BY LIONEL B. DYCK

How to Parse Command Line Parameters in REXX

While it is one thing to write a program in REXX, it is an entirely different process to enable the program to take command line arguments. This article demonstrates several different techniques you can use to process command line arguments in your REXX program.

THE REXX programming language is very flexible and powerful, but it lacks a simple built-in facility to extract arguments passed when you call a REXX program as a TSO command with arguments. The older, some say obsolete, TSO CLIST programming language supported command line arguments via the PROC statement, which has no equivalent in REXX.

However, by using the power of the *PARSE* function in REXX, you can easily come close to duplicating the TSO CLIST PROC functionality and with some additional coding, you can surpass it. Figure 1 shows the full PARSE syntax.

First, what is a command line argument? When you call a TSO command, you can provide additional information on the command line, such as arguments or parameters. The system passes this additional information to the called program. The REXX programming language provides a function called *arg*, which will take the command line arguments and put them into one or more REXX variables that you can use within the REXX program.

For example, if the user entered the command *test one two three*, the command name is *test* and the arguments are *one two three*. To access these arguments you would code the statement *arg options* to put all three arguments into the single variable options. You could also code *arg one two three* to place each argument into its own variable. These are called positional arguments because if you do not enter argument *two*, then argument *three* becomes argument *two*.

The *arg* function will translate the arguments into upper case. If you want to retrieve the arguments in mixed case use the statement *parse arg ...* which will take the arguments in whatever case the user entered them.

This is the simplest form of parsing the command line arguments.

Now, let's say you want to enter something other than positional arguments. For example, you want to use keywords. You would enter *test one(abc) two(def)* and you do not want to require that both arguments be entered. To deal with this, you could code the REXX statement shown in Figure 2. This works in the following way:

- Line 1 uses the *arg* function to pull the entire set of command line arguments and places them into the variable *options*. All of the arguments are converted to upper case for this example.

- Line 2 then parses the variable *options* using the *parse var* function, starting at position 1 looking for the literal *ONE(* and ending with *a)*. Any value found between the *ONE(* and the *)* will be placed into the variable one. Parsing then resumes starting at position 1, again looking for the literal *TWO(* and ending with another *)* and placing the data found between the *TWO(* and *)* into the variable *two*.

This simple parsing works and allows the user to enter only a single keyword with its parameter or to enter both keywords in any sequence.

Figure 3 shows a similar approach to parse the command: *test in dataset out dataset*. In this case, the parameters for the keywords are not enclosed in parentheses.

Line 1 uses the *arg* function to take all of the command line arguments and places them into the single variable *options*

while translating them to upper case. This is the default for *arg*; use *parse arg* to retrieve the passed data without case translation.

Line 2 inserts a single blank in front of the data in the variable *options*. This enables the next statement to find the string *IN* or the word *OUT* which requires that blanks surround them. If this statement is not surrounded by blanks, it is possible to find the literal *IN* or *OUT* within the parameters for the keywords.

Line 3 parses the variable *options* starting in position 1, looking for the literal *" IN "* and placing the next string into the variable *inda* and ignoring every string after that as indicated by the period. The parsing then resumes in position 1 looking for the literal *" OUT "* and placing the next word into the variable *outda* and ignoring every word after that as indicated by the period. Note that a string is a set of characters ending with a blank.

Lines 4 and 5 echo to the terminal the resulting values of *inda* and *outda*.

All of this is fine if all of your arguments are upper case, however that is not always the case (no pun intended).

To process input keywords and arguments with mixed case requires more extensive coding. Figure 4 shows an example of a simple case.

● Line 1 takes the command line
   arguments and places them into a
   single variable *option*. Because we
   are using parse *arg*, the case of the input will not be altered.

● Line 2 translates the variable *options* into upper case in variable *uoptions*.

● Line 4 uses the *parse* function to set variables *one*, *two*, and *null* to a null value. That way, if one of the keywords is not found, the associated variable will be null.

● Line 6 tests for the existence of the word *ONE* in the variable *uoptions*, and if the position is greater than 0,
   meaning that it was found, then lines
   7 to 9 are processed.

● Line 7 places the word position into variable *wp*.

● Line 8 extracts the word *wp+1* and places it into variable one from the variable options (the variable with the mixed case data).

● Line 9 is the end of the do loop.

● Line 10 tests for the word *TWO* in the variable *uoptions* and if found then
   executes the do loop in lines 11 to 13.

● Line 11 places the word *position* into variable *wp*.

● Line 12 extracts the word *wp+1* from the *options* variable into variable *two*.

● Line 13 is the end of the do loop.

● Lines 15 and 16 use the *say* function to echo to the terminal the values that were found.

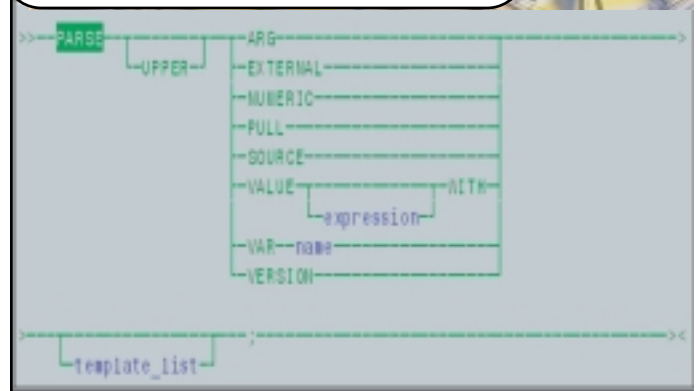FIGURE 1: THE PARSE SYNTAX (FROM IBM Z/OS REXX REFERENCE SA22-7790-02)

FIGURE 2: PARSING KEYWORDS

```
1. arg options
2. parse var options 1 "ONE("one")" 1 "TWO("two")"
```

FIGURE 3: PARSING KEYWORDS — NO PARENTHESIS

```
1. arg options
2. options = " "options
3. parse var options 1 " IN " inda . 1 " OUT " outda .
4. say "IN: " inda
5. say "OUT:" outda
```

FIGURE 4: PARSING MIXED CASE ARGUMENTS

```
1. parse arg options
2. uoptions = translate(options)
3.
4. parse value "" with one two null
5.
6. if wordpos("ONE",uoptions) > 0 then do
7.    wp = wordpos("ONE",uoptions)
8.    one = word(options,wp+1)
9.    end
10. if wordpos("TWO",uoptions) > 0 then do
11.    wp = wordpos("TWO",uoptions)
12.    two = word(options,wp+1)
13.    end
14.
15. say "one:" one
16. say "two:" two
```

The creation of a variable called *null* with a null value is very helpful in self documenting your code, as it is clearer to test for a variable of *null* than to test for "".

These are just a few of the many options you can use to process command line arguments in your REXX program.

USEFUL REXX REFERENCES

● The REXX Language Association on the Web at www.rexxla.org.
● An excellent list of REXX publications on the Web at www2.hursley.ibm.com/ rexx/rexxbook.htm maintained by the originator of REXX, Mike Cowlishaw.
● The IBM REXX Web site at www2.hursley.ibm.com/rexx.
● The IBM REXX Product Web page at www-4.ibm.com/ software/ad/rexx.

- *The IBM z/OS 1.2 TSO/E REXX Reference* at http://publibz.boulder.ibm.com/epubs/pdf/ikj4a310.pdf.
- *The IBM z/OS 1.2 TSO/E REXX User's Guide* at http://publibz.boulder.ibm.com/epubs/pdf/ikj4c310.pdf.
- *The IBM z/VM 4.2 REXX/VM Reference* at www.vm.ibm.com/pubs/pdf/hcse2a30.pdf.
- *The IBM z/VM 4.2 REXX/VM User's Guide* at www.vm.ibm.com/pubs/pdf/hcsb3a20.pdf.
- The IBM REXX Compiler Web site at www-4.ibm.com/software/ad/rexx/rexxcompd.html.

If you are new to REXX, be sure to check out Jeff Glatt's Web site, which claims you can "Learn REXX Programming in 56,479 steps" at www.borg.com/~jglatt/rexx/scripts/language/language.htm.

You can also find numerous REXX examples for OS/390 and z/OS at my Web site at www.lbdsoftware.com. Look on the ISPF Tools and Toys page and the TCP/IP page.

NaSPA member Lionel B. Dyck is a lead MVS systems programmer for a large HMO in California. He has been in systems programming since 1972 and has written numerous ISPF dialogs over the years. His current project is evaluating Linux on the S/390 and zSeries platform. Lionel is an active member of NaSPA and SHARE, and can be contacted via email at Lionel.B.Dyck@kp.org.