REXX LANGUAGE PARSING CAPABILITIES

KEITH WATTS
KILOWATT SOFTWARE

REXX language parsing capabilities

Keith Watts (Kilowatt Software)

Abstract

The REXX language has several powerful features which distinguish it from other programming languages that are generally available. Among these are the language's intricate collection of parsing capabilities. These enable the programmer to easily divide character strings by a diversity of methods. Herein, the syntax and semantics of these methods are described in detail. This paper is intended to help programmer's of varying proficiency gain a commanding grasp of these concepts. Many examples are also provided.

One of the more powerful features of REXX is its ability to parse. However, if you are like many others who are learning REXX you are unfamiliar with the word "parse". Webster's New World Dictionary contains the definition:

> **parse** *vt*, *vi* **parsed, pars'ing** [Now Rare]
>
> 1.to separate (a sentence) into its parts, explaining the grammatical form, function, and interrelation of each part 2. to describe the form, part of speech , and function of (a word in a sentence)

The above definition has little in common with the REXX parsing capability. The key phrase is: "to separate into its parts". For the word "parse" is computer science parlance for the act of separating computer input into meaningful parts for subsequent processing actions.

REXX is one of few languages which provides parsing as a fundamental statement. Most languages merely provide lower level string separation capabilities, leaving the preparation of parsing capabilities as user developed enhancements. Within REXX, these capabilities are immediately available, and as you will soon find, very powerful.

Let us learn about parsing by analyzing the following:

```
phrase = "  I think, therefore   I am [I  think].  "
```
[1]

If you scrutinize the above, you will notice that there are extra blanks at various points within the phrase. These extra blanks and the punctuation characters within the phrase complicates the parsing process.

The words within the phrase could be traditionally extracted as follows:

```
 /* try1 [the brute force approach] */
/* trace ?i */                  /* turn on the trace to see how this code works */
remaining_words = phrase
do i=1 by 1 while remaining_words <> ""
  remaining_words = strip( remaining_words, "Leading" )      /* skip lead blanks */
  end_pos = pos( " ", remaining_words )            /* locate blank after current word*/
  word.i = substr( remaining_words, 1, end_pos )                /* extract a word */
  remaining_words = substr( remaining_words, end_pos )        /* step over word */
 end
```

[1]This adaptation of Descartes famous insight is from "On the Threshold of a Dream", by the Moody Blues.

Alternatively, REXX contains built-in functions which are excellent for extracting words from phrases, as follows:

```
 /* try2 */
/* trace ?i */                  /* turn on the trace to see how this code works */
do i=1 for words( phrase )
  word.i = word( phrase, i )
 end
```

Finally, an approach which uses REXX parsing is:

```
 /* try3 */
/* trace ?i */                  /* turn on the trace to see how this code works */
rest = phrase
do i=1 while rest <> ""
  parse var rest word.i rest
 end
```

Of the 3 approaches shown above, the second is clearly the best choice for separating a string into words. However, the second approach is specifically capable of accessing words, it is inadequate for other parsing tasks. The third approach is slightly more intricate than the second, and is occasionally preferrable. All that can be said about the first approach is that it successfully obtains individual words, and the method used is familiar to those who have programmed with other languages; though the subroutine names are probably different.

Let REXX know what you mean

Notice that words within the phrase above are generically captured by relative position into the set of **word.i** symbols. Now you will see how phrases can be parsed into symbols which are syntactically significant.

For example, we can divide our phrase into meaningful constituents as follows:

```
parse var phrase precondition ',' consequence '[' qualifier ']'
```

This results in the following symbol assignments:

```
precondition          "I think"
consequence           "therefore  I am "
qualifier             "I  think"
```

[Please observe that there are extra spaces within the `consequence` and `qualifier` symbols].

Notice how easy it was to divide our phrase with the parse statement.. This partitioning can not be done by modifying the `try2` example shown earlier. The `try1` example can be modified with considerable effort to extract the precondition, consequence, and qualifier symbols based on syntactic dividers. However, the resulting code would be far more intricate than the simple parse statement above. Furthermore, revision of the brute force method requires similar complexity and effort as other parsing challenges arise.

For a more familiar example consider the following:

```
parse value "Sam likes green chili pizzas" with subject verb entree
```

The result of this parsing operation leads to the following symbol assignments:

```
subject     "Sam"
verb        "likes"
entree      "green chili pizzas"
```

Syntactic elements are now associated with meaningful symbols, instead of generic symbols **word.1**, **word.2**, etc.

How does parsing work

The REXX parse statement divides a *source string* into constituent parts and assigns these to symbols as directed by the governing parsing *template*. The parse statement has the following general form:

```
parse[2] [upper] source_identification  symbol_and_rule_template
```

Where:

upper

This is an *optional* keyword. When it is present, all values assigned to symbols are converted to upper case.

source_identification

This identifies where the *source string* for parsing is obtained. This is one of the following:

◊ ARG

Example: `parse arg a1.1 a1.rest , a2, a3.1 . , a4`

The ARG keyword indicates that one or more procedure arguments are to be processed as source strings. This is the only keyword which can have multiple source strings. Each argument passed to an internal or external procedure corresponds to the clauses separated by commas in the template pattern above. However, only 1 argument string is available for processing by the topmost procedure level associated with an EX command.

In the example above, the first word in the first argument string is assigned to symbol *a1.1* and the remainder of the first argument string is assigned to symbol *a1.rest*. The

---

2The PARSE keyword itself is omitted in ARG and PULL statements, which are actually shorthand forms for PARSE UPPER ARG ... and PARSE UPPER PULL ... respectively. Both of these forms assign uppercase values to associated symbols. The longer forms PARSE ARG and PARSE PULL must be used when you want to preserve mixed case values during assignment.

entire second argument string is assigned to symbol *a2*. The first word in the third argument string is assigned to symbol *a3.1*. The entire fourth argument string is assigned to symbol *a4*. Additional argument strings are ignored.

Empty strings [""] are assigned to all remaining symbols that appear in the template when insufficient source argument strings are available.

◊ LINEIN

Example: `parse linein first_letter 2 0 whole_line`

The LINEIN keyword indicates that the source string is obtained by reading one line from the default input stream.

In the example, the first letter within the line is assigned to symbol *first_letter* and the entire line, including the first letter, is assigned to symbol *whole_line*. When an empty line is read, then the empty string "" is assigned to *first_letter*.

◊ PULL

Example: `parse pull qline`

The PULL keyword indicates that the source string is obtained by extracting the topmost line from the external data queue.

If there are NO lines within the queue a line is obtained from the default input stream instead. This can be troublesome in numerous ways. First, if your program uses other stream functions to process lines from the default input stream, it is easy to overlook lines that are accidentally acquired by a PARSE PULL or PULL request.

Second, in many REXX environments, there is no indication that your REXX program is expecting input from the keyboard. This will cause you to MISTAKENLY believe that your session is HUNG ! Rather than automatically restarting your session, always try to type characters at the keyboard first. If you can type, your program is reading keyboard input. It is strongly recommended that you always precede keyboard input requests with prompt messages. And, you should assert that lines remain in the queue before performing PARSE PULL and PULL requests as follows:

```
if queued() = 0
  then
    if lines() = 0
      then do
         say "No more terminal input is available for parsing"
         exit 86
       end
    else
      say a_meaningful_prompt_message

parse pull keyboard_wd1 etc
```

Finally, when end of file has already been reached in the default input stream, the source string for parsing is the empty string [""]. This assigns the empty string to all

symbols that appear in the template. This can lead to unusual difficulties later during your program's execution.

You should activate the trace facility when you are developing programs that use the PARSE PULL and PULL statements, or perform other default input stream operations. Then, a helpful trace message will let you know that your program is waiting for keyboard input to complete.

◊ SOURCE

Example:

```
parse source environment proc_kind src_file proc_name implementation
```

The SOURCE keyword indicates that the source string is internally prepared by REXX with information describing the procedure's execution environment.

Within Portable/REXX™ the following symbol assignments can be expected:

environment	PCDOS | PCWIN
proc_kind	COMMAND [top level procedure] FUNCTION [procedure executing as function] SUBROUTINE [procedure was invoked by CALL statement] CALLONTRAP [procedure servicing CALL ON error handler]
src_file	the name of the file containing procedure source statements
proc_name	the current procedure's name
implementation	Portable/REXX [always]

◊ VALUE expr WITH

Example: `parse value getkey() with scan_code 2 key_code`

The "VALUE expr WITH" form establishes the result of any REXX expression as the source string to parse.

In the example above, the special Portable/REXX™ Getkey built-in function is used to obtain the double-byte code for a keyboard input action. The parse template indicates that the first byte is assigned to symbol *scan_code* and the second to symbol *key_code*. If the user had pressed function key "F1" then *scan_code* would be "3B"x and *key_code* would be "00"x.

◊ VAR variable_name

Example: `parse var rest word1 rest`

The VAR keyword indicates that the value of a symbol is the source string to parse.

In the example above, the source string is the value of symbol *rest*. The first word in this string is assigned to symbol *word1*. The remainder of the string is reassigned to symbol *rest*. Thus, every time this statement is executed, the first word is extracted, and the number of words associated with symbol *rest* is reduced by one.

◊ VERSION

Example: `parse version lang_ident lang_level release_date`

The VERSION keyword indicates that the source string is internally prepared by REXX with information which distinguishes the language implementation.

Within Portable/REXX™ for MS-DOS® the symbol assignments of the following form can be expected:

lang_ident	REXX-KilowattSoftware-Portable-BV112 [or later release]
lang_level	4.00 [or higher]
release_date	9 May 1991 [or later]

symbol_and_rule_template

This is the *template* which specifies how to parse the source string, so that symbol values can be assigned. The template can be omitted from the parsing statement. When the template is absent, the source string to parse is STILL prepared! This preparation may remove a line from the external data queue, perform a file input operation, or compose associated values when PULL, LINEIN, and VALUE are requested.

The parsing template has the following general form. Some templates can be significantly different. For example, the leading item can be a division specifier, and multiple division specifiers can appear without an intervening symbol name.

parse template form

```
symbol_1  division_specifier_1  symbol_2  division_specifier_2   etc...
```

The first character of each parsing template element is sufficient to distinguish whether it is a symbol name or a division specifier. The element is a symbol name, when the first character is an eligible symbol name character. Division specifiers are one of the following, with examples of each shown underneath:

◊ space_delimiter

Example: subject verb entree

When spaces separate symbol names within a template, then each word of the source is assigned to each corresponding symbol identified in the template. If there are more words in the source than there are names in the template, then the remainder of the source is assigned to the last symbol. All spaces within the remaining portion of the source string are preserved in this last symbol's value. If there are insufficient words in the source string for all template symbols, then words are assigned on a one-to-one basis to the leading symbols, and the empty string "" is assigned to all remaining template symbols.

Tabs are considered equivalent to spaces with respect to the *space_delimiter* division specifier. Tabs are preserved by all other division specifiers.

◊ literal_pattern

Example: "," consequence '[' qualifier ']'

Literal patterns are quoted strings within the pattern. These strings usually contain a single character, but may include many characters as well as spaces. In the example above, these are separated from other template items by spaces. However, these spaces are not necessary. Literal patterns can be immediately adjacent to other terms, as in the following example. Presume:

```
time()          15:27:14
```

Then

```
parse value time() with hour":"minute":"second
```

Causes the following symbol assignments:

```
hour            15
minute          27
second          14
```

The source string is searched from the current position until an exact match with the literal pattern is located. If the literal pattern is found within the source string, then the prior symbol is assigned all characters, including spaces, up to the last character preceding the matching source position. The characters in the source which match the literal pattern are skipped. The next character to be assigned is that which immediately follows the last character in the source string that matched the literal pattern.

◊ variable_pattern

Example: `before (delim) after`

Variable patterns are very similar to literal patterns. The only difference is that the pattern to match is the value of the parenthesized symbol name.

In the example, the value of symbol *delim* is used as a pattern. The part of the source string which precedes the pattern is assigned to symbol *before* , and the part which follows the pattern is assigned to symbol *after*. Presume the following:

```
rel_date        19 Dec 1990

delim           Dec
```

Then

```
parse var rel_date before (delim) after
```

Causes the following symbol assignments:

```
before          19_
after           _1990
```

[The '_' characters above indicate invisible spaces in assigned symbol values].

◊ column#

Examples:

```
hour 3 4 minute 6 7 second 9                    [parses: 12:44:37]

first_letter =2 1 whole_line

head =(offset) tail
```

Absolute columns are distinguished as numbers within the template, numbers preceded by an equals marker [=], or a variable reference which is also preceded by an equals marker. Absolute column 1 prepares for subsequent access to the 1st character in the source string, column 2 for the second character, etc. A column specification of 0, causes the 1st character to be accessed. Column specifications which exceed the source string length are truncated to the number of characters within the source string.

In the first example above, the following assignments occur:

```
hour      12
minute    44
second    37
```

In the third example, the value of symbol offset identifies where the source string is partitioned for assignment to symbols head and tail .

◊ relative_column

Examples:

```
hour +2 +1 minute +2 +1 second +2                    [parses: 12:44:37]

first_letter +1 0 whole_line

item1 +(width1) item2 +(width2) item3 +(width3)
```

Relative columns are distinguished as signed numbers within the template, or variable references preceded by plus and minus signs. Negative relative column motions can not access character positions less than the first, and positive motions can not access characters after the last.

In the first example above, the following assignments occur:

```
hour      12
minute    44
second    37
```

In the last example, symbols item1 , item2 , and item3 receive values from the source string according to the values of the corresponding width variables.

◊ period

Example: `file_name . revision_date .`

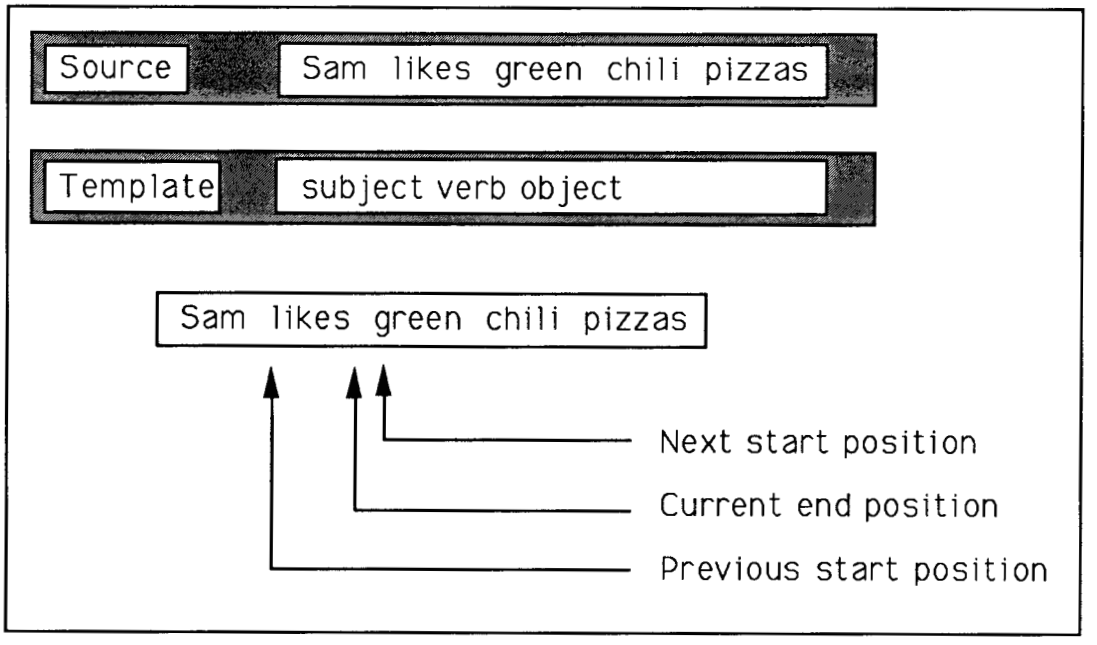
Periods within the template act as *placeholder* symbol names. These absorb values which would have been assigned to symbol names instead. A trailing period absorbs the remainder of the source string.

◊ comma

Example: `arg1_wd1 arg1_wd2 ., arg2_wd1 etc...`

Commas within the template are only used when multiple argument strings are processed by internal and external procedures. Hence, these are only valid when the ARG *source_identification* keyword is in effect. Only one argument source string is available for the topmost REXX procedure level. A comma in the template indicates that parsing of the current argument source string is to be discontinued, and processing ensues from the beginning of the next argument source string.

The following picture may help you to understand how parsing is performed.



While the *template* is processed from left to right, current positions in the *source string* are maintained. The motion of these positions is guided by the division specifiers within the template. This motion is toward the right, except when an absolute position or negative relative motion is specified. The initial start position is position 1, which corresponds to the first character at the leftmost end of the source string. An absolute position less than 1 is revised to be 1, as are negative relative motions which would precede the first source character. Likewise, the highest end position is the rightmost end of the source string.

The above picture shows positions associated with the *space_delimiter* which separates the *verb* and *object* symbol names in the template. The previous start position locates the "l" in "likes". The current end position locates the space between "likes" and "green". The next start position locates the "g" in "green". With these positions established, the word "likes" is assigned to the symbol name "verb". As only the object name remains in the template, the remainder of the source string from the next start position is assigned to symbol name *object*. This is the phrase "green chili pizzas". If there had been multiple spaces between the words

"likes" and "green" then the next start position would have located the second intervening space.

Power parsing

Now two common applications of parsing will be studied. The first shows how to meaningfully extract variable length text information from MS-DOS® files. The second shows how to extract fixed length information from files.

Parsing variable width text fields

Assume that you want to analyze information in a name&address file. Each line of information contains multiple fields of varying length. The fields are separated by tab characters ["09"x]. The file could have been obtained by extracting rows from a database or spreadsheet program. Alternatively, it could have been created by a REXX program which wrote lines with the following request.

```
tab = "09"x

call lineout "nad", ,
  fname ||tab|| mname ||tab|| lname ||tab|| company ||tab|| addr_line1 ||tab|| addr_line2 ||tab|| ,
  city ||tab|| state ||tab|| zip ||tab|| phoneno
```

The parsing of input lines into meaningful fields has the same structure, and uses the tab symbol as a variable pattern specification. Fields can be obtained as follows:

```
tab = "09"x

parse value linein( "nad" ) with ,
  fname (tab) mname (tab) lname (tab) company (tab) addr_line1 (tab) addr_line2 (tab) ,
  city (tab) state (tab) zip (tab) phoneno
```

Fixed width binary data analysis

Instead of a file containing variable width fields, suppose you have a file containing fixed width character fields and binary-encoded numbers This file could have been created by a REXX program which wrote lines with the following request.

```
call charout "tranfile.db", ,
  left( partno, 8 ) || left( serialno, 8 ) || d2c( unit_price, 2 ) || d2c( quantity, 2 ) || ,
  d2c( subtotal, 4 ) || d2c( tax, 4 ) || d2c( total, 4 )
```

The parsing of this information into meaningful fields has a similar structure, with an extra step to convert each binary-encoded value to a corresponding decimal value. Fields can be obtained as follows:

```
parse value charin( "tranfile.db",, 32 ) with ,
  partno +8 serialno +8 unit_price +2 quantity +2 subtotal +4 tax +4 total +4

unit_price    = c2d( unit_price )
quantity      = c2d( quantity )
subtotal      = c2d( subtotal )
tax           = c2d( tax )
total         = c2d( total )
```

This concludes the description of how to perform parsing operations in REXX. To fortify your understanding of parsing you should now try some experiments of your own choosing. You should also read the section titled "Parsing for ARG, PARSE, and PULL" in "The REXX Language".

This paper is an excerpt from:
Learning to Program with Portable/REXX™

which is published by Kilowatt Software at the following address:

1945 Washington St, #410 San Francisco, CA 94109-2968 (415) 346-7353