APL64 Control Structures

Basic Information

APL64 provides another method for controlling the execution within user-defined functions. Control structures identify blocks of statements that the system executes conditionally or repetitively, based on values in the control expressions.

Control structures comprise three parts: keywords, control expressions, and blocks of ordinary APL statements. Keywords identify the type of control structure and mark the boundaries of the blocks of statements they control. The control expressions define the conditions under which the system executes a block of statements. These statements perform the calculations or other actions you want to perform when the control expression is valid.

Throughout this section, control structures are always shown in lowercase for readability. They are case insensitive- in the system, so you can use :IF, :if, or :If interchangeably.

Table of Contents

Writing Conditional Structures

The simplest conditional control structure has a single condition statement, for example:

```
[30]  . . .
[31]  :if X=10
[32]        ⍝ Begin block of statements
[33]    Z←Y1 + Y2
. . .
[36]        ⍝ End block of statements
[37]  :endif
[38]  . . .
```

In the above example, the control structure spans seven lines. The keywords, which always start with a colon (:), mark the beginning and end of the structure. The control expression specifies that if the variable X has any value other than 10 when the system reaches line [31], the system does not execute the block of statements within the control structure. If the value of X is 10, the system executes the statements starting with line [32].

A control expression must evaluate to a Boolean scalar or one-element vector. If the value is 1, the condition is true. If the value is zero, the condition is false. If the expression returns a vector longer than one element, the system displays LENGTH ERROR and suspends execution.

The pattern of the keywords, those words beginning with a colon, defines the outer syntax of the control structure. The outer syntax has specific requirements, but you can build very complex structures.

Using Conditional Structures with Alternatives

You can include alternate blocks of statements for the system to execute.

```
[40]  . . .
[41]  :if X=10
[42]        ⍝ Block 1 of statements
[43]    Z←Y1 + Y2
. . .
[46]  :else
[47]        ⍝ Block 2 of statements
[48]    Z←Y3 + Y4
. . .
[51]  :endif
```

In this example, the system executes the first block of statements (from line [43] to the next keyword) if the value of X is 10. If the value of X is anything other than 10, it executes the second block of statements (from line [48] to the final statement in the structure).

You can use as many alternatives as you have meaningful conditions. The system evaluates the control expressions from the start of the structure and executes the block of statements associated with the first condition that evaluates to true. Note that the system does not evaluate subsequent conditions once it finds a condition that is true. The system executes only the block of statements following the first true condition.

```
[53]  . . .
[54]  :if X1=10
[55]        ⍝ Block 1 of statements
[56]    Z←Y1 + Y2
. . .
[59]  :elseif (X2-5)≥7.3
```

```
[60]        ⍝ Block 2 of statements
[61]    Z←Y3 + Y4
. . .
[64]  :else
[65]        ⍝ Block 3 of statements
[66]    Z←Y5 + Y6
. . .
[69]  :endif
```

You do not have to have an :else keyword. If you do have one, it must follow all the :elseif keywords in its outer structure. In a structure without an :else keyword, the system can execute one block of statements or none. If none of the control expressions associated with the :if statement or the :elseif statements is true, the system continues with the statement following the final statement of the structure.

Using Condition Extension Keywords

You can also make a block of conditions by compounding the control expressions. You can make multiple conditions with the :andif keyword or alternate conditions with the :orif keyword.

```
[70]  . . .
[71]  :if X1=0
[72]  :andif X219
[73]        ⍝ Block 1 of statements
[74]    Z←Y1 + Y2
. . .
[77]  :elseif X3≤5
[78]  :orif X4≥6
[79]        ⍝ Block 2 of statements
[80]    Z←Y3 + Y4
. . .
[84]  :endif
```

When you use the :andif condition extension statement, all the control expressions in the block of conditions must be true for the system to execute the block of statements following the block of conditions. If any condition is not true, the system does not execute the block of statements.

The system evaluates the condition statements sequentially. If it finds a control expression that is not true, it does not evaluate the other statements in that block of conditions. This makes it possible to use a combination of conditions such as the following:

```
:if X10
:andif (X2÷X1)≥19
```

A statement using the APL primitive AND function, for example, (X10)^(X2÷X1)≥19, causes a DOMAIN ERROR if X1 is zero.

When you use the :orif condition extension statement, the system evaluates each control expression in that block independently. If any control expression in the block is true, the system executes the block of statements immediately following the block of conditions; in the above example, starting with line [79]. If none of the statements is true, the system does not execute the block of statements.

The system evaluates the condition statements sequentially. If it finds a control expression that is true, it does not evaluate the other statements in that block of conditions. This makes it possible to use a combination of conditions such as the following:

```
:if X1=0
:orif (X2÷X1)<19
```

A statement using the APL primitive OR function, for example, (X1=0)∨(X2÷X1)<19, causes a DOMAIN ERROR if X1 is zero.

You can string multiple :andif statements in one block, and you can string multiple :orif statements in one block, but you cannot mix the two types of condition extension statements in the same block of conditions.

Writing Repetitive Structures

The simplest repetitive control structure has a single condition statement, for example:

```
[85]  . . .
[86]  :while X>20
[87]        ⍝ Begin block of statements
[88]    Z←Y1 + Y2
. . .
[91]  :endwhile
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. The control expression specifies that if the variable X is less than or equal to 20 when the system reaches line [86], the system does not execute the block of statements. If X is greater than 20, the system executes the statements starting with line [87].

When the system reaches line [91] it branches back to line [86] and evaluates the condition again. This typically implies that one of the statements in the block alters the value of X, a statement within the block branches outside the control structure, or something external to the function interrupts the function while the system executes the block. Otherwise, your program is in an infinite loop.

In this structure, the system evaluates the condition once more than it executes the block of statements. Since the test precedes the block of statements, it is known as a leading test.

Using a Trailing Test

You can also have one condition statement at the end of the control structure.

```
[93]  . . .
[94]  :repeat
[95]        ⍝ Begin block of statements
[96]    Z←Y1 + Y2
. . .
[99]  :until X>30
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. When the system reaches line [94], it automatically executes the block of statements. The control expression specifies that if X is greater than 30 when the system reaches line [99], it continues with the next line. If X is less than or equal to 30, the system branches back to line [94] and executes the block again.

This typically implies that one of the statements in the block alters the value of X, a statement within the block branches outside the control structure, or something external to the function interrupts the function while the system executes the block. Otherwise, your program is in an infinite loop.

In this structure, the system executes the block of statements before it evaluates the condition the first time. Since the test follows the block of statements, it is known as a trailing test.

Using Leading and Trailing Tests

You can combine leading and trailing tests in one control structure. The control expressions can test the same variable, as in the example below, or different variables.

```
[101]  . . .
[102]  :while X>20
[103]        ⍝ Begin block of statements
[104]    Z←Y1 + Y2
. . .
[107]  :until X>30
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. The control expression specifies that if X is less than or equal to 20 when the system reaches line [102], the system does not execute the block of statements. If X is greater than 20, the system executes the block.

When the system reaches the end of the control structure, it evaluates the second control expression. This control expression specifies that if X is greater than 30, it continues with the next line. If X is less than or equal to 30, the system branches back to line [102] and evaluates the first control expression again.

In this structure, the trailing condition must be false and the leading condition true for the system to execute the block of statements more than once. If the trailing condition is false, it does not necessarily mean the system executes the block a second time. If the value of X is reduced to less than 20 during the first execution of the block of statements, the system continues with the statement following line [107] even though X is not greater than 30.

Writing Repetitive Structures with No Control Expression Test

You can also have a repetitive control structure with no control expression.

```
[109]  . . .
[110]  :repeat
[111]        ⍝ Begin block of statements
[112]    Z←Y1 + Y2
. . .
[115]  :endrepeat
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. When the system reaches line [110], it automatically begins to execute the block of statements. If the system reaches line [115], it branches back to line [110] and begins to execute the block again. This implies that you have a statement within the block that branches outside the control structure or ends your program.

Using Condition Extension Keywords

You can also make a block of conditions in a repetitive control structure by compounding the control expressions. You can make multiple conditions with the :andif keyword or alternate conditions with the :orif keyword when you combine them with either the :while or :until keywords.

Valid outer syntax structures include:

```
:while     :while
:andif     :orif
block of statements     block of statements
:endwhile     :endwhile

:repeat     :repeat
```

block of statements block of statements
:until :until
:andif :orif

You can also combine condition keywords in a structure that combines leading tests with trailing tests.
:while :while
:andif :orif
:andif block of statements
block of statements :until
:until :andif
:andif :andif

:while :while
:andif :orif
block of statements :orif
:until block of statements
:orif :until
:orif :orif

Writing Iterative Structures

The iterative control structure has a two-part control statement. The first keyword specifies a controlled variable, and the second keyword defines an expression. The system assigns each of the values in the expression to the controlled variable and executes the block of statements once for each value. For example:

```
[117]  . . .
[118]  :for J :in ⍳10
[119]        ⍝ Begin block of statements
[120]    Z←Mat[J]
. . .
[123]  :endfor
```

In the above example, the control structure spans six lines. The keywords mark the beginning and end of the structure. When the system reaches line [118], it automatically begins to execute the block of statements, with J having the first value in the expression, (1 in this case).

When the system reaches line [123], it branches back to line [118], assigns the next value to J, and begins to execute the block again. The system executes the block of statement once for each value of the expression. When the system reaches a line containing a :for statement, it always executes the block of statements unless the expression following :in is empty.

The expression does not have to be an arithmetic sequence. You could craft a nested vector, for example:

```
:for J :in 2.2 (A-B) (20+2×⍳4) (⍟10)
```

The system would execute the block of statements four times, with J set to the value produced by each item of the vector in turn.

Note that localization of variables works only at the function level and not at the level of a control structure. When the system exits the structure, the controlled variable has the last value assigned it. If the variable had a value within the function before the system reached the beginning of an iterative control structure, that value is gone. However, you can reassign a value to the controlled variable within the block of statements; the execution of the iterative structure does not depend on the value of the controlled variable. When the system branches back to the :for statement, it simply assigns the next value to the

controlled variable. Similarly, the values of the vector following the :in keyword are evaluated only once at the beginning of the structure. If you specify the vector with a named variable, you can change the variable within the structure without altering the execution of the function (unless you exit the structure and return to it).

Continuing a Repetitive or Iterative Structure

To simplify the coding of certain kinds of looping patterns within repetitive or iterative control structures, you can use the :continue and :continueif keywords. The :continue keyword takes no argument, but :continueif does. When the system executes these statements, it branches to the end of the block of statements but does not necessarily leave the control structure. In a repetitive structure that ends with an :until keyword, the system performs the trailing test. In a repetitive structure that has the :while . . . :endwhile pattern, the system performs the leading test. In an iterative structure, the system starts the next loop if there is another value in the expression following the :for statement. This keyword has the effect of curtailing only the current loop of execution within a control structure. In the example below, which simulates processing data for weekdays, when the system reaches line [130] and executes the :continue statement, it next executes the trailing test on line [133] to determine whether to return to line [126] or continue to line [134].

```
[125]  . . .
[126]  :repeat
[127]      ⍝ Begin block of statements
[128]    :if DayName = Saturday
[129]    :orif DayName = Sunday
[130]      :continue
[131]    :endif
. . .
[133]  :until EndOfMonth = 1
[134]
```

Exiting a Repetitive or Iterative Structure

To exit one of these structures without meeting the completion condition or exhausting the list of arguments in a :for statement, you can use the :leave keyword. This keyword takes no argument. When the system executes this statement, it exits the block of statements defined by the beginning keyword and its corresponding :end keyword. The possibilities for the terminal identifier include :end, :endwhile, :endrepeat, :until and :endfor.

When the system encounters a :leave statement within one of these structures, it executes the statement following the terminal keyword. In the example below, if the system reached line [140] and executed the :leave statement, it would next execute the statement following line [144].

```
[136]  . . .
[137]  :while transactions > 0
[138]      ⍝ Begin block of statements
[139]    :if Balance < ItemValue
[140]      :leave
[141]    :endif
[142]    Balance←Balance - ItemValue
. . .
[144]  :endwhile
```

Writing Selection Structures

Selection control structures use multiple control expressions. The system identifies the block of statements to execute based on two expressions matching. Note that match means in the sense of the APL primitive function Match (≡); the expressions must have the same rank, shape and value. It is not sufficient if they are equal. A scalar 5 equals a one-element vector whose value is 5, but the two do not match.

The simplest selection control structure has one top-level control expression and a series of case expressions, each of which can potentially match the top-level expression; for example:

```
[146]  . . .
[147]  :select ×value
[148]  :case 1  ⍝ Positive case
[149]          ⍝ Block 1 of statements
[150]     Z←Y1 + Y2
. . .
[153]  :case ¯1  ⍝ Negative case
[154]          ⍝ Block 2 of statements
[155]     Z←Y3 + Y4
. . .
[158]  :endselect
```

In the above example, the control structure spans twelve lines. The keywords, which always start with a colon (:), mark not only the beginning and end of the structure, but also the segments of the structure.
The combined control expressions specify that if the signum of value is 1 (value is positive), when the system reaches line [147], the system executes the first block of statements. If the signum of value is ¯1 (value is negative) when the system reaches line [147], the system executes the second block of statements.

If the value of the :select expression does not match the value in any of the :case expressions, the system branches to the :endselect statement and begins executing the statement that follows.
Note that with the selection structure, the control expression in the :select statement does not have to evaluate to a Boolean 1. When the system reaches the :select statement, it evaluates that expression to find its value. It then evaluates the control expression in the first :case statement. The system tests the two expressions to see if they match. If the result of the match is a Boolean 1, the system executes the block of statements that follows the :case statement.

If the control expression for the first :case statement does not match the top-level expression, the system evaluates subsequent :case control expressions until it finds one that matches or exhausts the possibilities. You can use as many alternatives as you have meaningful conditions. Once the system finds a match, it does not evaluate any further expressions. The system executes at most one block of statements following a :case statement.

Writing Selection Structures with Alternate Matching Values

If there are several cases for which you want to execute the same block of statements, you can extend the possibilities by using a :caselist statement that specifies multiple alternate expressions. If any of the expressions in the list matches the :select statement expression, the system executes the block of statements associated with that :caselist. You must define the conditions so that any of the alternatives might match the top-level expression. You cannot have multiple expressions in the :select statement for a :case or :caselist expression to match.

```
[160]  . . .
[161]  :select state
[162]  :case 'DE'
[163]          ⍝ Block 1 of statements
[164]     Z←Y1 + Y2
. . .
[167]  :caselist 'GA' 'HI' 'NY'
[168]          ⍝ Block 2 of statements
[169]     Z←Y3 + Y4
```

```
. . .
[172]  :endselect
```

In this example, if state matches 'DE', the system executes the first block of statements. If state matches 'GA', 'HI', or 'NY', the system executes the second block of statements. You can have multiple :case and :caselist statements; they can be interspersed. The system executes at most one block of statements within this structure.

Writing Selection Structures with a Default Alternative

You can include a block of statements for the system to execute in the event that none of the :case or :caselist statement expressions matches the :select statement expression.

```
[174]  . . .
[175]  :select X1
[176]  :caselist J1 J2 J3
[177]        ⍝ Block 1 of statements
[178]    Z←Y1 + Y2
. . .
[181]  :case   K
[182]        ⍝ Block 2 of statements
[183]    Z←Y3 + Y4
. . .
[186]  :else
[187]        ⍝ Block 3 of statements
[188]    Z←Y0
. . .
[191]  :endselect
```

In this example, if X1 matches any of the variables J1, J2 or J3, the system executes the first block of statements. If X1 matches the variable K, the system executes the second block of statements. If X1 does not match any of the variables, the system executes the third block of statements.

If you have an :else keyword, it must follow all the :case and :caselist keywords in its outer structure. In a selection structure without an :else keyword, the system can execute one block of statements or none. In a selection structure with an :else keyword, the system executes one block of statements.

Using Control Expressions of Higher Rank

If you place multiple values or expressions on the :select line, you create an expression of higher rank (for example, a vector) which the :case statement expression must match. If your top-level expression is :select (1 + 2 3), then a :case expression must evaluate to a two-element vector with the value (3 4) to match. If the :case expression evaluates to either the scalar 3 or the scalar 4, it does not match. If you have the expression :caselist (3 4), that also does not match, because the values are interpreted as two different list values. If you have either :caselist (3 4) (5 6) or :caselist (3 4) ⍬, there is a match, and the system executes the block of statements associated with the :caselist.

Writing Nested Control Structures

You can nest one control structure inside another. When you nest one structure within another, the entire body of the inner control structure must be within one block of statements that the outer control structure controls.

For readability, APL2000 recommends that you keep the colons of the keywords of any structure on the same vertical column in your function. You should indent blocks of statements several characters. If you have a

nested structure, indent those keywords from the keywords of the structure that contains it, and indent the block of statements of the nested structure farther still.

```
:if condition
  block of statements
:elseif condition
  block of statements
  :if condition
    block of statements
  :else
    block of statements
  :endif
  block of statements
:else
  block of statements
:endif
```

You can nest different types of structures within each other; for example, you can nest a conditional structure within a repetitive structure. You can nest more than one structure within another, and you can nest structures more than two levels deep.

If you use indentation in your functions, make sure you select "Enable Automatic indentation in functions" on the Editor Options window, which you reach from the Options menu. Then after you press Tab at the beginning of a line in the full screen editor, APL inserts the specified number of spaces on each subsequent line until you press Backspace at the beginning of a line. This holds true for each level of indentation you add.

Other Syntax Rules

There are other rules that apply to all the control structures:

Errors

When you attempt to save a function in the function Edit window, the interpreter analyzes the outer syntax of a control expression and warns you of any OUTER SYNTAX ERROR. However, when you define (□def, □defl, or □fx) a function under program control that contains such an error, the system does not display an error message at that time. If a control structure is ill-formed, the system displays the error message when the function is called. You cannot execute any part of a function with an outer syntax error.

Other errors within control structures generate the same message they would elsewhere.
An attempt to execute a control statement in immediate execution mode results in a SYNTAX ERROR. You can use control structures only in user-defined functions.
An attempt to branch illegally into a control structure generates a DESTINATION ERROR. See the "Branching Restrictions in Control Structures" section later in this chapter.

Some modifications to suspended functions containing control structures may mark the suspension as having SI DAMAGE. When this happens, use the)reset or)sic commands or a naked branch statement to clear the function call from the stack; then invoke the function again..

The :end keyword

You can use :end in any control structure in place of the more explicit keywords shown above; for example, :endwhile. You cannot interchange the others, however.
Specifically, you cannot use the :endif keyword when you combine the condition extension keywords :andif or :orif with the keywords :while or :until. You must use :endwhile or :endrepeat in their structures, or you can use :end in either type of structure.

Interspersing Comment Lines and Blank Lines

You can place a blank line or a comment line between any two lines within a control structure without affecting the way the system executes your function.

You can use the diamond symbol (⋄) to combine lines in a control structure. Comments must be the rightmost portion of a line, if you include them. A label cannot follow a diamond on a line. The system executes the portion of the line to the left of the first diamond first, and treats the portion after each diamond on the line as though it were on a separate line; for example, the following line is a valid first line for a selection structure.

```
:select x ⋄ :case 1 ⋄ Y←3
```

You can place a label on any line in a control structure that does not begin with a keyword. See the "Using the Branch Statement" section earlier in this chapter for information on labels.

Branching Restrictions in Control Structures

You cannot branch into an iterative control structure (a :for loop). An attempt to do so results in a DESTINATION ERROR.

Since you cannot place a label on a line that contains a keyword, you cannot branch to a keyword statement using labels. In some cases you can branch to a keyword statement using absolute line numbers, although this is not recommended. You can branch into some control structures using labels.
You cannot branch to a :case statement or to a :caselist statement in a selection control structure. You can branch to a line within the block of statements following a :case or a :caselist. In this situation, the system executes the remainder of the block of statements up to the line where it encounters the next keyword. It then branches to the :endselect statement and continues.

You can branch into a conditional or a repetitive control structure. In this situation, the system begins execution at the line to which you branch and continues execution of the control structure following the same rules as if you entered the control structure without branching.

You can branch out of any control structure. This terminates execution of the control structure, and the system continues execution at the line to which you branch.

Using Keywords to Branch within Functions

To change the sequence of execution within a user-defined function, you can use the :goto keyword. This keyword requires exactly one argument, which must be a label identifier; it cannot be a variable, number, or other expression. When the system encounters this keyword, it continues executing the function with the line on which the label occurs. The :goto keyword is an alternative to using a branch arrow with a label. The following two examples are equivalent. When the system reaches line [193] of the function, the system branches to line [195] and continues execution.

```
[193] → TARGET
[194]
[195]  TARGET:
```

```
[193] :goto TARGET
[194]
[195]  TARGET:
```

Exiting a Function

As an unequivocal and easy-to-read method for ending the execution of a function, you can use the :return and :returnif keywords. The :return keyword takes no argument except when a function declares a result variable in which case the argument is the value returned . The :returnif keyword takes an argument that is the conditional statement. When the system executes these statements, it exits the function but continues execution.

The following examples are equivalent. The system stops executing the current function and returns either to the function that called this one or to immediate execution mode.

[198] :return
[198] :return value
[198] → 0

Control Structures Reference

:ASSERT

Syntax:

:ASSERT *control-expression*

The :ASSERT statement takes an conditional expression as its argument. This expression is only evaluatedwhen debug mode is ON. It is not evaluated and therefore has zero execution overhead when debug mode is OFF. When the expression evaluates to singleton value 1, the assertion is considered successful, and execution continues with the next logical statement. Similarly, an empty character vector is also considered to be a success. And likewise, if the expression does not return a value (such as a user defined function that doesn't set a result value) then the assertion is considered to succeed, and execution continues at the next statement. All other values cause an ASSERTION FAILURE error to be signaled.

:DEBUG

Syntax:

:DEBUG *expression*

The :DEBUG statement conditionally executes a single expression (whose body follows the :DEBUG keyword) if and only if debug mode is ON. Implicit output is OFF during :DEBUG statement execution.Unassigned result values in :DEBUG expressions are not displayed in the APL64 session. Session output can only be accomplished explicitly via assignment. Often however, the LOG system functionwill be used to write something to a debug log file and/or the Windows Application Event Log instead.

:DEF

Syntax:

:DEF *expression*

Local (Inner) functions can be defined inside of 'outer' functions via the :DEF control structure declaration. While the outer function is executing, it can make calls to the inner function. Unlike using ☐DEF to dynamically define a local function inside another function, the :DEF control structure does not have any execution overhead associated with it.

- The name of the inner function is implicitly localized in the outer function.
- The inner function is statically defined once, when the outer function is defined and doesn't incur any execution overhead to redefine the inner function each time it is needed. When ☐DEF is used to define a local function, there is a substantial execution cost incurred each time the function is defined, even before it is first executed.
- The :DEF control statement is easier to read and easier to code than functions specified using ☐DEF or ☐FX since quoted strings are not needed.
- The :DEF control statement supports setting of stop and trace lines via the session manager editor, just like in non-inner functions.
- Functions declared via :DEF are local to the functions they are defined within and don't "pollute" the global workspace with specialized functions that are only applicable to the context of the containing function where they are defined.
- The names of the outer function and inner function can be combined in a dot-notation name to call the inner function directly.

Example:

```
      ⎕vr 'Demo'
    ∇ Demo
[1]   '- before Inner'
[2]   Inner
[3]   '- after Inner'
[4]
[5]   :Def Inner
[6]       '+ inside Inner'
[7]   :EndDef
[8]
[9]   '* Exit'
    ∇

      Demo
- before Inner
+ inside Inner
- after Inner
* Exit
```

:FOR

Syntax:

:FOR *variable(s)* :in *expression(s)*
block of statements
:ENDFOR

The :FOR statement executes a block of statements once for each value in an expression, assigning the value to a specified variable.

:FOREACH

Syntax:

:FOREACH *variable(s)* :in *expression(s)*
block of statements
:ENDFOR

The name list between the :FOREACH and :IN keywords must contain two or more names. The argument value to the right of :IN must be a vector containing the same number of items as the number of names in the name list between the :foreach and :in keywords. Each argument item must be the same shape or a singleton

:CONTINUE :CONTINUEIF

Syntax:

:CONTINUE
:CONTINUEIF *condition expression*

You can use the :CONTINUE and :CONTINUEIF keywords to restrict the execution of a repetition. If the system encounters either within the loop, it branches to the :ENDWHILE or :UNTIL statement and performs the appropriate tests. If the test or tests are satisfied, the system starts another repetition. If the :WHILE statement is no longer true or the :UNTIL statement is true, the system continues with the statement following :ENDWHILE or :UNTIL.

You can use the :LEAVE and :LEAVEIF keywords to exit the loop. If the system encounters either within the loop, it branches to the :ENDWHILE or :UNTIL statement and, without performing any test, continues execution with the statement following :ENDWHILE or :UNTIL.

:LEAVE :LEAVEIF

Syntax:

:LEAVE
:LEAVEIFIF *condition expression*

You can use the :LEAVE and :LEAVEIF keywords to exit the loop. If the system encounters either within theloop, it branches to the :ENDWHILE or :UNTIL statement and, without performing any test, continues execution with the statement following :ENDWHILE or :UNTIL.

:ENDFOR

Syntax:

:FOR{EACH} *control variable* :IN *expression*
statement(s)

:ENDFOR

The :FOR control structure has a two part control-expression. The first part, *control variable*, specifies a variable. The second part is an APL expression. The system assigns each of the values in *expression* to*control variable* and executes the block of statements once for each value.

When the system begins to execute a :FOR control structure, it initializes *control variable* to the first value in *expression*. If *expression* is empty, the system does not execute the statements in the control structure.

The system does not localize control variables in :FOR control structures. When the system exits the control structure, the control variable has the last value the system assigned it. If the control variable had a value before the system began to execute the :FOR control structure, that value is overwritten in the control structure.

This control structure must end with an :ENDFOR or :END.

The :FOREACH statement behaves similarly to the :FOR statement except items are iterated in parallel. The name list between the :FOREACH and :IN keywords must contain two or more names. The argument value to the right of :IN must be a vector containing the same number of items as the number of names in the name list between the :FOREACH and :IN keywords. Each argument item must be the same shape or a singleton.

You can use the :CONTINUE and :CONTINUEIF statements to restrict the execution of an iteration. If the system encounters either within the loop, it branches to the :ENDFOR statement and assigns the next value,if there is one, to the control variable. If the current value is the last value in *expression*, the system continues with the statement following :ENDFOR.

You can use the :LEAVE and :LEAVEIF statements to exit the loop. If the system encounters either withinthe loop, it branches to the :ENDFOR statement and continues execution with the statement following :ENDFOR.

:GOTO

Syntax:

:GOTO *label*

The :GOTO statement causes a function to branch to the line on which the label occurs. This statement hasthe same effect as using the branch arrow followed by a label.

:RETURN
:RETURNIF
:RES

Syntax:

```
:RETURN result
```

The :RETURN statement causes the system to exit the current function but continues execution. This statement has the same effect as using the branch arrow followed by a zero or a nonexistent line number.

In addition, the following :RETURN with result value argument:

```
:IF condition
    :RETURN result
:END
```

can be replaced by the following single statement:

```
    :RETURNIF condition :RES result
```

:IF
:ANDIF
:ORIF
:ELSE
:ELSEIF
:EX
:ENDIF

Syntax:

```
:IF control-expression 1
:AND control-expression 2
    statement 1
    :EX expression
:ELSEIF
    control-expression 3
    statement 2
:ELSE
    statement 3
:ENDIF or :END
```

The *control-expression* must evaluate to a Boolean scalar or a one element vector. If the value is 1, thecondition is true. If control-expression equals 0, the condition is false.

The system evaluates each control-expression, beginning at the top, and executes the statements following the first control-expression that evaluates to true (1). You must end an :IF control structure with either :ENDIF or :END. The :ELSE and :ELSEIF parts and their following statements are optional. If you douse the :ELSE part, it must follow any :ELSEIF parts.

You can also test for several conditions at once by compounding control-expressions. The :ANDIF keyword allows you to test for multiple conditions. The :ORIF keyword lets you test for alternate conditions.

The :EX statement is used in cascading decision statements (such as :AND/:OR extensions of :IF, :WHILE,etc.,

statements as a place to compute something before taking the next decision.

Note: The :EX clause is not allowed in ICS expressions.

When you use :ANDIF, all expressions must be true for the system to execute the statement(s) following the :IF/:ANDIF control-expressions.

:IF *control-expression1*

:ANDIF *control-expression2*

statement 1
:ENDIF

When you use :ORIF, the system executes the statements following the :IF/:ORIF control-expressions ifeither of the expressions is true.

:IF *control-expression1*

:ORIF *control-expression2*

statement 1
:ENDIF

:IFTEST

Syntax:

:IFTEST

test statements

:ELSE

non-test statements

:ENDIFTEST or : ENDIF or :END

The :IFTEST statement controls a block of test statements that are conditionally executed in the test clausewhen a function is executed in test mode. If an :ELSE statement is present, the non-test statements are executed when not run in test mode.

:REPEAT

:UNTIL

:ENDREPEAT

Syntax:

:REPEAT

statement(s

:UNTIL *expression*

:REPEAT

statement(s)

:ENDREPEAT

The :REPEAT control structure allows you to execute a statement or block of statements repeatedly until *expression* is true (1). In a :REPEAT control structure, the statements in the control structure are executedat least once.

If you use the :ENDREPEAT statement at the end of the control structure, be sure to provide some meanswithin the structure to end the :REPEAT loop. You can include a statement within the block that branches outside the control structure or ends your program.

You can use the :ANDIF and :ORIF statements with :UNTIL to create compound control-expressions. For example:

:REPEAT

statement(s)

:UNTIL *expression*

:ANDIF *expression*

:ANDIF *expression*

:REPEAT

statement(s)

:UNTIL *expression*

:ORIF *expression*

You can substitute :END for :ENDREPEAT.

You can use the :CONTINUE and :CONTINUEIF keywords to curtail the execution of a repetition. If the system encounters either within the loop:

- it branches to the :ENDREPEAT statement and starts another repetition, or
- it branches to the :UNTIL statement and performs the appropriate test. If the test is not true, the systemstarts another repetition. If the :UNTIL statement is true, the system continues with the statement following :UNTIL.

You can use the :LEAVE and :LEAVEIF keywords to exit the loop. If the system encounters either within theloop, it branches to the :UNTIL or :ENDREPEAT statement and, without performing any test, continues execution with the statement following :UNTIL or :ENDWHILE.

:SELECT

:CASE

:CASELIST

:LIKE

:NEXTCASE

:ENDSELECT

Syntax:

:SELECT *expression*

:CASE
expression
statement
(s)

:CASE *expression*

:NEXTCASE

statement(s)

*:*LIKE
expressio
n
statemen
t(s)

:CASELIST (*expression*) (*expression*)
(*expression*) ...*statement(s)*

:ELSE

statement(s)

:ENDSELECT

:SELECT control structures allow you to use multiple control-expressions within the control structure. Each :CASE keyword marks off a block of statements. The system evaluates the expression following the :SELECT keyword to find its value, and then evaluates the :CASE expressions until it finds one thatmatches the :SELECT expression. The system executes the statements associated with that :CASE expression and exits the control structure.

expression can be a value, variable, or expression. The results of a :SELECT and :CASE expression not only must have the same value but must also have the same rank and shape. For example, a scalar 5 hasthe same value as a one-element vector that contains a 5; however the two 5s have different shapes and ranks.

If there are several alternate conditions under which you want to execute the same block of statements, use the :CASELIST statement . The parentheses around :CASELIST expressions are optional; however, they ensure that the system evaluates each expression as you intend. The following table shows examples of parentheses in a :CASELIST statement.

The system interprets...	as...
:CASELIST (3 4)	two scalar values
:CASELIST (3 4) (5 6)	two two-element vectors
:CASELIST (3 4) 0	one two-element vector and a scalar.

The :NEXTCASE statement flows from one case of a :SELECT statement into the next without branching.This can be useful for parsing varying length arguments where each element gets special processed and then falls into the case for the next smaller length.

The :LIKE clause is an alternative to the :CASE or :CASELIST clause in the :SELECT statement. Ratherthan matching the :SELECT argument by value equivalence, it selects character scalar and vector casesmatching its pattern similar to the :CATCH clause.

The :LIKE clause supports wildcard pattern matching codes as an alternative form of :CASE or :CASELIST clause in a :SELECT block. The :LIKE control characters in patterns are shown below:

STAR * matches zero or more of any charactersQUERY ?
matches one of any character
RHO is reserved for the future to denote a regular-expressionSHARP # is reserved for the future
SEMI ; separates patterns
SLASH \ makes the control character following it behave as a literal

The :ELSE keyword is optional. It allows you to specify a default action in the event that none of the :CASE or :CASELIST expressions match the :SELECT expression. If you do use the :else keyword, it must follow any :CASE or :CASELIST keywords in the control structure.

You can substitute :END for :ENDSELECT.

:TRY

:TRYALL

:CATCH

:CATCHALL

:FINALLY

:ENDTRY

Syntax:

:TRY

Statements to try
:CATCH first condition expression
Statements to handle exceptions satisfying the first condition
:CATCHIF second condition expression
Statements to handle exceptions satisfying the second condition
:CATCHALL

Statements to handle all other exceptions
:ENDTRY

The structure always begins with a :TRY statement and ends with a :ENDRY statement. As with other controlstructures, the keywords are case insensitive and :END may be substituted for :ENDTRY.

There must be at least one :CATCHIF or :CATCHALL clause. There may be no more than one :CATCHALL clause, which must be the last.

The :CATCHIF statement consists of its keyword followed by an expression that yields 1 or 0. The other control statements consist of only a keyword.

Each control statement except :ENDTRY is followed by a block of APL statements (which may be empty). These blocks may contain additional try-catch and other control structures. They must be properly nested.

The :TRYALL statement provides resume-on-next-line error handling, which in some cases is easier to usethan ⎕ELX or :TRY … :CATCH. Errors occurring in its context are handled by skipping any remaining statements on the same line and resuming execution on the next line. This is similar to the effect of setting
⎕ELX←'→⎕LC+1' but without the problems doing so could cause in called functions.

The :FINALLY statement can appear as the last clause of a :TRY statement, following all :CATCH, :CATCHIF, and :CATCHALL clauses (if any). Code placed in a :FINALLY clause can be useful for cleaning up resources such as closing files and database connections but can also simplifies program logic in many other cases.

As with other control structures, an incomplete or incorrect TRY-CATCH structure will cause an OUTER SYNTAX ERROR.

Example:

Handling a Single Specific Anticipated Error

```
∇ ProcessAllComponents ftn;i;eof
  i←1
  eof←0
  :while ~eof
     :try
        ProcessOneComponent ⎕fread  ftn i
        i← i + 1
     :catchif 'FILE INDEX' ≡ 10↑⎕dm
        eof←1
     :end
  :end
∇
```

Here, a FILE INDEX error is anticipated. A FILE TIE ERROR, probably a consequence of a coding error is not anticipated. Keep in mind that exceptions directly raised in ProcessOneComponent will not be seen by this try-catch; they may be handled by a try-catch in the called function or by its ambient ⎕elx.

:TRACE

Syntax:

```
:TRACE
```

The :TRACE statement implicitly calls the function. The expression is conditionally executed if andonly if running in debug mode and its result value (if any) is passed as an argument to the LOG function for writing to the debug log file.

:WHILE :UNTIL

Syntax:

```
:WHILE expression1
      statement(s)
 :ENDWHILE


:WHILE expression1
      statement(s)
 :UNTIL expression2
```

The :WHILE control structure allows you to execute a statement or block of statements repeatedly, until *expression1* is false (0).

The :UNTIL statement allows you to test for specific conditions at the end of the control structure as well. If*expression2* is false (0), the system returns to the top of the control structure and evaluates *expression1* again. If *expression2* is true (1), the system exits the :WHILE control structure and

executes the line immediately following the control structure.

You can also use the :ANDIF and :ORIF statements with the :WHILE and :UNTIL statements to create compound control-expressions. For example:

:WHILE *expression1*

:ANDIF *expression2*

statement(s)
:UNTIL *expression3*

:ORIF *expression*4

:ORIF *expression5*

You can substitute :END for :ENDWHILE.

:VERIFY

Syntax:

:VERIFY *condition expression*

The :VERIFY statement behaves the same as the :assert statement except that it is always executed, regardless of whether debug mode is ON or OFF. This provides a way to execute a condition and verify itsresult always, rather than only when debug mode is ON.

:REGION :ENDREGION

Syntax:

:REGION
    statement(s)
:ENDREGION

You can use :region and :endregion to define the bounds of collapsible regions. The :region and :endregion keywords can have an argument which is free text that is not executed. It is simply used as a label for the region both when the region is expanded and collapsed.

:PASSTHRU

Syntax:

:PASSTHRU *myvar*

The :PASSTHRU statement allows user defined (global) variables to be passed through into function local variables in a manner like how passthrough localization works for quad variables.

The global value of variables that are declared as :PASSTHRU is inherited rather than shadowed by the function's localization of the variable as if they were not localized. But modifications made to the variable inside the context of the function are strictly local to the function and don't pass back to the global context when the function exits. **Note**: The user defined variable name must be localized.

:PUBLIC

Syntax:

:PUBLIC *expression*

An APL64 public function can be defined via the :PUBLIC control structure prefix. APL64 public functions are those APL64 user-defined functions which will be exposed to a user of a cross-platform component. APL64 user-defined functions which are not specified by the APL64 programmer as public will not be exposed to the user of a cross-platform component.

Refer to the menu Help | Developer Version GUI | Create Cross-Platform Component for additional information on the :PUBLIC control structure prefix.

Inline Control Sequences

Inline control sequences (ICS) can articulate arbitrarily complex logical calculations with a concise and efficient notation, which uses colon-prefixed keywords :and, :or, :then, :else, and :choose embedded in expressions. It uses Progressive Partial Evaluation (PPE) similar to the &&, ||, and ? operators found in C, C++, C#, Java, JavaScript, etc. ICS expressions are an inversion of traditional control structures. They may contain cascading decisions and execution alternatives in a single statement, whereas traditional control structures involve multiple expressions spread across multiple statements typically on multiple lines. PPE is nothing new to APL. The :IF control statement uses PPE to jump into the True or False clause as soon as the overall outcome can be deduced from partial results. For example:

```
:IF Test 1
:ANDIF Test 2
:ANDIF Test 3
   True
:ELSE
   False
:ENDIF
```

If any of the test conditions are false the remaining tests are skipped and the False clause is executed immediately. If a test condition is true, the next test must be evaluated to determine the outcome of the calculation. The True clause is only executed if all tests are true. Step by step we have: if Test 1 is false, Test 2 and Test 3 are skipped and the False clause is executed. If Test 1 is true, Test 2 is evaluated. If it is false, Test 3 is skipped and the False clause is executed. If true, Test 3 is evaluated. The True clause is executed only if Test 3 is also true. Otherwise, the False clause is executed.

Similarly, when the :IF statement is followed by a series of :ORIF tests, we skip the remaining tests and begin execution of the True clause as soon as any of the test conditions are True. We only continue evaluation of tests and eventually execute the False clause if all previous test conditions are false.

PPE is a powerful tool for simplifying program logic. For example, the first test in a series might determine if a function's left argument has a value (i.e., whether called monadically or dyadically) and a second test might check its rank or shape. If the argument doesn't have a value we discontinue the test

sequence early since we cannot check its rank or shape without a value. By guarding later tests with previous tests that pre-screen certain conditions (such as existence of a value) we can efficiently and concisely test the conditions we want with a minimum of coding and a maximum of clarity.

The :IF → :ANDIF logical progression shown above using five statements can be stated more concisely using one ICS expression like this:

```
(Test 1 :AND Test 2 :AND Test 3 :THEN True :ELSE False)
```

The order of progressive evaluation above might not feel right to everyone at first glance. APL executes from right to left and this might feel like it's going the wrong way. While there isn't enough space in this paper to fully discuss the decisions that went into ICS design the following summary is offered:

- APL executes from right to left but is conceptualized, written, and read from left to right. We usually conceive of and type the left part before the right and always read it that way.
- Conditional expressions are usually typed with the condition to the left of the objective:

  →(Condition)/L1 ⍝ for branching
  ⍎ (Condition)/'Expression' ⍝ for execute
  ⎕ERROR(Condition)/'ERROR MESSAGE' ⍝ for signaling errors

  This is a good thing because we tend to think of the condition before we think of the objective, or at least we think about how to express the condition before the objective. Therefore, this ordering minimizes the number of superficial left and right key movements required to type the expression.
- Progressive logical decisions are generally conceptualized in their order of dependency. For instance, we need to know if a left argument exists before we test its rank or shape. So, we tend to think of existence testing before rank or shape testing.
- The left to right order of ICS progressions is reflective of this natural thought order.
- If ICS expressions were evaluated right to left like this:

  ```
  (True :ELSE False :WHEN Test 3 :AND Test 2 :AND Test 1)
  ```

  we would have to type the tests in the reverse order from how we would naturally think of them.
- If we think of ICS expressions as being like diamonds with decisional powers, then it is easier to understand why the left to right ordering makes sense. They follow the same flow pattern as diamond separated statements and are consistent with a pattern of programming that was common in pre-control-structure days when hierarchical conditions would often be coded like this:

  ```
  →(Condition1)/⎕LC+1 ⋄ →(Condition2)/⎕LC+1 ⋄ ObjectStatement
  ```

- The keywords in an ICS expression are not functions or operators. They are conjunctive in nature and exist in a different semantic space than the functional sub-expressions connected by them.
- It can be useful to think of ICS keywords as being like parentheses with decisional powers. ICS

keywords morph normal order of execution in a similar way and exist in a similar semantic space.

The following list shows all recognized forms of ICS expression (for brevity the :OR form of each expression is not shown; wherever :AND appears it can be replaced by :OR):

```
( Cond  :THEN   Case1 :ELSE Case0 )
( Cond  :THEN   Case1 )
( Cond  :ELSE   Case0 )
( Cond1 :AND    Cond2 )
( Cond1 :AND    Cond2 ... :AND  CondN )
( Cond1 :AND    Cond2 ... :THEN Case1 :ELSE Case0 )
( Cond1 :AND    Cond2 ... :THEN Case1 )
( Cond1 :AND    Cond2 ... :ELSE Case0 )
( Index :CHOOSE Case1 : Case2 : ... : CaseN :ELSE Else )
( Index :CHOOSE Case1 : Case2 : ... : CaseN )
```

Parentheses around ICS expressions are only necessary to the extent needed to denote logical groupings, alter progression evaluation order, or disambiguate cases where the colon-prefix of the first keyword would otherwise be interpreted as declaring the name to its left as a statement label.

Any statement beginning with a colon-space (or any other character that is not part of an identifier immediately following the colon) tells the interpreter to not look for labels in that statement. A colon-space at the beginning of any statement in a user defined function also suppresses the output for the :THEN expression as well as for any line of execution. Either of the following statements may be used to prevent Condition from being interpreted as a label:

```
( Condition :THEN Action )
: Condition :THEN Action
```

The :AND and :OR keywords can be used in sequences of any length. However, they cannot be mixed in the same expression unless parentheses are used to group them like this:

```
( Cond1 :AND Cond2 ) :OR ( Cond3 :AND Cond4 :AND Cond5 )
```

The :THEN and :ELSE keywords offer a binary choice between two alternative expressions. Only one of the expressions is executed. These keyword can be used together or alone if only one choice is needed, with the implicit other choice being no execution and no result value. If both are coded :THEN must come before :ELSE. In contexts that require a result, both clauses are required.

The :CHOOSE keyword selects one expression from a list of choices separated by colons, which must be followed by a space or other character that doesn't begin a name. The value to the left of :CHOOSE must be a singleton integer. It is used as a ⎕IO sensitive index to select one of the cases to execute. An optional :ELSE clause can be executed if there isn't a case corresponding to the index value. An error is not signaled if the index does not select a case unless the expression is used in a context that requires a value, in which case an INDEX ERROR is signaled.

The following :CHOOSE expression is roughly equivalent to the :SELECT statement below it:

```
( Index :CHOOSE CaseA : CaseB : CaseC :ELSE ElseCase )

:SELECT Index
:CASE ⎕IO
   CaseA
 :CASE ⎕IO+1
   CaseB
 :CASE ⎕IO+2
   CaseC
 :ELSE
   ElseCase
:ENDSELECT
```

Except :SELECT cannot be used to produce an argument value in the way :CHOOSE can:

```
X Foo ( Index :CHOOSE CaseA : CaseB : CaseC :ELSE ElseCase )
```