ARx_Instr2.ag

COLLABORATORS

	TITLE : ARx_Instr2.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		October 17, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARx_Instr2.ag

1.1 "

```
AN AMIGAGUIDE® TO ARexx                      Second edition (v2.0)
by Robin Evans

   Note: This is a subsidiary file to ARexxGuide.guide. We recommend
   using that file as the entry point to this and other parts of the
   full guide.

      Copyright © 1993,1994 Robin Evans.  All rights reserved.
```

1.2 ARexxGuide | Instruction Reference (13 of 25) | NUMERIC

```
                        | DIGITS [<expression>]
         |    FUZZ [<expression>]
NUMERIC  |       | [SCIENTIFIC]
         | FORM | [ENGINEERING]
         |       | [[VALUE] <expression>]
```

Controls the precision of numbers used in and returned by arithmetic
operations.

ARexx will round a numeric value that would otherwise be expressed as a
value greater than the number of digits specified by NUMERIC DIGITS to the
closest value with that number of digits.

NUMERIC DIGITS controls the precision with which arithmetic operations
are performed. If <expression> is supplied, it must evaluate to a
positive whole number between 1 and 14. If it is excluded, then the
default precision of 9 is used.

   The DIGITS() built-in function returns the current DIGITS setting.

NUMERIC FUZZ controls the precision with which numeric comparisons take
place. If <expression> is supplied, it must evaluate to a positive whole
number less than the current setting of NUMERIC DIGITS. The effect of the

FUZZ setting is to decrease the value of DIGITS during any numeric
comparison.

   The FUZZ() built-in function returns the current FUZZ setting.

NUMERIC FORM controls the type of exponential notation to be used when
numbers are presented. With SCIENTIFIC notation (the default), only 1
non-zero digit is used before the decimal point. Under ENGINEERING
notation, the power of 10 will always be a multiple of 3.

If an <expression> is used, it must evaluate to either 'SCIENTIFIC' or
'ENGINEERING'. The VALUE sub-keyword may be omitted if the expression does
not begin with a symbol or a literal string.

If NUMERIC FORM is used without other options, the default value will be
restored.

   The FORM() built-in function returns the current FORM setting.

            Interactive example: Demonstrate NUMERIC settings   *

   Also see:
                PARSE NUMERIC
                                Technique note: Format() user function

1.3 ARexxGuide | Instruction Reference (14 of 25) | OPTIONS

                        | RESULTS |
OPTIONS | PROMPT <expression> | [{ON | OFF}]
        | FAILAT <expression> |
        | CACHE |

Sets certain internal defaults for the currently executing ARexx program
or function.

OPTIONS RESULTS instructs ARexx to retrieve the string result field that
is set by many external programs. The value will be assigned to the system
variable RESULT .

OPTIONS PROMPT sets the prompt string used with the instructions PULL,

                PARSE PULL
              , or
                PARSE EXTERNAL
              . <expression> can be anything that
can be printed by the shell and can include ANSI escape sequences to
change colors, clear the screen, or reposition the cursor.

OPTIONS FAILAT sets the error number that will be considered a failure
condition if returned by an external command. The default failure level is
inherited from the calling environment. It is usually 10 unless changed by
the AmigaDOS command Failat. If the SIGNAL ON FAILURE trap is set, it

will be triggered at the value set by this instruction.

OPTIONS CACHE is a rarely used option that in ON by default and causes
ARexx to keep some repetitive instructions in memory for quicker
execution. Turning it off will slow down most scripts without much benefit.

1.4   ARexxGuide | Instruction Reference (15 of 25) | PARSE

```
                              |
                         ARG
                      |
                |
                         PULL
                      |
                |
                         EXTERNAL
                      |
                |
                         NUMERIC
                      |
PARSE [UPPER] |
                          SOURCE
                      |
                       <template>
                     ;
                |
                         VERSION
                      |
                |
                         VALUE <expression> WITH
                      |
                |
                         VAR <variable>
                      |
```

The more generalized form of this instruction is:

PARSE [UPPER] <source> <template> [, <template>]

Assigns values to one or more variables in <template> from a string
supplied by the source specified in the second option.

The assignment is based on positional or pattern characteristics in the
<template>.

If used, the first option, UPPER, will cause any alphabetic characters in
the <source> string to be translated to uppercase before assignments are
made.

   Compatibility issues:

              ARexx departures from REXX-standard PARSE

1.5 ARexxGuide | Instruction Ref. | parse (1 of 8) | ARG

```
PARSE
[UPPER] ARG <template>
```

   The ARG, PULL, and EXTERNAL options retrieve information from an
   external environment that can then be used within the script.

The ARG option specifies that the source string is to be supplied by the
arguments that were used to call the current program or subroutine.

ARG can be used as a keyword by itself, which makes it identical to the
instruction 'PARSE UPPER ARG ...'.

   Example:
      <the program is called with the command 'rx Prg MyFile.doc DOCS:'>
      /**/
      PARSE ARG File Device .
      Say File   >>> MyFile.doc
      Say Device   >>> DOCS:

All variations of
                <template>
                can be used with this option.

If
                multiple templates
                are used, the instruction will read the next
 argument string for each of the templates. Multiple argument strings
are available, however, only if the instruction is used in a script or
subroutine invoked by another ARexx script. AmigaDOS sends a single
argument string in which commas are treated as literal characters; the
commas do not divide the string into multiple arguments.

   Also see ARG()   function

The external library package RexxDosSupport.library , by Hartmut Goebel,
includes a function, ReadArgs(), that uses parsing routines supplied by
the operating system to split a command-line argument string into its
component values, each of which is assigned to a variable name supplied by
the program in a template. The function can greatly simplify the handling
of command arguments.

1.6 ARexxGuide | Instruction Ref. | parse (2 of 8) | PULL

```
                    PARSE
                    [UPPER] PULL <template>
```

   The ARG, PULL, and EXTERNAL options retrieve information from an
   external environment which can then be used within the script.

The PULL option specifies that the source string is to be retrieved from
 STDIN , which is ususally the shell from which the script was launched.
If data has been stacked to the shell using the PUSH or QUEUE
instructions, then the top-most item on the data-stack will be returned
with this instruction. Otherwise, the instruction causes the script to
pause and wait for input from the user. The input value will be retrieved
when the <Enter> key is pressed.

PULL can be used as a keyword by itself in which case it is identical to
the instruction 'PARSE UPPER PULL ...'.

```
   Example:
     /**/
     PARSE PULL File Device .
        /* the user types 'MyFile.doc DOCS:' and presses <Enter> */
     Say File                                  >>> MyFile.doc
     Say Device                                >>> DOCS:
```

The instruction OPTIONS PROMPT <string> can define a prompt string that
will be presented to the user whenever this instruction is issued.

All variations of <template> can be used with this option.

If
                  multiple templates
                  are used, the instruction will pull a new string
for each of the templates.

```
              More information:  Redirection of standard input
                  Technique note:  Data scratchpad with PUSH & QUEUE
```

1.7 ARexxGuide | Instruction Ref. | parse (3 of 8) | EXTERNAL

```
                    PARSE
                    [UPPER] EXTERNAL <template>
```

   The ARG, PULL, and EXTERNAL options retrieve information from an
   external environment which can then be used within the script.

The EXTERNAL option works in the same way as the PULL option except that
it will pull its string from the logical device defined as STDERR ,
rather than the device STDIN which is used by PULL.

The instruction OPTIONS PROMPT <string> can define a prompt string that
will be presented to the user whenever this instruction is issued.

The logical device STDERR is normally undefined unless the global tracing
console is open. ARexx will always define the tracing console as STDERR
which means that this instruction would retrieve its input from that
window. If the tracing console is not open, it is possible to define the
currently active console window as STDERR with this clause:

```
    call open(STDERR, "*", 'R')
```

All variations of <template> can be used with this option.

If
                multiple templates
                are used, the instruction will pull a new string
for each of the templates.

1.8   ARexxGuide | Instruction Ref. | parse (4 of 8) | NUMERIC

```
                PARSE
                [UPPER] NUMERIC <template>
```

   The NUMERIC, SOURCE, and VERSION options supply information about the
   current program and the system on which it is running.

The NUMERIC option supplies a string indicating the current settings of

                NUMERIC
                options in the following format:

```
    <digits> <fuzz> <form>
```

where the first two tokens are numbers indicating the setting of NUMERIC
DIGITS and NUMERIC FUZZ. The third token is either 'SCIENTIFIC' or
'ENGINEERING' and indicates the current setting of NUMERIC FORM.

The same information, individually, is available using the built-in
functions DIGITS() , FUZZ() , and FORM() .

```
    Example:
    /**/
    PARSE NUMERIC NumOpts
    SAY NumOpts                                  >>> 9 0 SCIENTIFIC
    NUMERIC DIGITS 7
    NUMERIC FUZZ 2
    NUMERIC FORM ENGINEERING
    PARSE NUMERIC NumOpts
    SAY NumOpts                                  >>> 7 2 ENGINEERING
```

All variations of
                <template>
                can be used with this option.

```
If
                multiple templates
                are used, a new copy of the source string will be
supplied to each of the templates.
```

Next: PARSE SOURCE | Prev: PARSE EXTERNAL | Contents: PARSE

1.9   ARexxGuide | Instruction Ref. | parse (5 of 8) | SOURCE

```
                PARSE
                [UPPER] SOURCE <template>
```

   The NUMERIC, SOURCE, and VERSION options supply information about the
   current program and the system on which it is running.

The SOURCE option supplies information about the calling environment for
the current program.

The string is formatted as:

   <type> <results> <called> <resolved> <extension> <host>

<type> is either COMMAND or FUNCTION, and indicates whether the program
was called as a program with a command (for example 'rx Prg') or as an
external function.

<results> is either 0 or 1 -- a Boolean value that indicates whether
a results string was requested.

<called> is the actual command (or function name) that was used to
initiate the script.

<resolved> is the resolved name of the program -- the complete path and
file name used by ARexx to call the program.

<extension> is the file extension currently used for searching the rexx:
directory. The default is '.rexx', but is changed by many application
programs.

<host> is the default address when the program begins -- the same as the
result of the ADDRESS() function if called when the program first began.

The following example will produce slightly different results on different
systems. In this example, its output assumes that the program is stored as
'test.rexx' in the 't:' directory, and is called, initially, with the
command 'rx t:test'.

```
   Example:
      /* test.rexx */
      PARSE SOURCE CallType SrcStr
      SAY CallType SrcStr
      if CallType = 'COMMAND' then
          CALL 't:test'
```

The program outputs the source string first when called from a shell, and
then calls itself as an external function, which prints the string again.
This is one possible result:

        COMMAND 0 t:test Ram Disk:T/test.rexx REXX REXX
        FUNCTION 1 t:test Ram Disk:T/test.rexx REXX REXX

All variations of
                <template>
                can be used with this option.

If

                multiple templates
                are used, a new copy of the source string will be
supplied to each of the templates.

    Also see ADDRESS
            Determining initial host

Next: PARSE VERSION | Prev: PARSE NUMERIC | Contents: PARSE

1.10   ARexxGuide | Instruction Ref. | parse (6 of 8) | VERSION

                PARSE
                [UPPER] VERSION <template>

    The NUMERIC, SOURCE, and VERSION options supply information about the
    current program and the system on which it is running.

The VERSION option supplies information about the of the REXX language
processor, and about the system on which it is running. The string is
formatted as:

    <lang> <version> <processor> <math proc> <video std> <video freq>

<lang> is the language -- 'ARexx' in virtually all cases on the Amiga. It
could be REXX370 or REXXSAA if the script is run on other systems.

<version> is the version number of the language.

<processor> indicates the processor under which the program is running.
(Version 1.15 may incorrectly report that a 68040 processor is a '68070').

<math proc> is either 'NONE' or the number of the math coprocessor on the
current system.

<video std> will be 'PAL' or 'NTSC' in most cases and indicates the video
standard being used on the current system.

<video freq> indicates the clock frequency for the current machine and
will be either '50HZ' or '60HZ'.

The result of the following program will be slightly different on
different systems:

```
      /**/
      PARSE VERSION VerStr
      SAY VerStr
```

The result might be:

```
        ARexx V1.15 68030 NONE NTSC 60HZ
```

All variations of
                <template>
                can be used with this option.

If
                multiple templates
                are used, a new copy of the source string will be
supplied to each of the templates.

1.11   ARexxGuide | Instruction Ref. | parse (7 of 8) | VALUE

```
                PARSE
                [UPPER] VALUE [<expression>] WITH <template>
```

   The VALUE and VAR options allow internal program values to be used with
   the PARSE instruction.

The evaluated result of <expression> is provided as the source string.
The sub-keyword WITH must be supplied, since, without it, there would be
no way to determine where the expression ends and the <template> begins.

   Examples:
```
      /**/
      PARSE VALUE  time()  WITH hr ':' min ':' sec
      SAY min 'minutes and' sec 'seconds after' hr
```

The output might be

```
        24 minutes and 22 seconds after 12
```

<expression> can be any valid expression, although if it is a single
variable, it could then be processed with PARSE VAR <template> .

If <expression> is not included, then the null string will be used. This
will set several variables to null with a single instruction.

```
      /**/
      PARSE VALUE WITH Var1 Var2 Var3
      Say Var1 Var2 Var3
```

The output will be an empty string of two spaces created by
 concatenation operators .

All variations of
                <template>
                can be used with this option.

If
                multiple templates
                are used, a new copy of the original source string
will be supplied to each of the templates. <expression> will not be
re-evaluated, however, so variable assignments made in a previous template
will not affect the value of the source string even if one of the newly
assigned variables is used within <expression>.

                Technique note: Copy data from source code

1.12   ARexxGuide | Instruction Ref. | parse (8 of 8) | VAR

                PARSE
                [UPPER] VAR <variable> <template>

   The VALUE and VAR options allow internal program values to be used with
   the PARSE instruction.

The source string for the <template> is the value of <variable>. If more
than one variable is used in an <expression> , then PARSE VALUE <expr>
WITH <template> must be used.

All variations of
                <template>
                can be used with this option.

If
                multiple templates
                are used, a new copy of the source string will be
supplied to each of the templates.

                Technique note: Using the clip list

1.13   ARexxGuide | Instruction Ref. | parse (1 of 1) | TEMPLATE

                PARSE
                [UPPER] <source> <template> [, <template>]
                                       The general form of <template> is:
   [<marker>] <target> [<marker>] [<target>] [...]

At least one <target> variable must be included. A substring derived from
<source> is assigned to <target>. The optional <marker> may be a string

value specifying a pattern to be matched in <source> or a number
indicating a position within the source string where the division into
substrings should occur.

The following nodes describe the many <template> options:

                    Assigning words to variables
                          Tokenization
          placeholder { . } symbols

                    Matching patterns in the source string
                          Pattern markers

                    Splitting strings at defined positions
                          Positional markers

                    Using variables as template markers

                    Combining different types of markers

                    Using multiple templates
                   It is through its template that the PARSE instruction gains its ←
                      great
versatility. The template allows this one instruction to do the work of
several assignment clauses and far more. Options to the template allow
it to divide a string into its component words as the WORD() function
can do; to perform pattern matching like that done by POS() and its
cousin functions; and to split strings at defined locations like several
iterations of the SUBSTR() function.

Next, Prev & Contents: PARSE

1.14   ... parse | Template (1 of 7) | TOKENIZATION

                      TOKENIZATION: Assigning words to variables
     ~~~~~~
In its most basic form, <template> comprises a single variable called a
target to which the entire value derived from the input source is assigned.

    Example:
       /**/
       PARSE VERSION VerString
       SAY VerString   >>> ARexx V1.15 68030 NONE NTSC 60HZ
                                         /* for example. varies with machines */

In this example, the PARSE instruction acts in much the same way as a
simple assignment clause . Assuming there were a function that would
supply the same version information (which there isn't in the standard set
of functions), then the same information could be retrieved by using one
assignment clause: 'VerString = version()'.

The PARSE instruction is not, however, limited to just one assignment.
Multiple variables can be used in <template> causing the source value to

be split into its component words (with 'word' defined as any set of
characters separated by at least one blank from other 'words'). Each of
those words, in order, will then be assigned to the variables in the
<template>. The last variable in the template acts a bit differently than
the others, however: Any words that have not been assigned, along with any
leading or trailing blank characters will be assigned to the final
variable.

What happens is really much simpler than the explanation:

```
    Example:
      /**/
      PARSE VERSION Prog VNum Proc MathCoP Video
      SAY Prog                                    >>> ARexx
      SAY VNum                                    >>> V1.15
      SAY Proc                                    >>> 68030
      SAY MathCoP                                 >>> NONE
      SAY Video                                   >>>  NTSC 60HZ
```

Notice that two words are assigned to the final variable [Video] and that
the string has a leading space since it has picked up all of the <source>
string that was not already assigned.

In this format, the PARSE instruction acts much like multiple iterations
of the WORD() function, but is often quicker because of the way
instructions are implemented.

THE PLACEHOLDER TOKEN   Some words in the <source> string may be unneeded
                        by the program, but must still be assigned before
getting to the position of significant words. For those situations, the
PARSE instruction recognizes a special placeholder token -- the period
'.' -- which acts in all ways like a target variable except that no
actual assignment occurs. When used in this way, the placeholder token
must be separated by at least one blank from other tokens .

In the following example, placeholders are used to throw away all but two
words from the
                VERSION
                string:

```
      /**/
      PARSE VERSION . . Proc MathCoP .
      SAY Proc                                    >>> 68030
      SAY MathCoP                                 >>> NONE
```

The value assigned to [MathCoP] does not include a leading space since it
is not the last 'variable' in the template; a placeholder token is, and
that placeholder grabs (and tosses out) the remainder of the string along
with the leading blank.

In many circumstances an extra trailing placeholder token is used to
assure that the final variable will be stripped of the leading space. In
the next example, a placeholder is used for each word except the last two:

```
      /**/
      PARSE VERSION . . . . . VType VFreq
      SAY VType                                   >>> NTSC
```

```
    SAY VFreq                                    >>>  60HZ
```

Because it is the last target variable in the template, [VFREQ] is
assigned a leading space, but when one more placeholder is added at the
end of the template, that space disappears:

```
    /**/
    PARSE VERSION . . . . VTYPE VFreq .
    SAY VType                                    >>> NTSC
    SAY VFreq                                    >>> 60HZ
```

```
  Also see  WORD()
            WORDS()
```

1.15   ... parse | Template (2 of 7) | PATTERN MARKERS

```
    Matching patterns in the source string
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
A pattern marker is a string token of any length that is used as a
pattern to be matched in the <source> string. If the pattern is found, the
source string will be split at the matched pattern. The substrings on
either side of the pattern will be assigned to the target variables on
either side of the marker. The pattern itself will be removed from the
string before the assignment.

```
    Example:
    /**/
    FName = 'sys:rexxc/tco'
    PARSE VAR FName Dev ':' File
    SAY Dev                                      >>> sys
    SAY File                                     >>> rexxc/tco
```

Multiple pattern markers can be used in the same template:

```
    /**/
    FName = 'sys:rexxc/tco'
    PARSE VAR FName Dev ':' Path '/' File
    SAY Dev                                      >>> sys
    SAY Path                                     >>> rexxc
    SAY File                                     >>> tco
```

If a pattern cannot be located in the <source> string, all of the string
that has not already been assigned will be assigned to the target
variable to the left of the unmatched pattern marker:

```
    /**/
    FName = 'sys:rexxc/tco'
    PARSE VAR FName Dev 'REXXC' File
    SAY Dev                                      >>> sys:rexxc/tco
    SAY File                                     >>>
```

Pattern matching is case-sensitive, just as it is with POS() and similar

functions. Because the upper-case 'REXXC' was not found in the source
string, the entire string was assigned to the previous variable, [Device].

The UPPER option is helpful when using alphabetic pattern markers because
guarantees that the source string will match the case of patterns entered
in uppercase:

```
    /**/
    FName = 'sys:rexxc/tco'
    PARSE UPPER VAR FName Dev 'REXXC' File
    SAY Dev                                     >>> SYS:
    SAY File                                    >>> /TCO
```

```
  Also see  FIND()
            POS()
            WORD()
```

1.16   ... parse | Template (3 of 7) | POSITIONAL MARKERS

                        Splitting strings at defined positions
    ~~~~~~
Positional markers are numbers that indicate the character position at
which the source string should be divided. There are two forms of
positional markers: absolute and relative. Both forms must be whole
numbers.

ABSOLUTE POSITIONAL MARKERS   are entered without a prefix '+' or '-' sign
                              and indicate the starting column position
within <source> of the substring that will be assigned to the following
variable.

```
    Example:
      /**/
      Prod = 'Widget   red and white  99.95'
      PARSE VAR  Prod 1 Name 10 Des 25 Price
      SAY Name                                  >>> Widget
      SAY Des                                   >>> red and white
      SAY Price                                 >>> 99.95
```

The first target variable, [Prod], was assigned all characters in the
source string from column 1 through column 9. [Des] was assigned the
characters between and including columns 10 through 24, and the final
target variable, [Price], received all characters from column 25 through
the end of the string. (The values of the first two variable include
trailing spaces.)

In the example, the '1' before [Name] could have been excluded since the
first column is the default starting position. On the other hand, the
starting number can be higher than the first column, which would cause the
initial characters in the source string to be ignored.

RELATIVE POSITIONAL MARKERS   are whole numbers entered with a prefix of
                              either '+' or '-' and indicate the position

within <source> of the next substring relative to the start of the
previously assigned substring. The movement can be forward (indicated by a
'+' sign) or backward (indicated by a '-' sign).

A fragment like {PARSE VAR Line 4 Sub1 +3 Sub2} would instruct that the
first variable be assigned characters beginning in column 4 (using an
absolute marker, in this case, although a relative marker of '+4' in that
position would have exactly the same effect) and that the second variable
be assigned all remaining characters starting at the position 3 characters
to the right of that. (The '+3' can also be interpreted as referring to
the length of the substring to be assigned to the variable to the left of
it).

Using relative markers, the previous example could be entered as:

```
/**/
Prod = 'Widget   red and white  99.95'
PARSE VAR  Prod Name +9 Des +15 Price
SAY Name                                     >>> Widget
SAY Des                                      >>> red and white
SAY Price                                    >>> 99.95
```

When the marker is a negative number, the parse will move left in the
string (towards column 1):

```
/**/
Src='1234567890'
PARSE VAR Src 4 strA -1 strB -2 strC
SAY strA                              >>> 4567890
SAY strB                              >>> 34567890
SAY strC                              >>> 1234567890
```

Each of the substrings in this example include all characters from the
starting position up to the end of the source string. That is due to
another characteristic of template scanning:

   Whenever a marker causes the scanning position to remain in the same
   place or to move to a lower position, the variable prior to that
   marker will receive the remaining characters up to the end of the
   source string.

When a positional marker causes the scan to back up in a string, the value
of the string will have changed if a pattern marker was used previously.
(See notes on
               PARSE compatibility
               with the standard REXX instruction.)
In the following, a pattern 'V' is used to match the character that
identifies version number in the PARSE VERSION string. That character is
removed from the working copy of the string, so when the scan is backed up
with an absolute positional marker of 0, it is not included in value of
[VerStr]:

```
/**/
PARSE VERSION 'V' Ver . 0 VerStr
SAY 'Version' Ver':' VerStr
```

The output of this might be

```
        Version 1.15: ARexx 1.15 68030 NONE NTSC 60HZ


    Also see  SUBSTR()

                    Technique note:  AddComma() user function
```

1.17   ... parse | Template (4 of 7) | MARKER VARIABLES

```
                    Using variables as template markers
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
A variable symbol used in <template> is normally considered to be a
target for a new assignment. Any value that might have been assigned to
the variable prior to its use in the PARSE template will be lost.

Clearly, then, a variable cannot be used as a marker in the template
without some kind of special indication that it should be evaluated. The
PARSE instruction recognizes these special characters for that purpose:

```
    ()      Parentheses around a variable symbol indicate that the value of
            the enclosed variable should be used as a
                    pattern marker
                .

    =       An '=' sign to the left of a variable symbol indicates that its
            value should be used as an  absolute marker .

 + or –  A signed prefix to a variable indicates that its value should be
            used as a  relative marker .

    Example:
      /**/
      FName = 'work:words/notes/ARx_Changes.doc'
      DevDiv = POS(':', FName) +1
      FNPOS = LASTPOS('/', FName) +1
      ExtPos = LASTPOS('.', FName) +1
      if ExtPos < FNPOS then ExtPos = length(FName)
      PARSE VAR FName Dev =DevDiv Path =FNPOS File 90 =ExtPos Ext
      SAY 'Device: "'Dev'", Path: "'Path'"'
      SAY 'File: "'File'", Extension: "'Ext'"'

        >>> Device: "work:", Path: "words/notes/"
        >>> File: "ARx_Changes.doc", Extension: "doc"
```

The value of the marker variable can be set during the same instruction in
which it is used by including its symbol first as a target variable before
it is used as a marker.

In the following example, the first three tokens are numbers that are used
as marker variables later in the template. Numbers like this could be used
to read variable-length fields in a file of one-line records.

```
    Example:
       /**/
       Rec = '4 12 4, The quick brown fox jumped over ...'
       PARSE VAR Rec Len1 Len2 Len3 ', ' Fld1 +Len1,
          Fld2 +Len2 Fld3 +Len3 Fld4
       SAY Fld1                                      >>> The
       SAY Fld2                                      >>> quick brown
       SAY Fld3                                      >>> fox
       SAY Fld4                                      >>> jumped over ...
```

If the datatype of a relative or absolute marker variable is not numeric,
ARexx will generate this error:

    +++ Error 47 in line <#>: Arithmetic conversion error

Next: COMBINED MARKER | Prev: POSITION MARKERS | Contents: TEMPLATES

1.18   ... parse | Template (5 of 7) | COMBINED MARKERS

                         Combining different types of markers
      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Each of the options available in creating templates can be combined with
any of the other options to create powerful text processing routines.

The following program fragment shows how information can be pulled from
tightly packed lines of data. Although just one variable is used here, the
same PARSE instruction could be used within a loop to read a file of many
similarly formatted records forming the basis of a scheduling utility.

```
   /**/
   Apt = '10-Aug-9315:00Cut, Perm, Color|Winnie Foo|Bruce 3'
   PARSE VAR Apt Dt 10 Tm +5 Proc '|' Cust '|' Sty Chr . ,
      /* continuation of above */    1 (Proc) Sch.Sty.Tm (Sty)
```

The output from a trace of the PARSE instruction is shown below at the
right with the name of the target variable on the left.

```
  Dt                >>> "10-Aug-93"
  Tm                >>> "15:00"
  Proc              >>> "Cut, Perm, Color"
  Cust              >>> "Winnie Foo"
  Sty               >>> "Bruce"
  Chr               >>> "3"
  .                 >.> ""
  SCH.Bruce.15:00   >>> "Winnie Foo"
```

The process used to arrive at these assignments is examined below by
showing the position within the source string of the anchor position and
end position within the scan.

    ^ ANCHOR POSITION is the starting point in the source string from
    A which ARexx will scan for its next marker. The character at the
       anchor position is included in the assigned substring.

```
  ^   END POSITION is the point to which the scan will move before the
  E   substring is assigned to a target. The character at the end position
      is not included in the assigned substring.
```

Unless an intervening marker changes its position, the anchor position
(the starting point) for one target variable is the end position for the
previous target.

ASSIGNING VALUE TO Dt USING AN ABSOLUTE MARKER :

```
   Source: 10-Aug-9315:00Cut, Perm, Color|Winnie Foo|Bruce 3
           ^         ^
           A         E
```

Because the target variable [Dt] is the first item in the template, the
default anchor position is used -- the first character in the string. The
item following the target is '10', an absolute marker. The end position
is therefore moved to the tenth character in the string, causing the first
9 characters to be assigned to [Dt].

ASSIGNING VALUE TO Tm USING A RELATIVE MARKER :

```
   Source: 10-Aug-9315:00Cut, Perm, Color|Winnie Foo|Bruce 3
                    ^    ^
                    A    E
```

The next item in the template is another target, so the anchor position
moves to previous end position. The following item, '+ 5' is a relative
marker that causes the end position to be set five characters beyond the
anchor.

ASSIGNING VALUE TO Proc USING A
               PATTERN MARKER
             :

```
   Source: 10-Aug-9315:00Cut, Perm, ColorWinnie Foo|Bruce 3
                              ^               ^
                              A               E
```

Again, the next item is a target, so the current anchor position is the
old end position. The next item '|' is a pattern marker. That pattern is
found between 'r' and 'W' and is then removed from the working copy of the
source string (the original value of the variable [Apt] isn't altered).

ASSIGNING VALUE TO Cust USING A
               PATTERN MARKER
             :

```
   Source: 10-Aug-9315:00Cut, Perm, ColorWinnie FooBruce 3
                                       ^         ^
                                       A         E
```

The process used with [Proc] is used again to find the substring for
[Cust]. Again, the matching pattern is removed from the working copy of
the string.

```
ASSIGNING VALUE TO Sty AND Chr USING
                 TOKENIZATION
               :

    Source: 10-Aug-9315:00Cut, Perm, ColorWinnie FooBruce 3
                                                ^      ^^
                                                A      E
```

The next three items in the template 'Sty Chr .' are all targets. When
one target is followed by another, the next available blank-delimited word
in the source string is assigned to the first target of the pair. That
assigns 'Bruce' to [Sty] and '3' to [Chr]. The placeholder '.' target
is used to remove the leading space from the '3'.

BACKING UP IN THE SOURCE STRING

The next item in the template is another positional marker that moves the
scan to the left, back to column 1. If we had not already reached the end
of the source string with the previous scan, all that remained of the
string would have been assigned to the previous target. In this case,
however, the marker simply repositions the anchor:

```
Source: 10-Aug-9315:00Cut, Perm, ColorWinnie FooBruce 3
        ^
        A
```

With the anchor at that position, a new pattern marker is introduced. This
one uses a variable for the pattern -- a variable that was assigned during
the previous scan. The value of [Proc] is 'Cut, Perm, Color'. That value
is matched in the source string and then removed:

```
Source: 10-Aug-9315:00Winnie FooBruce 3
                      ^
                      A
```

The target variable [Sch.Sty.Tm] is a compound variable that uses as
branches two of the variables defined in the just-completed scan. The
 derived name of this target is 'SCH.Bruce.15:00'.

The end position is determined by another pattern-marker variable,
'Bruce':

```
Source: 10-Aug-9315:00Winnie Foo 3
                      ^          ^
                      A          E
```

See the next node for a method of re-using pattern markers in a template.

Next: MULTIPLE TEMPLATES | Prev: MARKER VARIABLES | Contents: TEMPLATES

1.19   ... parse | Template (7 of 7) | MULTIPLE TEMPLATES

```
                  Using multiple templates
      ~~~~~~~~~~~~~~~~~~~~~~~~~
```
                            :

When a comma ',' is entered as a literal value (without quotation marks)
in a template, the markers and targets that follow it are considered part
of a new template. Multiple templates are handled differently by different
source options. The ARG, PULL, and EXTERNAL options will retrieve a new
string from the input stream. The other source options will make a new
copy of the original string for the new template.

Multiple templates are often used with the ARG option to retrieve each of
the arguments to an internal function. Unlike AmigaDOS, which passes
command-line options as a single argument, ARexx can pass multiple
arguments (up to 15) to an internal function, just as it does to
 built-in functions. Each argument is separated by a comma in the calling
syntax and can be retrieved by successive templates in the PARSE ARG
instruction.

```
    Example:
        /**/
        say MultArg('3 arguments', 'mixed up', 'Multiple')
        exit

        MultArg: procedure
        parse arg Num Arg1 ., Arg2, Arg3
        return Arg3 '('Num')' Arg2 Arg1'.'
```

The output will be:

        Multiple (3) mixed up arguments.

The first template 'Num Arg1 .' uses
                tokenization
                to retrieve the two
'words' used in the first argument string. The second and third templates
use a single target variable to retrieve the entire string used as the
second and third argument strings.

Multiple templates can be used instead of a
                positional marker
                to back up
in the string. That's especially useful when a
                pattern
                marker is re-used
in a template. Using a second template, the code from the previous node
could be rewritten to reuse the same markers:

```
 /**/
 Apt = '10-Aug-9315:00Cut, Perm, Color|Winnie Foo|Bruce 3'
 PARSE VAR Apt Dt 10 Tm +5 Proc '|' Cust '|' Sty Chr .,'|' Sch.Sty.Tm '|'
```

Even though the '|' characters were removed from the copy of the source
string used for the first template, they can be used again in the second
template since a new copy of the string is made.

Next: Template | Prev: OVERVIEW | Contents: TEMPLATE

1.20   ARexxGuide | Instructions | PARSE (Note) | Standards

```
                    ARexx departures from REXX-standard PARSE
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

According to the REXX standard, all
                template variables
                should be
surrounded by parentheses. The ARexx method will not work on other systems.

When it is used with simple tokenization_markers, the PARSE instruction in
ARexx works in the standard way. When more complex templates are used,
however, ARexx produces results that are not standard.

Here are examples of a PARSE from ARexx compared to standard results:

```
   a=" alpha, beta, gamma, "      /*    ARexx        Standard       */
                                  /* x= ~~~~~~~~~~    ~~~~~~~~~~~~~~~~~*/
   parse var a 'ha,' x +2         /*    " b"       - "ha"            */
   parse var a 'ha,' +2 x .       /*    "eta,"     - ","             */
   parse var a . . x              /*    " gamma, " - "gamma, "       */
   parse var a ',' . 1 ',' x      /*    " gamma, " - " beta, gamma, "*/
```

Notes about the differences:

```
   > parse var a 'ha,' x +2         /* x=" b"       - "ha"           */
   > parse var a 'ha,' +2 x .       /* x="eta,"     - ","            */
```

ARexx apparently misinterpreted the "last position". Cowlishaw says, "The
'last position' set by a literal pattern is the position at which the match
occurred...," but ARexx uses the end point of the previous match as its
last position.

```
   > parse var a . . x              /* x=" gamma, " - "gamma, "      */
```

In TRL1 , Cowlishaw writes, "...only the blank that delimits the previous
word is removed from the data." ARexx, however, does not remove the blank.

```
   > parse var a ',' . 1 ',' x  /* x=" gamma, " - " beta, gamma, "   */
```

Here, ARexx removes the matched pattern from _both_ the assigned string and
the source string.

The ARexx method is non-standard, but it's a useful non-standard feature.
The advantage of the ARexx method is that it produces helpful side-effects
without losing the ability to produce the same output. In ARexx, a PARSE
instruction can be used to remove unwanted characters from a string:

```
   /**/
   TestStr = 'Find and remove #special field@ markers'
   parse var TestStr '#' SpecFld '@' 1 FixedStr
   say SpecFld':' FixedStr

   >>> special field: Find and remove special field markers
```

If one wishes to keep the matched pattern, that can be done with multiple

templates:

```
/**/
TestStr = 'Find and remove #special field@ markers'
parse var TestStr '#' SpecFld '@', FixedStr
say SpecFld':' FixedStr

>>> special field: Find and remove #special field@ markers
```

The ARexx method also makes this easier than the standard:

```
/**/
TestStr = 'Get marked# words in# a# string'
do while pos('#', TestStr) > 0
   StPos = lastpos(' ', ' 'TestStr, pos('#', TestStr))
   parse var TestStr =StPos SWord '#' 1 TestStr
   say SWord
end
say TestStr

>>>          marked
             in
             a
             Get marked words in a string
```

One of the source options available in standard REXX is not implemented in
ARexx: In the standard definition, 'PARSE LINEIN' reads a single line of
input from the standard input device. As with all other file_I/O, ARexx
uses non-standard methods and does not implement this instruction. It can,
however, be duplicated with this instruction:

```
PARSE value readln(STDIN) with <...>
```

The STDIN file is always available in a script (although it might be
redirected), so it need not be explicitly opened with the OPEN()
function.

```
                PARSE EXTERNAL
                in ARexx is not implemented in the standard.
```

   (These notes use examples provided in a letter to the author by Ian
   Collier, the developer of REXX/imc, one of the Unix REXX interpreters.)

Next, Prev, and Contents: PARSE