color.x = 'cyan'
x = 'Belinda;
color.x = 'magenta'
x = 'Cyndie'
color.x = 'yellow'
```

There is one trick you can use to avoid using a series of assignments when you have to incorporate a large amount of textual data - for instance program help text. Simply enclose the text in a comment, and use the SOURCELINE built-in function to process it or copy it into an array. You must begin the comment at some known line of the program, or else you can determine the current line number with the SIGNAL instruction.

```
...
signal around /*
```

```
  This example shows you how to initialize an array
  with several lines of text read from the program itself.
  You might use this to display help information.
*/
around:
j = 0
do i = sigl + 1
    line = sourceline(i)
    if line = '*/' then
        leave
    j = j + 1
    array.j = line
    end
```

There is one serious potential problem with this technique: SOURCELINE may not work if the program is tokenized and the source is discarded. This can happen if the program is saved in a macro space or you have used Personal REXX to process it and stripped off the source. It would also fail when the program is processed by a true compiler so that the original source code is not available.

(Back to title page)

Last updated: September 20, 1995 / October 24, 1995