ARx_Elements2.ag

COLLABORATORS

	TITLE : ARx_Elements2.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 8, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARx_Elements2.ag

1.1   main

```
AN AMIGAGUIDE® TO ARexx                    Edition: 1.0
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, Robin Evans.  All rights reserved.
```

1.2   ARexxGuide | Basic Elements (2 of 4) | EXPRESSIONS

The term 'expression' is used more broadly in REXX than it is in some
other programming languages. An expression can be anything from a single
number to a complex mixture of numbers, variables, strings, function
calls, and sub-expressions.

An expression is one or more tokens that can be evaluated to produce a
single value. Any of the following elements of the language can serve as
an expression in ARexx:

```
    a number              Examples: 5  10  1245.8976
    a string              Examples: 'foo' 'The lazy fox.' '1020 Seneca'
    a variable            Examples: Foo Area.areas
    a function call       Examples: right(Foo, 3)  max(InpA, InpB)
    an operation          Examples: 5+10  'foo'>Area.areas 'foo'||'bar'
```

In some other languages, the term 'expression' is used only to refer to
the last of the five items listed above. Operations include arithmetic,
comparisons, and string concatenation. Any of the previous items in the
list can be tied together in an operation.

One type of expression is used so often in ARexx -- and in most other
computer languages -- that it deserves special mention:

```
                      Conditional expressions
```

Unless they are used as commands , expressions are not considered
complete program statements in ARexx. They are tied to other elements of
the language to form clauses that can be executed by the interpreter .

```
              NOTE: Avoiding accidental commands from expressions
```

Next: CLAUSES | Prev: Tokens | Contents: Basic elements

1.3   ARexxGuide | Basic Elements | Expressions (1 of 5) | NUMBERS

In ARexx, a number is a string of characters consisting only of digits, a
decimal point { . }, and -- optionally -- the letter 'E' followed by a
positive or negative number indicating exponential notation.

A number is a constant symbol , which means that it cannot be used as a
 variable .

Numbers, in other words, are exactly what we would expect them to be:
symbols like 3, or 4.987 or 5.123e-9.

When numbers are the terms of an operation , they again act as they would
be expected to act by a school child learning arithmetic: 3 + 5 equals 8.
3/5 equals .6. ARexx will even maintain the trailing 0's in most
operations so that 234.00 + 2 will result in 236.00. (The 0's disappear
only in a division operation.)

There is one point, however, where the discussion of numbers must become
more complex and make concessions for the machines on which calculations
are performed. That concerns numeric precision : How big can a number be
and still be workable within the program.

Next: NUMBER | Prev: Expressions | Contents: Expressions

1.4   ARexxGuide | Basic Elements | Expressions | Numbers (1 of 1) | PRECISION

The precision with which calculations and numeric comparisons are
performed in ARexx is arbitrary -- a default precision is set by the
language rather than by the system on which the program is running, but
can be changed by the programmer when a lesser or greater precision is
desired.

The default precision is nine digits, whether the digits are before or
after a decimal point. All of the following numbers are expressed in the
highest precision available by default in a script:

```
   56.9238007
   432987002
   .876309298
```

Because ARexx stores numbers as strings, a number larger than the default

precision can be assigned to a variable and used in an expression that
does not perform a numeric operation:

```
/**/
BigNum1 = 453.98762340706
BigNum2 = 578987234.9753
say BigNum1 'has' length(compress(BigNum1,'.')) 'digits.'
                                 >>> 453.98762340706 has 14 digits.
say BigNum1 + BigNum2            >>> 578987689
```

The LENGTH() and COMPRESS() functions were both able to handle the
large number since they can handle any string up to 65535 characters in
length. The issue of precision arises only in numeric operations.

The result of the addition operation, on the other hand, is not what would
be expected by a teacher asking a pupil to add the numbers. Since the
result could not be expressed within the default 9-digit precision, ARexx
rounded the result of the addition operation to a figure as close as a
9-digit number can be to expressing the correct result.

The integral value of the largest number (578987235) in the above
operation is still within the 9-digit default precision. A shift in the
position of the decimal point in line 3 of the code above presents ARexx
with the challenge of handling an integer larger than its default
precision. This is the new line:

```
BigNum2 = 5789872349.753
```

The result of the new version is { 5.7898728E+9 }. Even with rounding, the
number could not be expressed within nine digits, so ARexx reported the
result with exponential notation. The E+9 part of the number indicates
that the decimal point should be moved 9 places to the right, giving a
number of 5789872800.

Rounding of this sort can be helpful in some circumstances and problematic
in others. ARexx allows the programmer considerable control over the
precision of numeric operations. The NUMERIC instruction allows
precision to be set at any value between 1 and 14. A small setting for
NUMERIC should be avoided, however, except for specialized operations
since the setting will affect even the calculation of index variables in a
 DO loop.

The 14-digit maximum precision in ARexx is one point where this REXX
implementation imposes a restriction greater than the language standard.
Most implementations allow the programmer to set a maximum precision that
is virtually unlimited, although storage and speed restrictions of the
system on which the script is running may limit the useful range.

Next, Prev & Contents: NUMBER

1.5 ARexxGuide | Basic Elements | Expressions (2 of 5) | STRINGS

An expression in ARexx can be as simple as 'Hello world' -- something
called a string in program-ese. The string token is identified by the
quotation marks (either { ' } or { " }) that enclose it and can contain

anything in between.

Strings are the basic form of storage for all values within ARexx. Even
numbers are treated as strings * until they are used in an
 arithmetic operation. A machine address is also stored as character
string rather than as a number as it would be in some programming
languages. Even the symbols used as names of variables are shifted to
uppercase and treated as strings until they are assigned a value.

In an operation called concatenation , strings can be combined with one
another and with other types of expressions so easily that it sometimes
seems as though no operation is specified at all. Placing one string next
to another value is usually enough to specify a concatenation operation.

1.6   ... Expressions | Strings | Note (1 of 1) | NUMBERS AS TEXT

ARexx stores all values -- both numbers and characters -- as character
strings. Number strings are converted on the fly when an operation
requires a numeric value.

Any symbol or variable -- even one with a numeric value -- can be used in
expressions requiring textual input as the bizarre expression in the
instruction below illustrates:

```
    say right(25.05, 4) + ('2' || '5')     >>> 30.05
```

Here is the output of a TRACE I done on the instruction:

```
    >L> "25.05"        /* original value            */
    >L> "4"            /* take the 4 chars. on right */
    >F> "5.05"         /* ... and get 5.05          */
    >L> "2"
    >L> "5"
    >O> "25"           /* concatenate 2 & 5 to get 25 */
    >O> "30.05"        /* Add that to 5.05          */
    >>> "30.05"        /* and output this value     */
```

The quotation marks used in the subexpression on the right are not
necessary, but were included to show that even a digit entered as a
 literal string will be treated the same way as a number entered as a
 constant .

In both subexpressions, the digits are treated as text strings and used in
expressions that return a text value. Nonetheless, those values can be
used without further translation in the addition operation which requires
numbers as its terms.

1.7   ARexxGuide | Basic Elements | Expressions (3 of 5) | VARIABLES

A variable is a symbol that becomes a placeholder for another value and
can, in most cases, be used in place of the literal value it represents.
The process of giving a variable a new value is called assignment.

An assignment clause is the most common way to give a value to a
variable. The PARSE instruction is a powerful way to assign values to
several variables in a single statement. Even the DO instruction causes
an assignment to its index variable .

     Using variables
     Compound variables
     Special variables

Before it is used in an assignment, a variable name (its symbol), has a
default value as all symbols do: a string comprising the variable name
shifted to uppercase. Because they have a default value, unassigned
variables may be used in expressions:

    Example:
      /**/
      say hello world   >>> HELLO WORLD

Since they are not quoted and are therefore not treated as literal
 strings , both 'hello' and 'world' in this example are unassigned
variables. The output is translated to uppercase because ARexx shifts all
symbols to uppercase before using them.

      /* erroneous use of symbols */
      say good-bye world

This example produces:

    +++ Error 47 in line 2: Arithmetic conversion error

The '-' character is a reserved token that represents subtraction.
Finding the subtraction character in the expression, the interpreter
tries to subtract the value of the variable [bye] from the value of the
variable [good]. Since neither symbol has been assigned a numeric value,
the arithmetic operation fails.

Next: FUNCTIONS | Prev: Strings | Contents: Expressions

1.8   ... Expressions | Variables (1 of 3) | USING VARIABLES

The variables used in the examples below are simple symbols . More
complex collections of values can be stored using compound variables,
which are explained in the following nodes. Whichever form of symbol is
used, however, variables work the same way in an assignment.

As the name suggests, the value of a variable is not fixed and can be
changed at any time during execution of a program:

      /**/
      Two = 2

```
    Three = 3
    say Two + Three          >>> 5
    Two = 3
    Add = 'Two + Three'
    Six = 'is Six'
    say Two + Three          >>> 6
    say Add Six              >>> Two + Three is Six
```

The value assigned to a variable can be either textual or numeric and can
change from one assignment to another. Unlike some programming languages,
ARexx does not make a fundamental distinction among various types of data
stored by a variable.

```
    NOTE: 'Natural' data typing in ARexx
```

Next: COMPOUND VARIABLES | Prev: Variables | Contents: Variables

1.9   ... Expressions | Variables (2 of 3) | COMPOUND VARIABLES

```
    Compound variables
    ~~~~~~~~~~~~~~~~~~~
```
In any kind of assignment, a compound variable works like a simple
variable and may be used anyplace a simple variable can be used. The
compound variable is named by a compound symbol that looks something
like this:

```
    Foo.1.branch
```

The periods within the name identify the symbol as a compound variable.
Its definition is:

```
    <s.><b>[.<b>[.<b> ...]]
```

where <s.> is a stem symbol and <b> is a simple or constant symbol.
There can be up to 49 <b> branches in a compound variable.

The following nodes explain in more depth how compound variables are
formed and how they are used:

```
        Overview: Using compound variables
        Stem variables
        Extending stem variables
        Substituting values in compound variables
        Using strings as the derived name of a branch
        Setting the default value of a compound variable
        Finding values in a compound variable
```

Next: SPECIAL VARIABLES | Prev: Variable des. | Contents: Variables

1.10   ... Expressions | Variables | Compound variables (2 of 8) | OVERVIEW

```
   Using compound variables
   ~~~~~~~~~~~~~~~~~~~~~~~~~~
```
Compound variables provide a way to assign values to dynamically named
variables. Consider the problem of collecting data from a file of one-line
records: A value from each of several fields is to be assigned to a
variable. Defining a unique variable name for each field in each record
would become awkward if simple symbols were used: It would be nearly
impossible to assign values from within a loop to such variables.

Some languages offer a form of variable called an array for a task such as
this. In REXX, however, compound variables serve the purpose. A compound
variable for the task above might look like this:

```
   REC.35.4
```

where { 35 } is a count of the records, and { 4 } is a count of the fields
in the record. A variable with such a name can be used instead of a simple
variable as the target of any type of assignment -- in an assignment
clause, a PARSE instruction, or as the index variable in a DO
instruction.

The branch names of a compound variable may themselves be variables that
can be used in assignment statements:

```
   /**/
   REC.35.4 = 'Record 35, Field 4'       /* an assignment */
   i = 35; j = 4                          /* two more assignments */

   SAY REC.i.j                    >>> Record 35, Field 4
```

Even though we did not make an assignment to a variable named [REC.I.J], a
variable of that name did have a value when used in the SAY instruction.
Why? Because ARexx substituted the assigned values of the simple symbols
between the { . } periods: For [i], it substituted its assigned value of
{ 35 }, and for [j] the value of { 4 }. The substitutions turned the
variable [REC.i.j] into [REC.35.4].

If [i] and [j] had been used as the index variables in a pair of loops,
then they would have supplied the compound variable with a unique name on
each pass through the loops.

Next: STEMS | Prev: Introduction | Contents: Compound variables introduction

1.11   ... Expressions | Variables | Compound variables (3 of 8) | STEMS

```
   Stem symbols
   ~~~~~~~~~~~~
```
The name of a compound variable begins with a stem symbol , which, by
definition, ends with a period.

As with simple symbols, a stem symbol has the value of its name translated
to uppercase until it is used as the target of an assignment. After an
assignment, the stem variable will represent whatever value (text or
numeric) was assigned to it. The stem symbol can be assigned a value just

like a simple variable, but it is distinct from any simple symbol using
the same characters, but without the period:

```
/**/
Foo = 'arc'; Foo. = 'boo'
SAY Foo                                      >>> arc
SAY Foo.                                     >>> boo
```

The assignment to [Foo.] did not affect the value of [Foo] because they
are different variables. A stem variable, by itself, acts just like a
simple variable, but it can be greatly extended by adding branches to the
stem.

1.12   ... Expressions | Variables | Compound variables (4 of 8) | BRANCHES

```
Extending stem variables
~~~~~~~~~~~~~~~~~~~~~~~~~
```
Any stem variable can sprout branches when symbols are added after the
name of the stem. The added symbols can be either fixed symbols (usually
numbers) or simple variable symbols . Such a symbol -- now called a
compound variable -- can still serve as the target of an assignment. The
following assignment clauses might form the beginnings of the Foo family
tree.

```
        Foo.1 = 'Winnie'
        Foo.2 = 'Minnie'
        Foo.3 = 'Jimmie'
```

The grouping above can be extended to serve as a family tree with Winnie's
children added as new branches to [Foo.1]

```
        Foo.1.1 = 'Sunshine'
        Foo.1.2 = 'Moonglow'
        Foo.1.3 = 'Herbert W.'
```

Each of those branches might grow another to hold each child's year of
birth:

```
        Foo.1.1.1 = 1970
        Foo.1.2.1 = 1973
        Foo.1.3.1 = 1989
```

Used in this way, with numeric branches, compound variables will look
familiar to those who have used multi-dimensional arrays in other
languages. We will see in the next node that compound variables offer some
powerful alternatives to numeric values, but these examples show how they
can be used in what is to some programmers a familiar context.

Unlike the arrays used in other languages, however, compound variables in
ARexx are not part of a dimension of values. The variable [Stm.1000.2] is
still one variable and does not necessarily indicate that there are 999
other branches to [Stm.].

It is also worth noting again that compound variables act like simple
variables when they are used as the target of an assignment. Each uniquely
named compound variable can represent a value -- either text or numeric,
but that value can be changed at any time.

Although it need not concern the ARexx programmer in most cases, there is
a technical difference in the way the interpreter stores compound
variables; a difference that causes the interpreter to handle a compound
variable somewhat more slowly than it would handle a simple variable in
the same circumstance. The added versatility of compound variables will
usually make up for the slower assignment and reference process.

1.13   ... Expressions | Variables | Compound variables (5 of 8) | SUBSTITUTION

    Substituting values in compound names
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In the previous node, we introduced the beginnings of a family tree
attached to the stem variable [Foo.] using numeric constants as the
branches of the tree.

Dozens of numeric values representing family members or relationships
could become confusing. One way to avoid confusion is to use a more
familiar symbol in place of the numbers. Simple variables serve that
purpose well: The assignment { Winnie = 1 } might remind the programmer
that branch 1 to the [Foo.] stem refers to the family of Winnie. Once that
assignment is made, the symbol [Winnie] can be used in a compound variable
in place of the number 1:

    Foo.Winnie   means the same thing as Foo.1 if Winnie = 1

       For any simple symbol that appears as a branch of a compound
       variable, ARexx will substitute its assigned value before the
       compound variable is used.

Just as the subscripts of arrays used in other languages can be expressed
as variables (ary[i][j][k] or ary(i,j,k) instead of ary[4][8][1] or
ary(4,8,1)), so too can the branches of a compound variable be expressed
as simple variables.

If the assignment statements listed in the previous node were included as
part of the same program, then the following fragment would output the
names of Winnie's three children and their year of birth:

```
/* include the assignment clauses from the previous node */
Winnie = 1; BirthYear = 1
do Child# = 1 to 3
   say Foo.Winnie.Child# 'was born in' Foo.Winnie.Child#.BirthYear
end
```

The output would be:

     Sunshine was born in 1970
     Moonglow was born in 1973

```
    Herbert W. was born in 1989
```

One of the simple symbols used above ([Child#]) is an index variable to
the loop -- a common usage that makes numeric branches versatile, but --
as we will see in the next node -- the branches can be named with any kind
of value.

While a variable can be used to represent a branch name, an operation
cannot. A statement like { STEM.1+1 } will not be interpreted as [STEM.2].
It is, instead, an arithmetic operation that tells ARexx to add 1 to the
value of [STEM.1]. The VALUE() function can be used, in some situations,
to force the evaluation of an operation before a compound variable
reference is evaluated. In all cases, the result of an operaton can be
assigned to a variable that is then used in a compound variable:

```
    /**/
    Op = 1+1
    STEM.Op = 'Second value.'
```

[STEM.Op] become {STEM.2} before teh assignment is made.

Next: STRING BRANCHES | Prev: Substitution | Contents: Introduction

1.14   ... Expressions | Variables | Compound variables (6 of 8) | STRING BRANCHES

```
    Using strings as the derived name of a branch
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
In the previous nodes, numeric constants were used as the derived name
(the name that results after variable substitutions are made) for the
branches of compound variables. We've also used assigned symbols as
placeholders for the numeric values of the branches. What happens when one
of those variables is unassigned?

```
    /* no previous assignments used */
    Foo.Winnie.Child.1.BirthYear = 1970
    say Foo.Winnie.Child.1.BirthYear              >>> 1970
```

Because no assignments were made to the simple symbols that form the
branches of the stem, the derived name of the variable is
    [FOO.WINNIE.CHILD.1.BIRTHYEAR]
Following the standard treatment of symbols * , each of the branch symbols
is translated to uppercase. The variable with that text-heavy name can,
nonetheless, be used as a target in any kind of assignment.

The simple variables in the branches can also be assigned string values
which then become part of the derived name of the compound variable. If it
is added to the lines listed above, the following fragment will print out
the value of the same variable, once substitutions are made.

```
    FGen = 'WINNIE'; GenFld = 'BIRTHYEAR'
    say Foo.FGen.Child.1.GenFld                   >>> 1970
```

The derived name of the variable in the SAY instruction is, once again,
[FOO.WINNIE.CHILD.1.BIRTHYEAR].

It is important to note that the final derived name of a compound variable
is case sensitive:

```
/* two different variables: */
Foo.Winnie  = 1
FGen = 'Winnie'
say Foo.FGen                                >>> FOO.Winnie
```

The variable [Foo.FGen] is unassigned because 'Winnie' -- the value
assigned to [FGen] -- is entered in a mixture of upper and lowercase
letters. In the first assignment clause, on the other hand, [Winnie] is
used as an unassigned simple symbol. The symbol is translated to uppercase
before the compound variable is used as the target in the assignment. This
results in two distinct variables: The first one has the name [FOO.WINNIE]
and the second one has the name [FOO.Winnie].

The derived name a compound symbol may contain any kind of value. Not only
can the derived name of the branches be non-numeric, they can even include
non-printable characters:

```
/**/
Char = '07'x
say ASCII.Char                              >>> ASCII. /* flash */
ASCII.Char = 'Bell'
say ASCII.Char                              >>> Bell
```

Running this little program will cause a screen flash on most systems
because ARexx encounters a character that is printable only as a screen
flash when it attempts to print out the unassigned name of the compound
variable in the first SAY instruction. The variable with that unprintable
name can, nonetheless, be used in an assignment statement.

1.15   ... Expressions | Variables | Compound variables (7 of 8) | STEM ASSIGNMENT

```
    Setting default values with stem assignment
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
The name of the stem variable that forms the basis of each compound
variable is not changed by an assignment, but it does demonstrate a unique
trait: any value assigned to the stem will become the value of all
compound variables formed from it. A value previously assigned to one of
the compound variables will be replaced by the value of the new stem
assignment:

```
/**/
SAY Foo.1                                   >>> FOO.1
Foo.1 = 'I have a value'
SAY Foo.1                                   >>> I have a value
Foo. = 'Value of the stem'
SAY Foo.1                                   >>> Value of the stem
Foo.1 = 'A new value'
SAY Foo.1                                   >>> A new value
SAY Foo.2                                   >>> Value of the stem
```

The assignment to the stem variable changed the value of the compound
variables formed from it. This does not, however, prevent each compound
variable formed from the stem [Foo.] from having a unique value. When
[Foo.1] was used a second time as the target of an assignment, the new
value was assigned to it.

A stem assignment can give a default value to all possible branches of the
stem formed from the stem: Even though [Foo.2] had not previously been
used as the target of an assignment, it still has an assigned value. This
trait makes the statement { Stem. = '' } particularly useful in comparison
operations since it will give all compound variables formed from the stem
a default value equal to the null string.

Next: FINDING VALUES | Prev: Stem assignment | Contents: Introduction

1.16   ... Expressions | Variables | Compound variables (8 of 8) | FINDING VALUES

```
    Finding values in a compound variable
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
ARexx does not provide a search mechanism to locate values within a set of
compound variables. Except for their unique naming characteristics and the
 stem-assignment feature that allows a whole group of variables to be
changed with a single assignment, they work like simple variables.

Numeric branch values provide one way to locate a certain value in a
compound variable set:
```
    /**/
    do i=1 to Foo.0 until upper(Foo.i) = 'WINNIE'
    end
```

It has become an ARexx convention that the '0' branch in a set of numeric
compound variables holds the total number of variables in the set. That
convention is used in this example, which does an otherwise empty loop
through the values of [Foo.<x>] in search a specific value. (The value of
[i] will be one more than [Foo.0] if there is no match in the example.)

An quicker method, however, is to use 'WINNIE' as a branch name.
[Foo.WINNIE] can be used as the name of a variable that holds all the
information about the element in sub-branches, or as a variable that holds
just the numeric value of the branch that actually contains the
information:

```
    /**/
    Foo.1 = 'Winnie'
    Foo.WINNIE = 1
    Foo.1.1 = 'Sunshine'; Foo.1.2 = 'Moonglow'; Foo.1.3 = 'Herbert W.'
    Foo.1.0 = 3
    Name = 'WINNIE'
    Element = Foo.Name
    say Foo.Element"'s children are:"
    do i = 1 to Foo.Element.0
       say '   'Foo.Element.i
    end
```

Here, a text-named branch that is instantly addressable, is points to the

number-named branch that actually holds the desired information.

Text branches might also be used in the manner of a linked-list, with
information in each branch pointing to the next element in the list:

```
        Foo.0 = 'WINNIE'
        Foo.Winnie = 'info about Winnie'
        Foo.Winnie.Prev = ''
        Foo.Winnie.Next = 'MINNIE'
        Foo.Minnie = 'info about Minnie'
        Foo.Minnie.Prev = 'WINNIE'
        Foo.Minnie.Next = 'JIMMIE'
           /* ... */
```

Care must be taken, of course, to assure that the simple symbols used in
such a name are not inadvertently used in an assignment that would change
the derived name of the variable. One way to prevent such accidents is to
use a constant symbol like { 1Next } for the branch names. Because of the
leading digit, ARexx will treat that symbol as a fixed symbol that
cannot be assigned a new value.

Next: Compound var. | Prev: Default values | Contents: Compound var.

1.17   ... Expressions | Variables (3 of 3) | SPECIAL VARIABLES

Most variables are created and assigned within a program, but ARexx
maintains a few of them itself. The values of these special variables can
be read in the same way normal variables. They can even be used in
 assignment statements , although care should be taken when that is done
since the values will be changed by the interpreter under appropriate
conditions.

The three special variables are:

```
    RC          a numeric return code from a  command .
    RESULT      a string or numeric value set by command or function.
    SIGL        the line number of the clause that redirected program
                flow to an internal subroutine.
```

Next: Variables | Prev: Compound Variables | Contents: Variables

1.18   ARexxGuide | Basic Elements | Special variables (1 of 3) | RC

RC is one of three special variables controlled by the interpreter .

The interpreter assigns to this variable a numeric return code sent back
by a command or the number of a syntax error. The return code from
commands usually indicates the success or failure of the command. By
convention, a return code of 0 indicates that the command was executed
successfully. A code between 1 and 19 indicates an error condition of some
sort while a code greater than 19 indicates that the command failed.

When SIGNAL traps are enabled, the error number that caused a transfer
of control will be assigned to this variable.

1.19   ARexxGuide | Basic Elements | Special variables (2 of 3) | RESULT

RESULT is one of three special variables controlled by the interpreter .

When an external command is issued, the interpreter might assign to
RESULT a string value or numeric value returned by the command host. Such
result strings are available, however, only if the instruction
 OPTIONS RESULTS is used prior to issuing the command.

The interpreter will also assign to this variable the return code sent by
a function invoked by the CALL instruction.

1.20   ARexxGuide | Basic Elements | Special variables (3 of 3) | SIGL

SIGL is one of three special variables controlled by the interpreter .

When a jump in program flow is encountered, ARexx assigns to SIGL the line
number of the clause that caused the jump.

The SIGNAL and CALL instructions cause such a jump as does a
 function call to an internal subroutine.

When SIGNAL traps are used, the SIGL variable may be examined to determine
the line number of the clause that caused the transfer of control. If
 SIGNAL ON SYNTAX is used, for example, SIGL will hold the line number of
the clause containing a syntax error.

   Also see @{ " SOURCELINE() " link ARx_Func3.ag/SOURCELINE()} Function

1.21   ... Expressions | Variables | Note (1 of 1) | TYPELESS VARIABLES

In many programming languages, variables have a certain 'type'. In most
versions of BASIC, for instance, variables that represent text values have
names that end in the '$' character while variables for numeric values are
distinct and cannot be used to represent a string of text. C and Pascal
have even more restrictive typing and variables must be 'declared'
(assigned to a certain type) before they are actually used in a program.

Those methods were adopted because they represent more closely how the
values are treated by the machine the program runs on. REXX, on the other
hand, attempts to remove the user from such concerns. The language

processor (RexxMast on the Amiga) must handle such things, but the REXX
programmer can blithely ignore those issues.

In keeping with that spirit, variables are completely 'typeless' in ARexx:
they can hold a value of any type and can hold different value types at
different times in the execution of a program:

```
/**/
Var1 = 5; Var2 = 567.0097
say Var1 + Var2            >>> 572.0097
Var1 = 'Any type'; Var2 = 'of value'
Say Var1 Var2             >>> Any type of value
```

There are, of course, times when such freedom can give rise to problems
(as is the nature freedom). For example:

```
/**/
Var1 = 'One'; Var2 = 2
say Var1 + Var2
                +++ Error 47 in line 4: Arithmetic conversion error
```

Not even ARexx can do arithmetic on words. Most operators and
 functions require data of a certain type and will generate errors
similar to the one above if the supplied data does not meet the
requirements. The DATATYPE() function can be used to check the type of
data before a variable is used in an operation that might generate an
error.

Next, Prev & Contents: VARIABLE