Program Statements

Table of Contents

DIM name (dim1, dim2, …, dimK)

DIM ("dimension") is a statement that provides a short cut to creating roArray objects. It sets variable *name* to type "roArray", and creates Arrays of Arrays as needed for multi-dimensional arrays. The dimension passed to Dim is the index of the maximum entry to be allocated (the array initial size = dimension+1); the array will be resized larger automatically if needed.

```
Dim array[5]
```

Is the same as:

```
array=CreateObject("roArray",6,true)
```

Note that x[a,b] is the same as x[a][b]

Another Example

```
Dim c[5, 4, 6]
 For x = 1 To 5
     For y = 1 To 4
         For z = 1 To 6
             c[x, y, z] = k
             k = k + 1
         End for
     End for
 End for


 k=0
 For x = 1 To 5
     For y = 1 To 4
         For z = 1 To 6
             If c[x, y, z] <> k Then print"error" : Stop
             If c[x][y][z] <> k Then print "error": Stop
             k = k + 1
         End for
     End for
 End for
```

variable = expression

Assigns a variable to a new value.

Example

```
a$="a rose is a rose"
b1=1.23
x=x-z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

END

Terminates execution normally.

STOP

Interrupts execution return a STOP error. Invokes the debugger. Use "cont" at the debug prompt to continue execution, or "step" to single step.

GOTO label

Transfers program control to the specified line number. GOTO *label* results in an branch. A label is an identifier terminated with a colon, on a line by itself. Example:

Example

```
mylabel:
print "Anthony was here!"
Goto mylabel
```

RETURN [expression]

Used to return from a function back to the caller. If the function is not of type Void, return can return a value to the caller.

FOR counter = exp TO exp [STEP exp] / END FOR

Creates an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times. The general form is (brackets indicate optional material):
FOR *counter-variable* = *initial value* TO *final value* [STEP *increment*]
[*program statements*]
END FOR
In the FOR statement, *initial value, final value* and *increment* can be any expression. The first time the FOR statement is executed, these three are evaluated and the values are saved; if the variables are changed by the loop, it will have no effect on the loop's operation. However, the counter variable must not be changed or the loop will not operate normally. The first time the FOR statement is executed the counter is set to the "initial value" and to the type of "initial value".
At the top of the loop, the counter is compared with the *final value* specified in the FOR statement. If the counter is greater than the *final value*, the loop is completed and execution continues with the statement following the END FOR statement. (If *increment* was a negative number, loop ends when counter is less than *final value*.) If the counter has not yet exceeded the *final value*, control passes to the first statement after the FOR statement.
When program flow reaches the END FOR statement, the counter is incremented by the amount specified in the STEP *increment*. (If the *increment* has a negative value, then the counter is actually decremented.) If STEP *increment* is not used, an increment of 1 is assumed.

Example

```
For i=10 To 1 Step -1
 print i
End For
```

"EXIT FOR" is used to exit a FOR block prematurely.

FOR EACH item IN object

The FOR EACH statement iterates through each item in any object that has an "ifEnum" interface (enumerator). The For block is terminated with a END FOR statement. The variable *item* is set at the top of the loop to the next item in the object. Objects that are intrinsically ordered (like a List) are enumerated in order. Objects that have no intrinsic order (like AssociativeArray) are enumerated in apparent random order. It is okay to delete entries as you enumerate them.

"EXIT FOR" is used to exit a FOR block prematurely.
The following objects can be enumerated: roList, roArray, roAssociativeArray, roMessagePort.

Example

```
aa = { joe: 10, fred: 11, sue:9 }

For Each n In aa
 Print n;aa[n]
 aa.delete(n)
End For
```

WHILE expression / EXIT WHILE / END WHILE

The While loop executes until expression is false. The "exit while" statement can be used to terminate a while loop prematurely.

Example

```
k = 0
while k = 0
 k = 1
 print "loop once".
end while

while true
 print "loop once"
 if k <> 0 then exit while
end while
```

REM

Instructs the compiler to ignore the rest of the program line. This allows you to insert comments (REMarks) into your program for documentation. An ' (apostrophe) may be used instead of REM.

Example

```
Rem ** this remark introduces the program **
' this too is a remark
```

IF expression THEN statements [ELSE statements]

There are two forms of the IF THEN ELSE statement. The single line form (this one), and the multi-line or block form (see next section). The IF instructs the Interpreter to test the following *expression*. If the *expression* is true, control will proceed to the statements immediately following the expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line.

Example

```
if x > 127 then print "out of range"
If caveman = "fred" then print "flintstone" else print "rubble"
```

> THEN is optional in the above and similar statements.

Block IF, ELSEIF, THEN, ENDIF

The multi-line or block form of IF THEN ELSE is more flexible. It has the form:

```
if BooleanExpression then
statements
elseif BooleanExpression then
statements
else
statements
end if
```

There may be any number of elseif statements, or there may be none. The else statement may also be omitted. "elseif" can also be written as two words: "else if".

Example

```
msg = wait(0, p)
if type(msg) = "roVideoPlayerEvent" then
 if debug then print "video event"
 if msg.isFullResult()
   if debug then print "video finished"
  return 9
 end if
else if type(msg) = "roUniversalControlEvent" then
 if debug then print "button press "; msg.GetInt()
 HandleButton(msg.GetInt())
elseif msg = invalid then
 if debug print "timeout"
 return 6
end if
```

PRINT item list

Prints an item or a list of items on the console. The items may be either strings, number, variables, or expressions. Objects that have an ifInt, ifFloat, or ifString interface may also be printed.
The items to be PRINTed may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next print zone before printing the next item. If semi-colons are used, no space is inserted between the items printed.
Positive numbers are printed with a leading blank (instead of a plus sign); floating point numbers are printed with a trailing blank; and no blanks are inserted before or after strings.

Example

```
x=5:print 25; " is equal to"; x^2
 25 is equal to 25

a$="string":print a$;a$,a$;" ";a$
stringstring    string string

print "zone 1","zone 2","zone 3","zone 4"
zone 1          zone 2          zone 3          zone 4
```

Each print zone is 16 char wide. The cursor moves to the next print zone each time a comma is encountered.

```
print "print statement #1 ";print "print statement #2" Output:
print statement #1 print statement #2
```

Semi-colons can be dropped in some cases. For example, this is legal:

```
Print "this is a five " 5 "!!"
```

A trailing semi-colon over-rides the cursor-return so that the next PRINT begins where the last one left off. If no trailing punctuation is used with print, the cursor drops down to the beginning of the next line.

A few examples of printing enumerable objects:

Printing Enumerable Objects

```
Print {}
' this will print: <Component: roAssociativeArray> =  { }

Print {a:1}
' this will print: <Component: roAssociativeArray> =  { a: 1 }

Print []
' this will print: <Component: roArray> = [ ]

Print [5]
' this will print: <Component: roArray> = [ 5 ]
```

TAB (expression)
Moves the cursor to the specified position on the current line (modulo the width of your console if you specify TAB positions greater than the console width). TAB may be used several times in a PRINT list.

```
print tab(5)"tabbed 5";tab(25)"tabbed 25"
```

No punctuation is required after a TAB modifier. Numerical expressions may be used to specify a TAB position. TAB cannot be used to move the cursor to the left. If the cursor is beyond the specified position, the TAB is ignored.

POS (x)
Returns a number from 0 to window width, indicating the current cursor position on the cursor. Requires a "dummy argument" (any numeric expression).

```
print tab(40) pos(0) 'prints 40 at position 40
print "these" tab(pos(0)+5)"words" tab(pos(0)+5)"are";print tab(pos(0)+5)"evenly"
tab(pos(0)+5)"spaced"
```

FUNCTION([parameter [= default] AS type, …]) AS type / END FUNCTION

A function is declared using the function statement. In parentheses, one or more optional parameters to be passed may be declared. The return type of the function may also be declared. If the parameter or return type are not declared, they are assumed to be "dynamic"
Intrinsic types are passed by value (a copy is made). Objects are passed by reference.
Parameters can be of type:

- Integer
- Float
- Double
- Boolean
- String
- Object
- Dynamic
- Function

In addition to the above types, the return type can be:

- Void

Each parameter can have a default value, which is used if the parameter is not included in the call. The default parameter is declared by following the parameter name with an equal sign and the default value. If any parameter has a default value, then each following parameter must also have a default value. In the following examples, the functions add2 and add3 can be called with either one or two parameters.

Example

```
Function cat(a, b)
 Return a+b 'a, b could be numbers or strings
End Function

Function five() as Integer
 Return 5
End function

Function add(a as Integer, b as Integer) As Integer
 Return a+b
End function


Function add2(a as Integer, b=5 as Integer) As Integer
 Return a+b
End Function


Function add3(a as Integer, b=a+5 as Integer) as Integer
 Return a+b
End Function
```

Functions have their own scope.

The statement "Sub" can be used instead of "function" as a shortcut to a function of Void return Type.

If a function is called from an associative array, then a local variable "m" is set to the AssociativeArray that the function is stored in.

Example

```
Sub main()
 obj={
  add: add
  a: 5
  b: 10
 }

 obj.add()
 print obj.result
End Sub


Function add() as void
 m.result = m.a + m.b
End Function
```

If a function is not called from an AssociativeArray, then its "m" is set to an AssociativeArray that is global to the module, and persists across calls.

Anonymous Functions

A function is anonymous if it does not have a name. Note that Anonymous Functions do not currently create closures. An Anonymous Function can be declared like this:

```
myfunc = Function (a, b)
 Return a+b
End Function

print myfunc(1,2)
```

They can be used with associative array literals like this:

```
q = {
 starring : Function(o, e)
  str = e.GetBody()
  print "Starring: " + str
  toks = box(str).tokenize(",")
  For Each act In tok
   actx = box(act).trim()
   If actx <> "" Then
    print "Actor: [" + actx + "]"
    o.Actors.Push(actx)
   End If
  End For
  Return 0
 End Function
}

q.starring(myobj, myxml)
```