ARx_Elements.ag

COLLABORATORS			
	TITLE : ARx_Elements.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 3, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARx_Elements.ag

1.1 ARexxGuide | Basic Elements: The chemistry of ARexx

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


              About this section

              Structure of an ARexx program

              Tokens

              Comment tokens

              String tokens

              Symbol tokens

              Operator tokens

              Reserved characters
                  Expressions
      Numbers                    Strings                     Variables
      Function call         Operations

    Clauses
       Assignment clauses       Instructions          Command clauses
       Label clauses            Null clauses
```

1.2 ARexxGuide | Basic Elements: The chemistry of ARexx | ABOUT

```
The chemistry of ARexx
```

~~~~~~~~~~~~~~~~~~~~~~~
Like a child's virtual chemistry set, ARexx is offers vials of intriguing
substances that hold a promise of wonderful creations if mixed together.

It's possible to mix with abandon and see what happens -- possible, but
dangerous. Even a careless experimenter with ARexx will rarely produce a
system crash -- the Amiga equivalent of a chemical explosion, but it is
easy enough to produce the smoke of endless loops and syntax errors.

Knowing what's in the vials of the chemistry set will make the experiments
both safer and more productive. Error messages like

        +++ Error 41 in line 1: Invalid expression
    or
        +++ Command returned 20

will be a less frequent occurrence if the programmer understands what an
expression is and how to use it; what a command is and why it might
"return 20".

This chapter explains the structure of the ARexx language to help
programmers understand and perhaps avoid error messages like the ones
above.

Next, Prev, & Contents: Basic Elements

1.3   ARexxGuide | Basic Elements (1 of 5) | STRUCTURE

                        Structure of an ARexx script
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
An ARexx script or program is composed of words and symbols that have
special meaning to another program that runs on the Amiga. That second
program -- RexxMast is its name -- is called an 'interpreter'. It reads
each line of the script (sometimes called 'source code') and converts the
words and symbols into the kind of language that the computer's operating
system can understand and act on.

As the interpreter reads the source code, each line is significant. It is
treated as a logical whole. That's different from the way sentences are
constructed in English prose: in an article like this one, a line is not
significant. No matter how it happens to be divided into lines on the
screen, a sentence continues until it is closed with a period. Some
programming languages use the same kind of structure, ignoring line-breaks
in favor of a special sentence-ending character (often a semicolon).
ARexx, on the other hand, treats each line as a unit unless it ends with a
special character -- the
                    comma
                    -- to indicate that it should be treated
otherwise.

A line is called a clause rather than a 'sentence', but -- like a
sentence in prose -- it encapsulates a complete unit, an action that is to
be performed. There are several broad categories of actions that can be
performed by a clause.

Each line must be further broken down by the interpreter before its
purpose can be identified. The clause is divided into elements called

                    tokens
                   . A token might be a word like 'select' or a symbol like '+'. It
is the smallest element that the interpreter will be concerned with.

Some tokens represent another value. Before acting on the clause, the
interpreter looks for a token or a collection of tokens that can be
converted into a new value. Those representational tokens (sometimes a
single token like a string , a number , or a variable ; and sometimes
one or more tokens tied together by operator tokens) are called
 expressions . The final clause is a line made up of the derived value of
each expression in the original.

1.4   ARexxGuide | Basic Elements (2 of 5) | TOKENS

                   The smallest element: Tokens
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
When the ARexx interpreter scans a line of code in a script, it divides
the line into small elements called tokens. A token is the atom of a
script. It is the simplest item in the language, the basic stuff from
which other elements of the language are made. A token might be a single
character like '+' or '/' or a word like 'FOO' or 'CALL'. ARexx recognizes
a valid token by characteristics explained in this section.

ARexx recognizes five types of tokens:

                    Comments

                    Strings

                    Symbols

                    Operators

                    Special Characters
                   Single-character tokens like '+' and '/' need not be separated ←
                       from others
by spaces because the interpreter recognizes them as tokens even when they
are abutted with other characters as in 'Answer=34+42;'.

When ARexx encounters an element in a line that does not match any of its
rules for forming tokens, it will generate the error message

    +++ Error 8 in line < # >: Unrecognized token

This error is often caused by the use of invalid characters in a symbol
name.

Next: EXPRESSIONS | Prev: Elements contents | Contents: Elements contents

1.5   ARexxGuide | Basic Elements | Tokens (1 of 5) | COMMENTS

                        Comment tokens
~~~~~~~~~~~~~~~
The '/*' characters indicate the start of a comment in ARexx. The
 interpreter will ignore anything that comes between those characters and
the characters '*/' that indicate the end of a comment.

The whole comment is treated as a single token: ARexx considers everything
between and including the '/*' and '*/' markers to be one program
element, an element that can be ignored.

Comments can (and should) appear anywhere in a program, and can be nested
-- that is, multiple '/* */' pairs can be used in a comment.

To distinguish between a file meant to be an ARexx script and other files,
ARexx expects to find '/*' at the beginning of each script along with the
closing '*/', which does not need to be on the same line. With one

                  minor exception
                  , ARexx will not run a script that doesn't begin with a
comment token. Instead, an error message similar to this will be returned:

    Command returned 5/1: Program not found

    Also see: Null clause

Next: STRINGS | Prev: Tokens | Contents: Tokens

1.6   ARexxGuide | Basic Elements | Tokens (2 of 5) | STRINGS

                  Literal strings (String tokens)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Just as paired markers indicate the beginning and end of a
                  comment
                  , a
string token (or string literal) is indicated by paired quotation marks
enclosing any collection of characters. The characters in a string are one
token and will not be evaluated by the interpreter .

Either single-quotation marks { ' } or double marks { " } may be used to
mark a string token, but the opening mark must be paired with another of
the same type. (Single marks are preferred in the REXX standard.)

Anything that can be typed on a keyboard (and more) can be included
between the quotation marks. ARexx will not alter the value except to
remove the opening and closing quotation marks. A string token can
comprise up to 65535 characters.

```
    Examples:
        'I get used to the muck as I go along'    /* a literal string */
        ''                                        /* null string. Literally */
```

Characters that cannot be typed on the keyboard can also be included in a
string. The language uses a special kind of string token to represent such
characters:

                    Hex and binary strings
                    entering unprintable characters.

    Using quotation marks within a string
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Hex strings ('27'x and '22'x) can be used to identify quotation marks used
as literal values within a string, but ARexx recognizes other methods:

    Use the alternate type of quotation mark to enclose the the string:
        say 'He said, "No way."'   >>> He said, "No way."

    Use doubled marks to include a single mark within a string
        say 'I can''t go on...'   >>> I can't go on...

The double-quote marks in the first string will be printed as they are
entered since the string is identified to ARexx by single-quote marks. In
the second example, one of the two marks in 'can''t' will be removed
when the string is processed.

    Also see: String expressions

 Compatibility issues :
    A string in ARexx may be defined over several lines. The REXX standard
    specifies that a string literal must be defined on a single line.
    Multiple line strings are contrary both to the definition and to the
    overall spirit of the REXX language.

    Because it allows multiple-line string definitions, ARexx is often
    unable to identify missing quotation marks. ARexx does not treat the
    resulting string as expected. It will, for instance, remove the
    line-end characters in the string, but will include all of the white
    space -- tabs and spaces -- between the opening and the closing
    quotation mark.

    It is a bad idea to use this multi-line definition 'feature' of ARexx.

    Also see: Error 5
               Error 9
               Error 26
```

## 1.7  ARexxGuide | Basic Elements | Tokens | STRINGS (1 of 1) | HEX/BINARY

                        Hex and binary strings
~~~~~~~~~~~~~~~~~~~~~~~~
Strings are the universal lingo of ARexx. Where other languages would use
a number to represent data, ARexx often uses a character string -- even
for the computer's memory addresses. (See the functions  GETSPACE()  and
 OPENPORT()  for examples.)  These are not, however, strings that can be
easily entered with the number and alphabetic keys on a keyboard.

For these instances, ARexx recognizes two special types of strings --
strings in which the characters are represented by hexadecimal or binary
numbers.

A hexadecimal string is indicated by an 'x' or 'X' character
immediately following the closing quotation mark of a string of valid
digits and letters (0-9, a-f, A-F).

A binary string is indicated by 'b' or 'B' immediately following the
closing quotation mark of a string of 0's and 1's.

    Examples:
        '416D696761'x                         /* hexadecimal string     */
        '0a'X                                 /* hex string = return    */
        '09'x                                 /* hex string = tab       */
        '00011101'b                           /* a binary string        */

The hex or binary number may be divided by spaces to aid readability, but
only at the 'byte boundaries' of the digits/characters.

Each byte in the string (2 digits in hex, 8 in binary) represents one
character. The letter 'O', for instance, has an ASCII value of 4F in
hexadecimal notation; the character 'm' has a value of 6D. It is,
therefore, possible to represent the string 'Ommm' as '4F 6D 6D 6D'x

The hex or binary numbers must be literal strings.  Variables  and other
 expressions  cannot be substituted. The following, for instance, is an
invalid construction:

    /* Invalid as a hex string */
    HNum = '4152657878'
    say HNum||x                       >>> 4152657878X

    /* Valid alternatives     */
    HNum = '4152657878'
    say x2c(HNum)                     >>> ARexx
    say '4152657878'x                 >>> ARexx

Although they can be used as
                variable symbols
                , the characters 'X' and
'B' act as reserved tokens when abutted against a literal string:

    /**/
       x = 'voidable'
       say 'una'||x        >>> unavoidable
       say 'ae'x           >>> ®
       say 'una'x          >>> +++  Error 8  in line 4: Unrecognized token

It is normally valid to use the  abuttal operator  to join a string and a
variable. In this case, however, the variable [x] is not recognized as a
variable, but rather, as the hexadecimal indicator. The second SAY
instruction output the character represented by the hex number 'AE'. The
third instruction generates an error because 'u' and 'n' are invalid
characters in a hex string.

ARexx includes a number of  translation functions , such as  X2C()  above,
that will translate data from one format to another.

A hex or binary string can be used to identify characters not otherwise
accessible within a program. The line-feed character, for instance, which
is used to mark the end of a line on the Amiga, is ASCII character 10 in
decimal notation or '0A' in hexadecimal notation. That character can be
included in a string as a hex-string:

    SAY 'This will print an extra blank line.'||'0a'x

The effect of using the hex string is the same as this more complex
fragment:

    SAY 'This will print an extra blank line.' || x2c(0a)

Next, Prev, & Contents: STRING

## 1.8   ARexxGuide | Basic Elements | Tokens (3 of 5) | SYMBOLS

                    Symbol tokens: The proper names in ARexx
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Symbols are the most versatile of the various kinds of tokens. The tokens
called symbols in ARexx would be called 'words' and 'numbers' in English.

A symbol is a grouping of any of the following alphabetic characters,
digits, or special characters:

            A to Z  a to z  0 to 9  . _ ! $ @ #

To make it more readable, a symbol can be entered in a mixture of upper-
and lowercase alphabetic characters. ARexx will, however, translate all
such characters to uppercase before the symbol is processed.

There are two types of symbols:


            Fixed symbols
              are usually numbers, although they can be any
                 symbol that begins with a digit or a period '.'.


            Variable symbols
              begin a letter or one of the characters ! $ _ @ #
                 but may include embedded digits or periods.

This node, and the ones referenced here, explain the rules that govern the

making of symbols, but do not explain how they are actually used. For
that, see the sections listed below.

            Interactive example:  Test for valid symbols              *

   Also see:  Expressions
              Expressions: Variables
              Expressions: Numbers

 Compatibility issues:
   ARexx is more generous than the REXX standard in the characters
   recognized as valid in symbols. The characters '$', '@', and '#' are
   not accepted in the current standard. As Cowlishaw points out in
    TRL2  (page 21), "Use of these characters is best avoided ... as
   programs written using them are often not portable between different
   countries and between different computer systems."

## 1.9   ARexxGuide | Basic Elements | Tokens | Symbols (1 of 2) | CONSTANTS

Constant symbols
~~~~~~~~~~~~~~~~~
Fixed symbols, which are also called 'constants', are formed according to
these rules:

   -- begin with a digit {0-9} or a period {.}, but may contain other
      characters after the first.
   -- cannot be assigned a new value.

Most constants are  numbers , but a symbol that starts with a numeral (or
a period) and includes alphabetic characters other than 'e' -- the mark of
an  exponent  -- may be used for special purposes. Such symbols are not
numbers, but neither can they be subject to variable  assignment , which
makes them useful as  file handles  and in  compound variables  as
 branch names .  (SIGNAL ON NOVALUE  will, however, trap symbols of this
kind as errors.)

        Examples:
            9
            2.987
            0.653E4     /* scientific notation */
            6foo        /* might be used as the logical name of a file. */

 Compatibility issues:
   To allow for future extensions to the language, the ANSI committee that
   is working on a standardized definition of REXX has recommended against
   use of '.' as the first character in a non-numeric constant symbol.
   Although ARexx may not be changed to meet the ANSI definitions, users
   might want to avoid use of such symbols to maintain greater
   compatibility with other versions of REXX.

## 1.10   ARexxGuide | Basic Elements | Tokens | Symbols (2 of 2) | VARIABLES

```
Variable symbols
~~~~~~~~~~~~~~~~
Variable symbols are formed according to these rules:

   -- begin with an alphabetic character or one of the special characters
   -- may be assigned a value
```

If a variable symbol has not been assigned a value, it is called
'uninitialized', but can still be used in most expressions because its
default value is the symbol's name translated to upper case. *

Simple symbols are used not only for variables, but also to name
 functions or subroutines within a program, and as keywords to mark
 instructions .

There are three kinds of variable symbols that are explained more fully in
the nodes on variables and assignments . The following rules govern the
formation of the symbols:

```
   Simple Symbols
      -- do not include a period in their name.

   Stem Symbols
      -- name ends with a single period {.}.

          Stem Symbols form the base of compound symbols.

   Compound Symbols
      -- contain at least one period {.} inside the name
```

1.11   ARexxGuide | Basic Elements | Tokens (4 of 5) | OPERATORS

```
              Operator tokens
~~~~~~~~~~~~~~~~
Used singly or in combination, the following characters perform specified
operations:

              + - | / | & = ~ \ > < ^
```

The characters are called 'operators'.

A blank between strings also acts as an operator unless it is next to
another operator.

Operators, which are explained more fully in the section on expressions ,
can be divided into four basic groups:

```
        Concatenation  || <blank> <abuttal>
        Arithmetic     + - | / // %
        Comparative    < > = == >= <=
        Logical        & | && ~
```

The characters representing operators are reserved : They can be used
only for their defined purposes and are invalid in other contexts. This
sometimes presents a problem when commands are used without enclosing
quotation marks. If operator characters are included within the command,
ARexx will attempt to perform the indicated operation. The following, for
instance will generate an error:

```
  say exists(help:arx/arx_elements.ag)
                      >>> +++  Error 41  in line 1: Invalid expression
```

The error occurred because the AmigaDOS file name use colons (a
 reserved token in ARexx) and the
                 token
                 that represents division. It is
the colon in the filename that causes this error, but even without it, the
'/' would have generated a different error. To avoid these problems,
quotation marks should enclose strings that are not meant to be evaluated.

Next: SPECIAL CHARACTERS | Prev: Symbols | Contents: Tokens

1.12   ARexxGuide | Basic Elements | Tokens (5 of 5) | SPECIAL CHARACTERS

```
              Special-character tokens
~~~~~~~~~~~~~~~~~~~~~~~~~~
```
These characters have special meaning as ARexx tokens and can be used only
in the proper context:

```
          , ; : ) (
```

Except when they are used in literal strings or comments (which, since
they are tokens already, are not further broken down), any of those five
characters will act as a token delimiter, just as a space does. Blanks
adjoining any of these characters are removed when ARexx processes the
clause.


```
          ,  Comma continuation character

          ;  Clause end symbol

          :  Label identifier

          () Grouping / Function argument list
```
          Next: Tokens | Prev: Special Characters | Contents: Tokens

1.13   ... Tokens | Special characters (1 of 4) | COMMA

```
                    Commas -- grouping and continuation
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
Commas are most often used to divide the arguments in a function call.
If a null (empty) value is to be sent to a function, two commas act as
place-holders for the argument slot:

```
   Example:
      say show('P',,'0a'x)
```

This example will print out a list -- one per line -- of each message port
defined for the current system. Since the second argument to SHOW()
isn't needed in this context, the double-commas show that '0a'x (a
                hex
                string that represents a line-feed character) is to be used as the ←↩
                     third
argument.

Commas can also be used in a complimentary way by the PARSE instruction
to represent multiple templates . When template variables are separated
by commas, PARSE ARG will assign to each comma-separated template the
corresponding comma-separated arguments to an internal or external
function.

```
CONTINUATION CHARACTER  A less common usage of the comma can make some
~~~~~~~~~~~~~~~~~~~~~~~  programs more readable: When a line ends with a
```
comma that is not included within a string token, ARexx will combine that
line with the following line and treat both as one clause.

The continuation character allows long and complex clauses to be split
into multiple lines that can more easily be read on one screen or page.

```
   Example:
      Filename = substr(FilePath, 1 + max(lastpos(':',,
         FilePath),lastpos('/', FilePath)))
```

Because of the extra comma at the end of the first line, ARexx will treat
both of the lines as one clause .

```
                    Technique note:  Format a table of information
                                     Check unique datatypes
                                     Extract file name from full spec
```

Next: SEMICOLON | Prev: Operators | Contents: Operators

1.14   ... Tokens | Special characters (2 of 4) | SEMICOLON

```
                    Semicolon: Clause-end token
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
Each clause in ARexx ends with a semicolon, but it is rarely necessary
for the programmer to include the character since the interpreter will
add it automatically in these situations:

```
    -- at the end of a line of text unless the line is included as part of
       a
                   comment token
                   or ends with the comma
                   continuation character
                   ,
    -- after the keyword  THEN ,  WHEN , or  OTHERWISE  in an  IF  or
        SELECT  instruction,
    -- after the colon ':' in a  label  clause.
```

The automatically-added semicolons are called implied semicolons. They
make the programmer's task simpler since it is unnecessary to worry about
these end-of-line characters.

The following, for example, is a valid construction:

        IF a=b THEN c=a

It will, however, be processed as

        IF a=b THEN; c=a;

Implied semicolons make line-ends significant in ARexx: An instruction
can be split only at defined places. ARexx allows a variety of forms,
however, when entering control structures . The keyword THEN , for
example, may either be included within the same clause as IF or WHEN, or
be used as the initial keyword in a separate clause.

It is sometimes useful to include more than one short clause on a line.
That can be done by separating each clause with an explicit semicolon:

        a=b; c=d; e=f

                  More information:
                    One-line programs
                   Next: COLON | Prev: Comma | Contents: Special characters

1.15   ARexxGuide | Basic elements | Note: INLINE scripts

                  One-line scripts
~~~~~~~~~~~~~~~~~
There is an exception to the rule that all ARexx scripts must begin with a

                  comment
                  : If an entire program is contained on one line, it can be
executed directly by many environments and need not be marked by a
comment. Such a program is called an 'in-line script' and can be directly
executed by the RX command, or by entering the string with an opening
quotation mark in WShell . In-line scripts are used by some programs to
tie ARexx macros into particular keys.

If two or more distinct clauses are included on the same line, they must
-- in most cases -- be separated by a

```
                    semicolon
                  . Although ARexx supplies
the semicolons automatically at the end of each line in a conventional
program, it will not do that when clauses are entered on a single line.
```

The following program -- with the three lines be entered as one line of
text -- might be used as an alias in an AmigaDOS startup-file. (To use it
as an alias for the standard Amiga shell, an 'rx' would be added prior to
the opening quotation mark.)

```
   "say 'Press the desired key then press <Enter>'; options prompt '::: ';
      parse pull key; say 'The decimal value of that key is' c2d(key);
      say 'The hex value is' c2x(key)"
```

Next: Semicolon | Prev: Comment | Contents: Semicolon

1.16   ... Tokens | Special characters (3 of 4) | COLON

```
                Colons: Label markers
~~~~~~~~~~~~~~~~~~~~~~
A colon following a
                simple symbol
                creates a label that identifies the
beginning of a subroutine in ARexx.
```

 Labels and the subroutines the define are explained more fully in the
 section on expressions .

Next: PARENTHESES | Prev: Semicolon | Contents: Special characters

1.17   ... Tokens | Special characters (4 of 4) | PARENTHESES

```
                Parentheses: grouping
~~~~~~~~~~~~~~~~~~~~~~
Parentheses serve two primary purposes in ARexx: they may be used to group
expressions together, controlling the order of evaluation, or to enclose
the argument list to a  function  call.
```

Parentheses surrounding an expression force evaluation of the enclosed
expression before other operations are performed.

```
   More information:  Expression priority
```

When used in a function call, the parentheses enclosing the argument
list must immediately follow the
                symbol
                that names the function. There
cannot be a space between the symbol and the opening parenthesis.

```
   More information:  Function arguments
```

When used in a PARSE template , parentheses surrounding a variable symbol
indicate that the variable's assigned value is to be used, not as a target,
but as a pattern marker .

Although rarely used, parentheses can force the distinction between
 commands and instructions :

```
    (say "Hi there")        /* will cause some Amigas to speak */
    say 'Hi there'          /* will output text to the screen  */
```

Using parentheses in this way might be helpful in a few instances, but
quotation marks are still the preferable way to identify and isolate
commands since, unlike parentheses, they prevent ARexx from attempting to
interpret the command expression.