Programmer-Defined APL Functions

Table of Contents

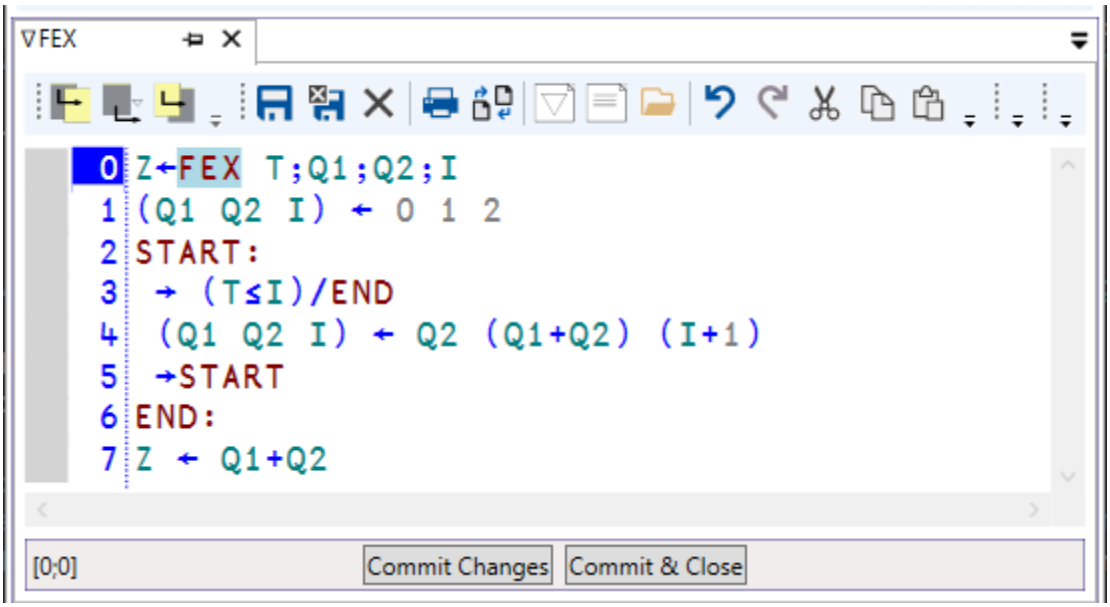
Overview

An APL64 Developer version user may create programmer-defined function to perform a task repeatedly, potentially with different input data, without having to retype the APL executable statements that are needed for your task.

An APL64 programmer-defined function can be used like an APL64 primitive function or system function. The APL64 programmer-defined function can be evaluated in immediate execution mode, combined other APL functions in an executable statement or called from within another programmer-defined function. If the APL64 programmer-defined function has an explicit result, the result can be displayed in the history, used in an executable statement or assigned to an APL64 variable.

As an example, this chapter uses a programmer-defined function with name FEX:

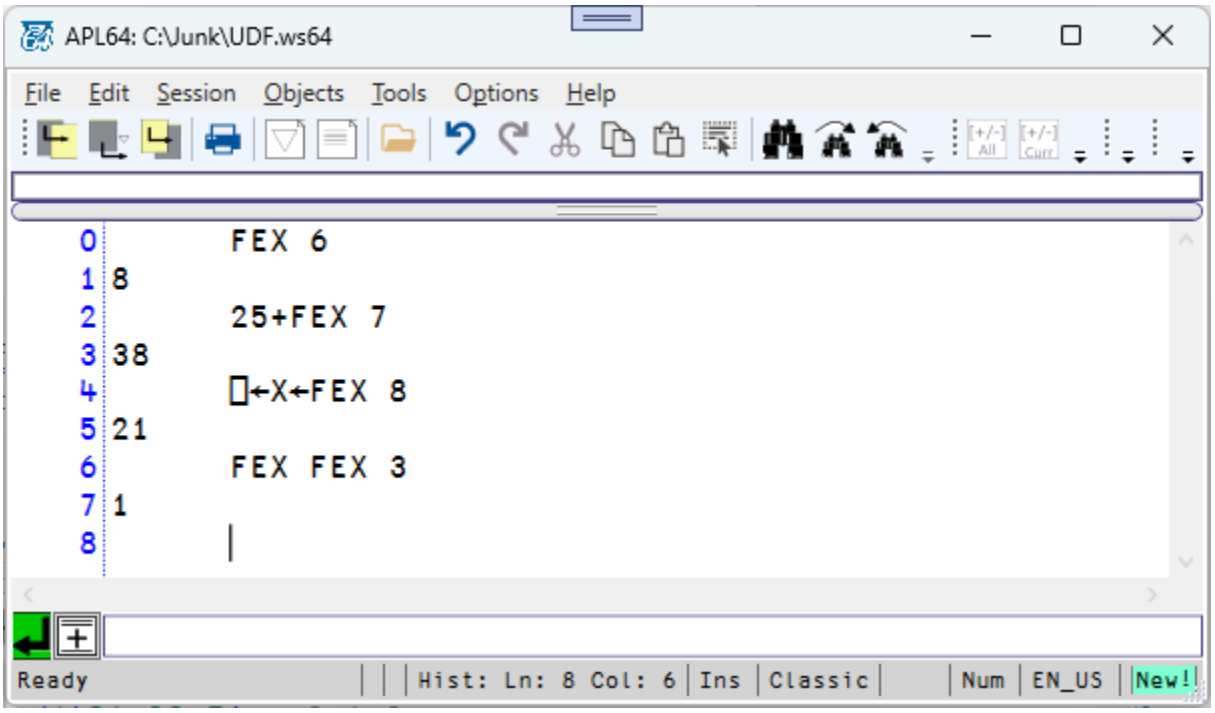
```
Z←FEX T;Q1;Q2;I
(Q1 Q2 I) ← 0 1 2
START:
 → (T≤I)/END
 (Q1 Q2 I) ← Q2 (Q1+Q2) (I+1)
 → START
END:
Z ← Q1+Q2
```



The FEX function is monadic and returns an explicit result. These APL executable statements using FEX are valid:

```
FEX 6
25+FEX 7
```

```
⎕←X←FEX 8
FEX FEX 3
```



When evaluated, a programmer-defined function can have an effect on the current workspace state, even if the function has no explicit result, because the function statements represent an algorithm which process information within the current workspace.

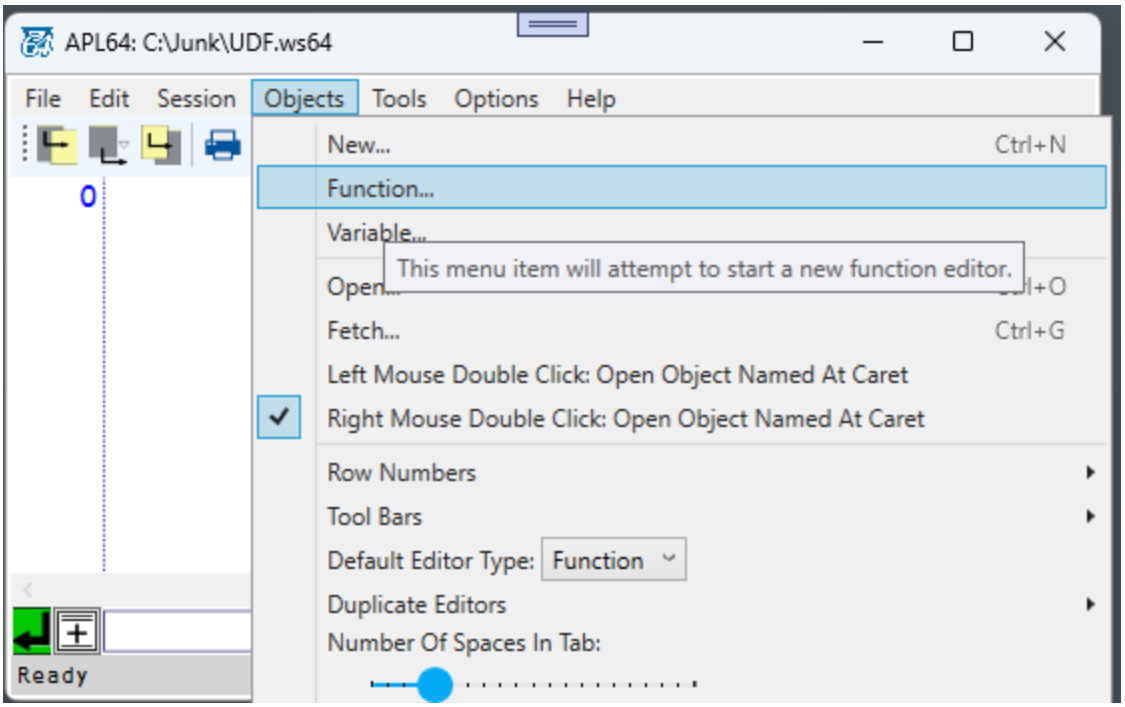
Basic Concepts

Function Header

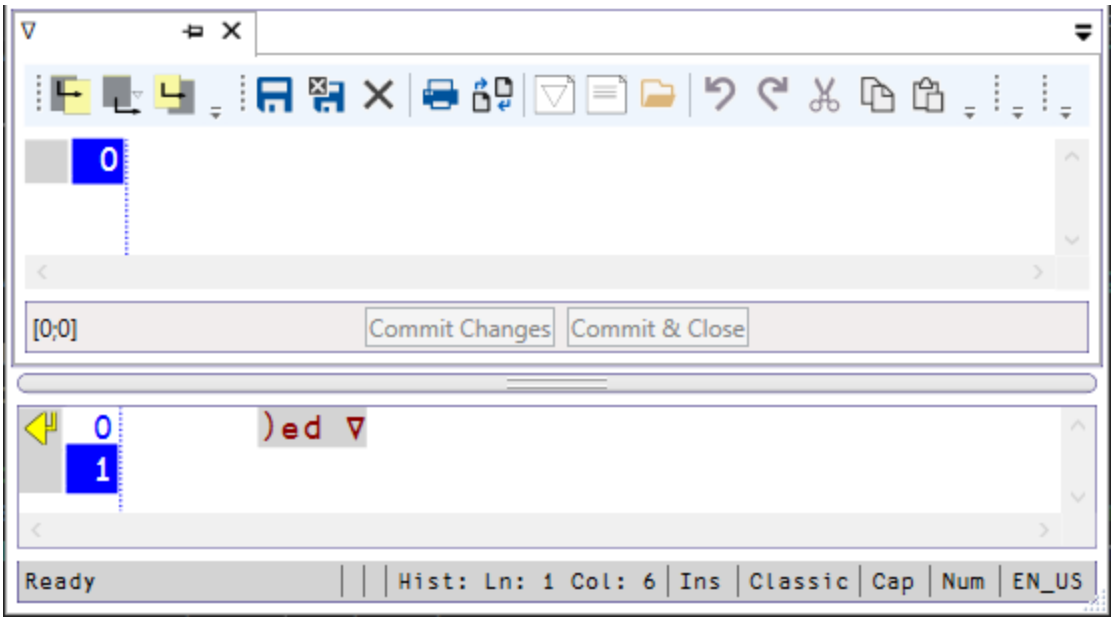
The function header specifies the function name, an optional result and optional left and right arguments. The function header is line number zero in the function.

Present a Function Editor

To create the example function FEX, in the APL64 Developer version click the **Objects | Function** menu item:



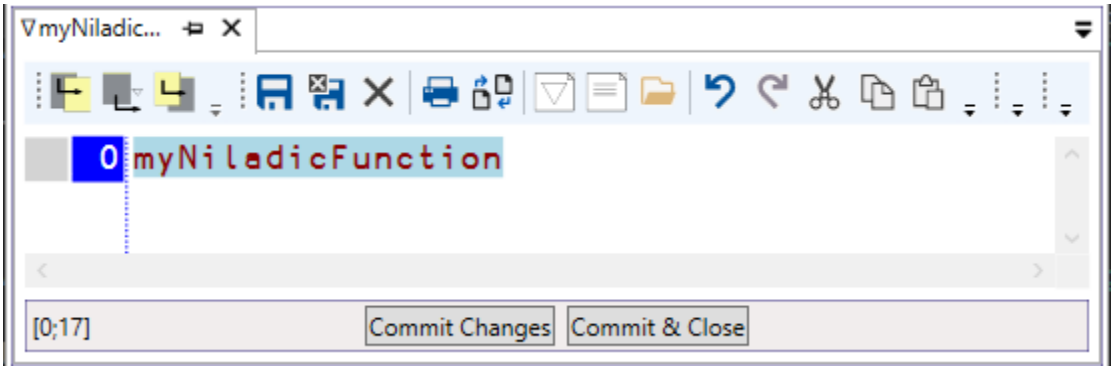
A function editor will be presented and the keyboard focus will be on line zero of the function editor:



The function header is a function line that the system uses to obtain the optional result variable name, the required function name and the optional left and right argument variable names.

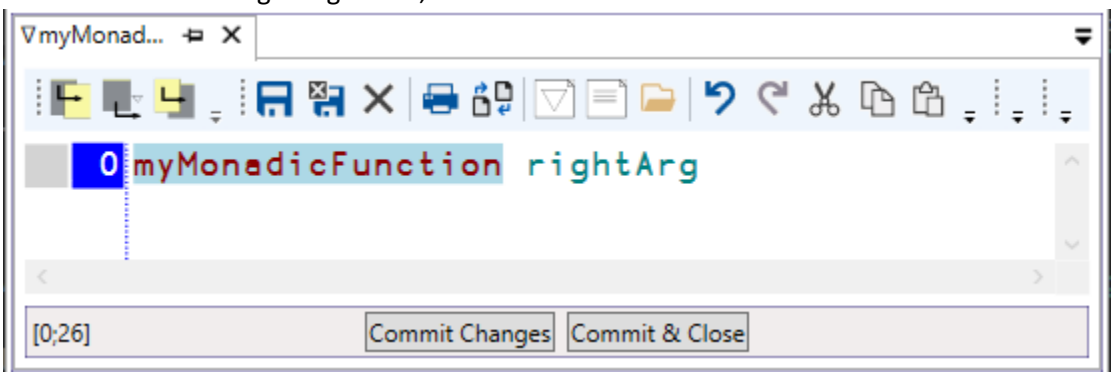
Niladic Function

The minimum content of a function header line is the function name, in which case the function is niladic without a result:



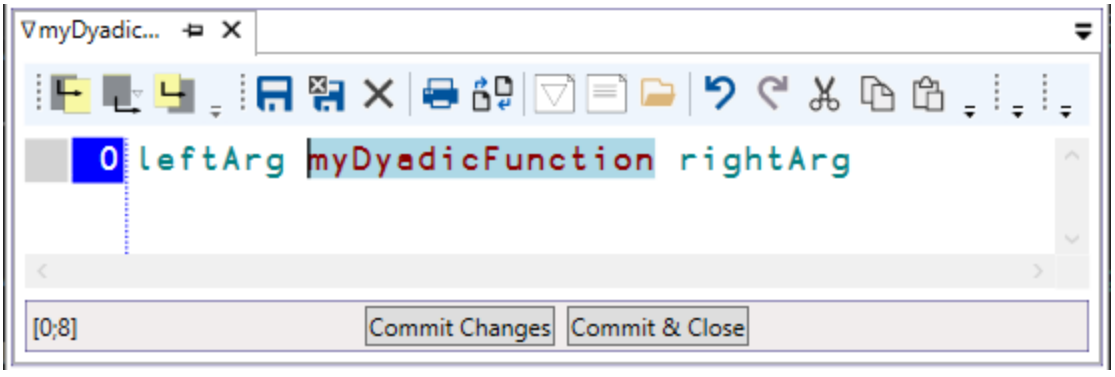
Monadic Function

If the function header has two names, separated by a space, the first name is the function name and the second name is the right argument, in which case the function is monadic without a result:



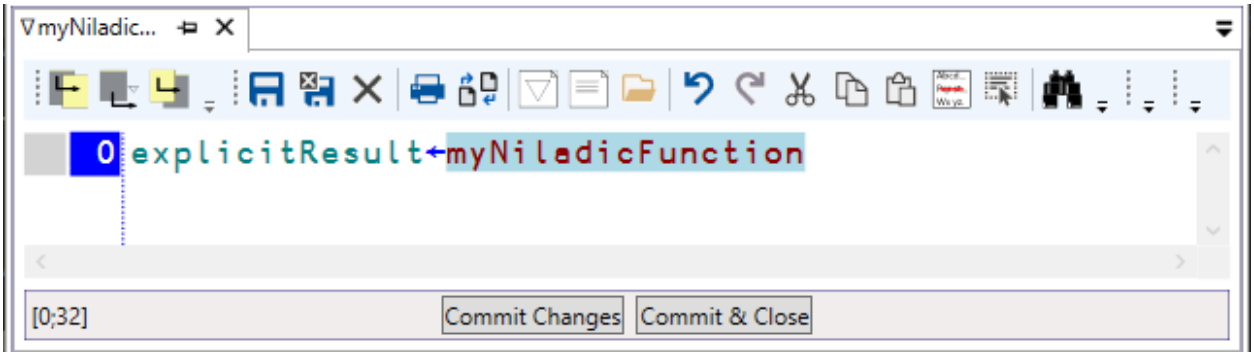
Dyadic Function

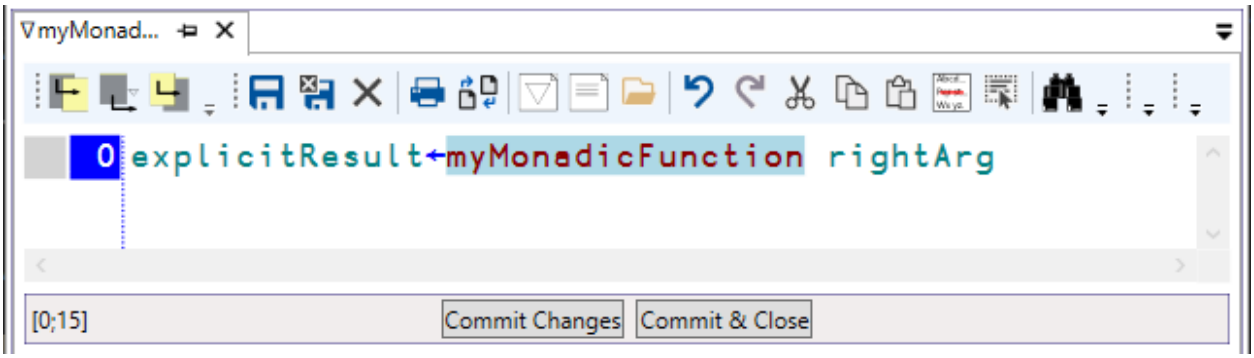
If the function header has three names, separated by spaces, the first name is the left argument, the second name is the function name and the third name is the right argument, in which case the function is dyadic without a result:

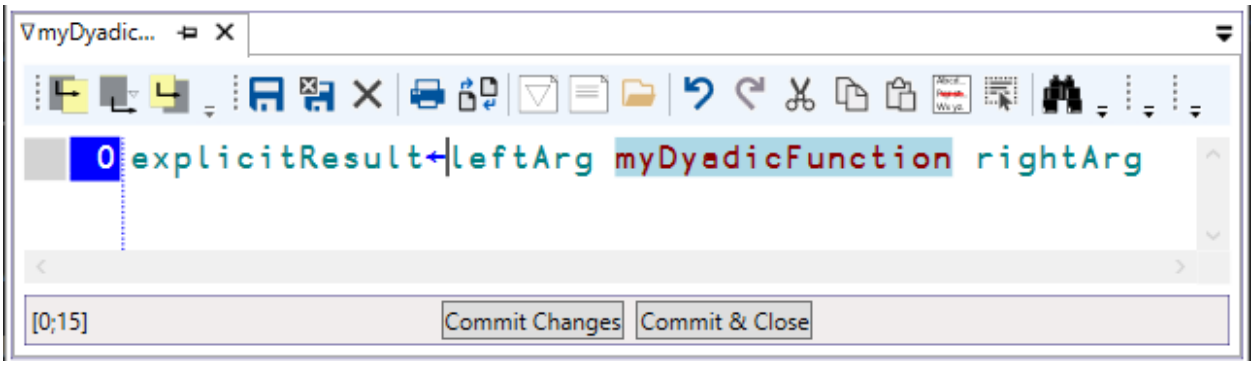


Function with Explicit Result

If the function header begins with a name followed by the assignment glyph (←), the function has an explicit result. A niladic, monadic or dyadic function can have an explicit result:



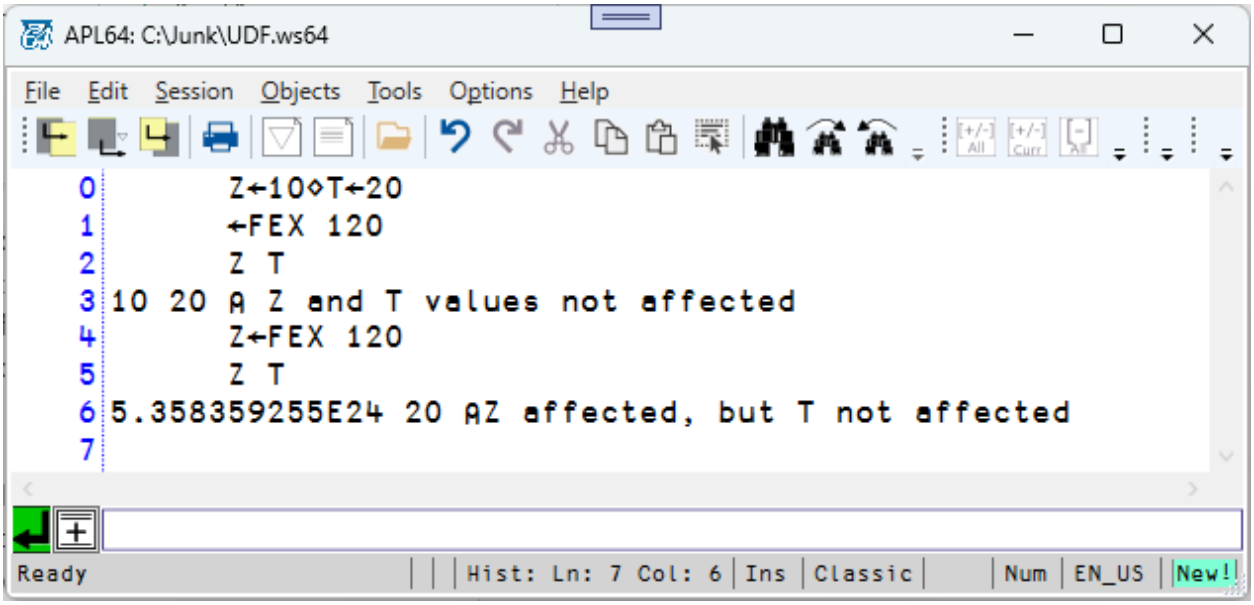




Functions as Black Boxes

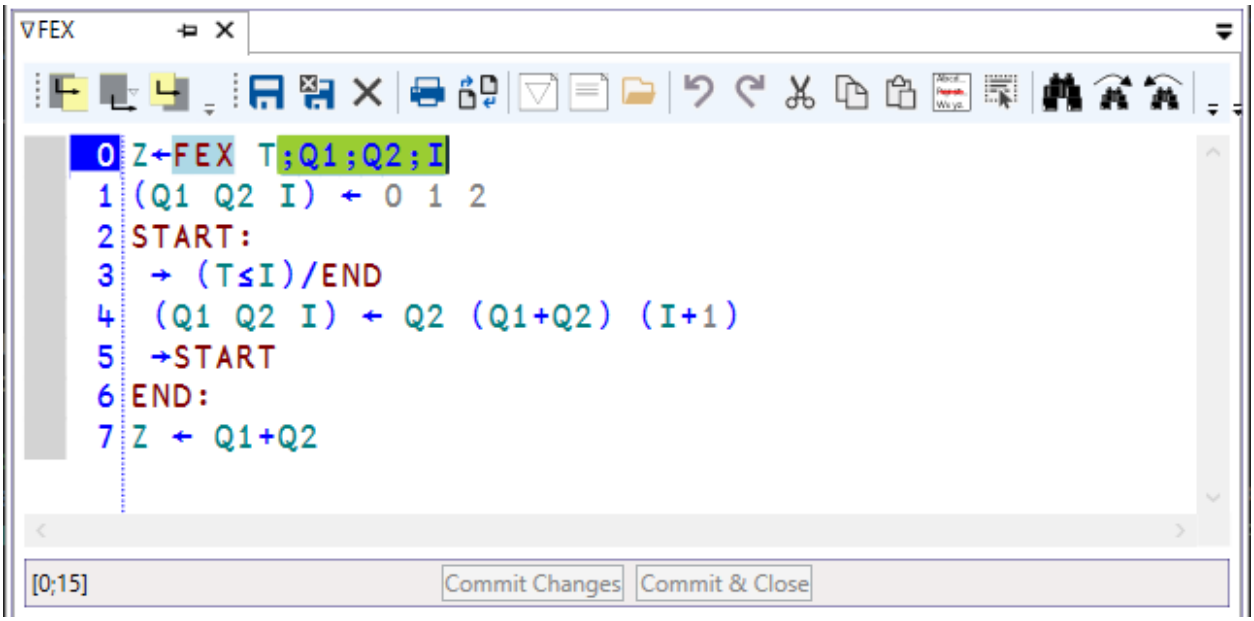
An important concept in APL programming is that the processing inside a function occurs in a protected environment. The system uses any variable that appears in the header of a function only within that function. When it finishes evaluating the function, it erases the variable without affecting anything else in the workspace. If a variable with the same name is defined in the workspace, the processing within the function does not affect it if that variable is also defined within the function.

The system accomplishes this with a process called localization. If variables named Z and T exist in the workspace and the function has the header Z←FEX T, APL64 stores the workspace values of Z and T temporarily. While APL64 evaluates the function, APL64 uses the input value of T specified in the header as the local value of T and the result value of Z within the function. When the function evaluation is completed, the local variables defined within the function are erased and the result prepared by the function, if any, may be assigned to a workspace variable, depending on the syntax used when the function execution was started:



Local and Global Variables

Variables can be assigned within a function. Specify a variable as local to the function by separating its name at the end of the function header line with a semi-color (;) or space, so that it will be used only for intermediate results while the function is evaluated. A local variables in a function does not affect the same named variable in the workspace. In the definition of the example FEX function, the local variables are Q1, Q2 and I:



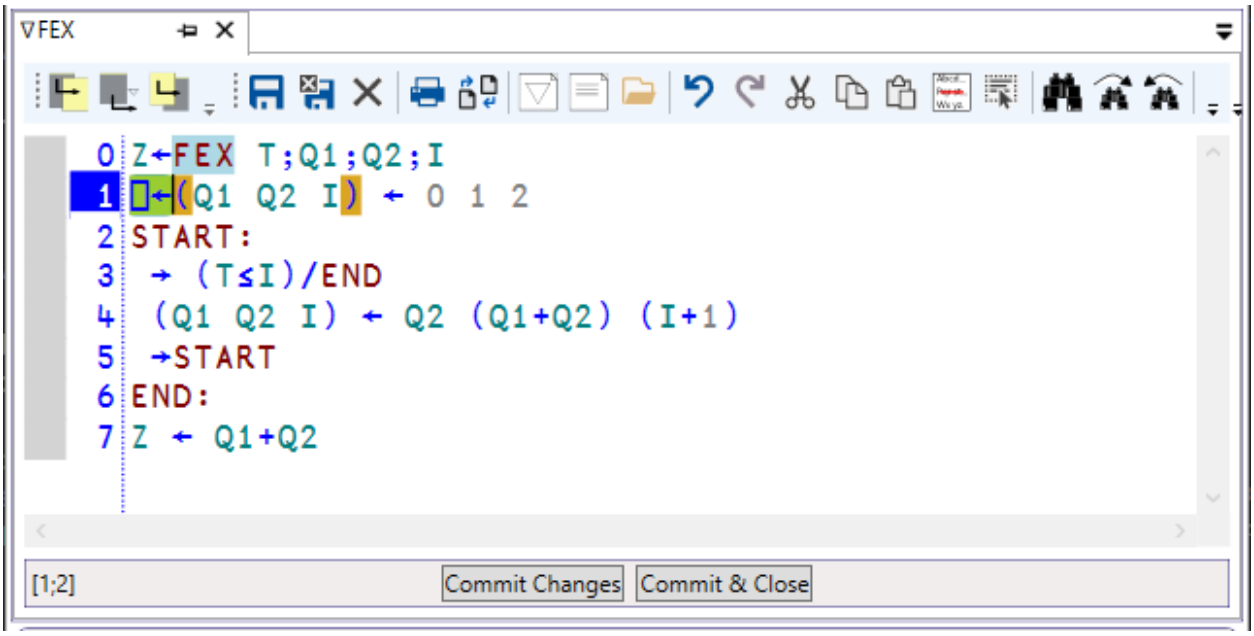
A variable defined in a function that is not specified as local in the function header, the value of that variable will be retained in the workspace when the evaluation of the function is completed.

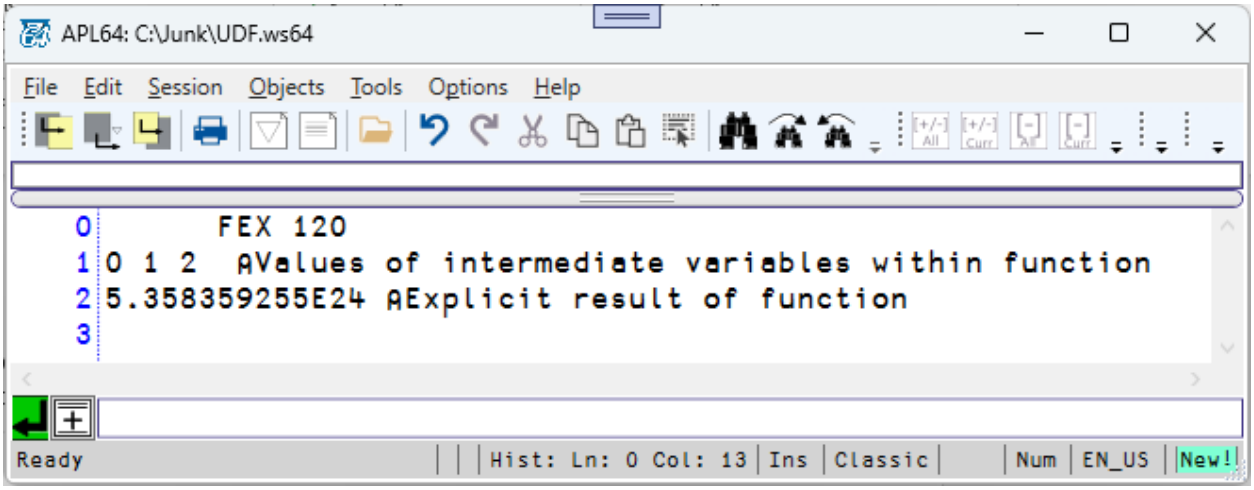
Typically, variables like counters, I in the FEX function, or intermediate results, Q1 and Q1 in the FEX function are specified as local variables in the function header, because they are only used within the function and so that their values do not remain in the workspace when function evaluation is completed.

Assignment and Implicit Output Statements

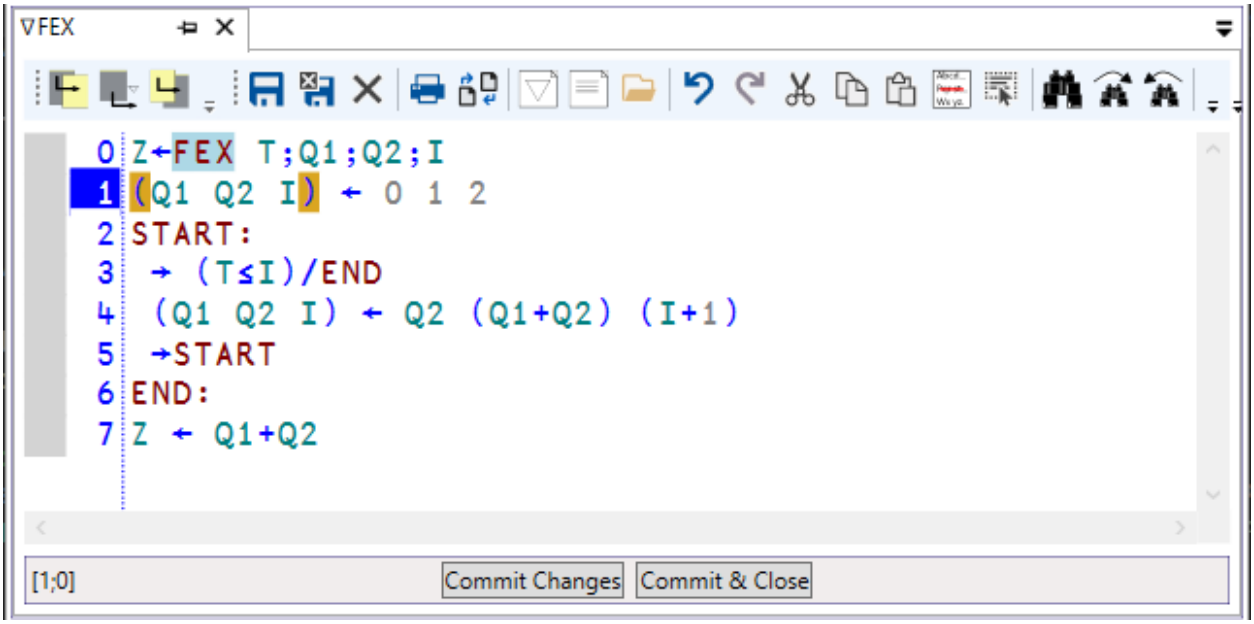
Some statements in a function are simply APL expressions, which use number constants, variable values, APL primitive symbols, and other functions to manipulate numeric and character data in arrays. When the system executes the function, it executes these statements in sequence.

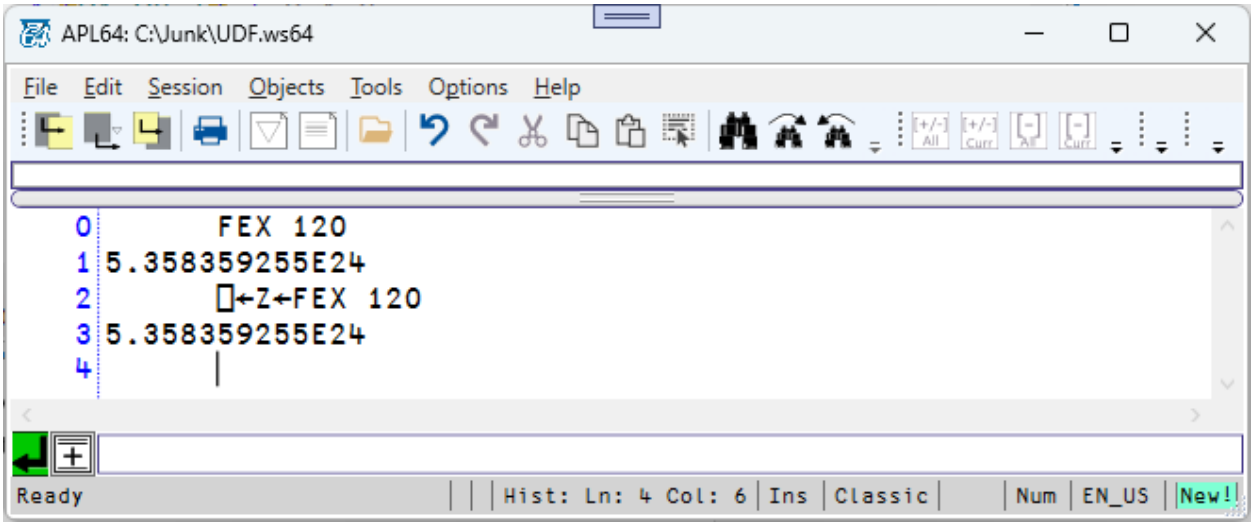
When an intermediate result in the function is calculated, its value can be assigned to a local or global variable using the assignment glyph (←), so that its value is not displayed in the history. If an intermediate result is not assigned to a local or global variable, its value will be displayed in the history when the function is evaluated. To both assign an intermediate result and see it in the history, use the quad function:





When a function with an explicit result is evaluated the result value is displayed in the history unless the result is assigned. The result may be assigned and displayed in the history using the quad function:





Permitted Statements in a Function

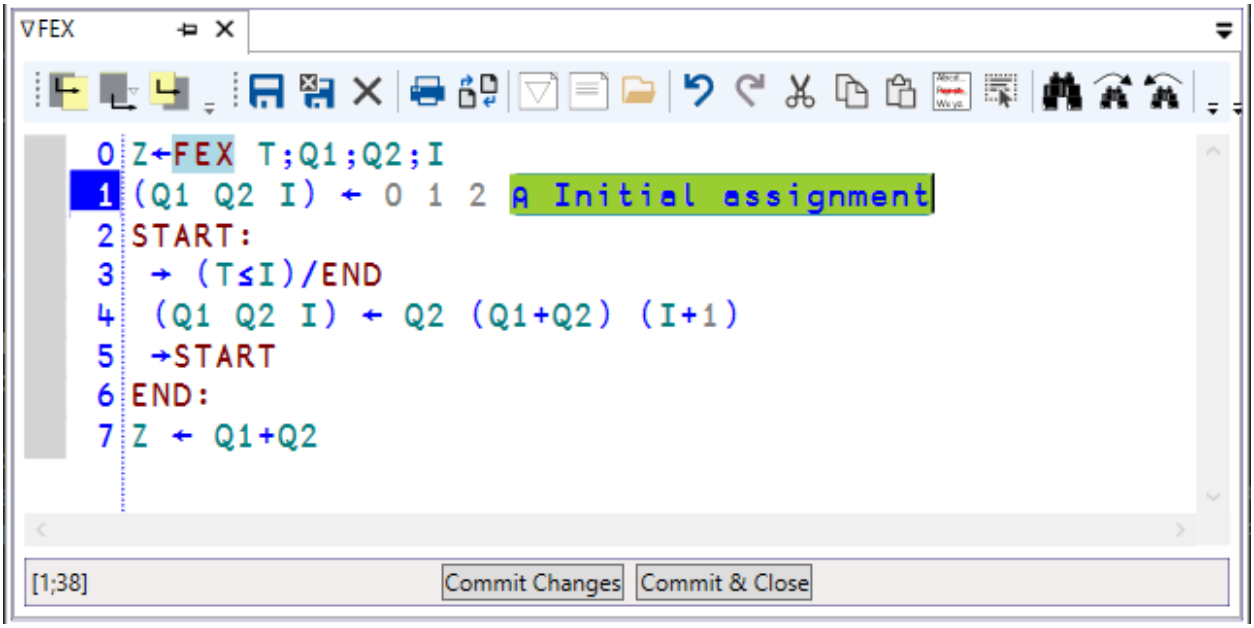
Except for system and user commands, e.g.)… or]…, all other APL statements may be used directly in a function. Many system commands, e.g.)COPY, have system function analogues, e.g. ,⎕COPY. User command can be indirectly used as arguments to the ⎕UCMD system function.

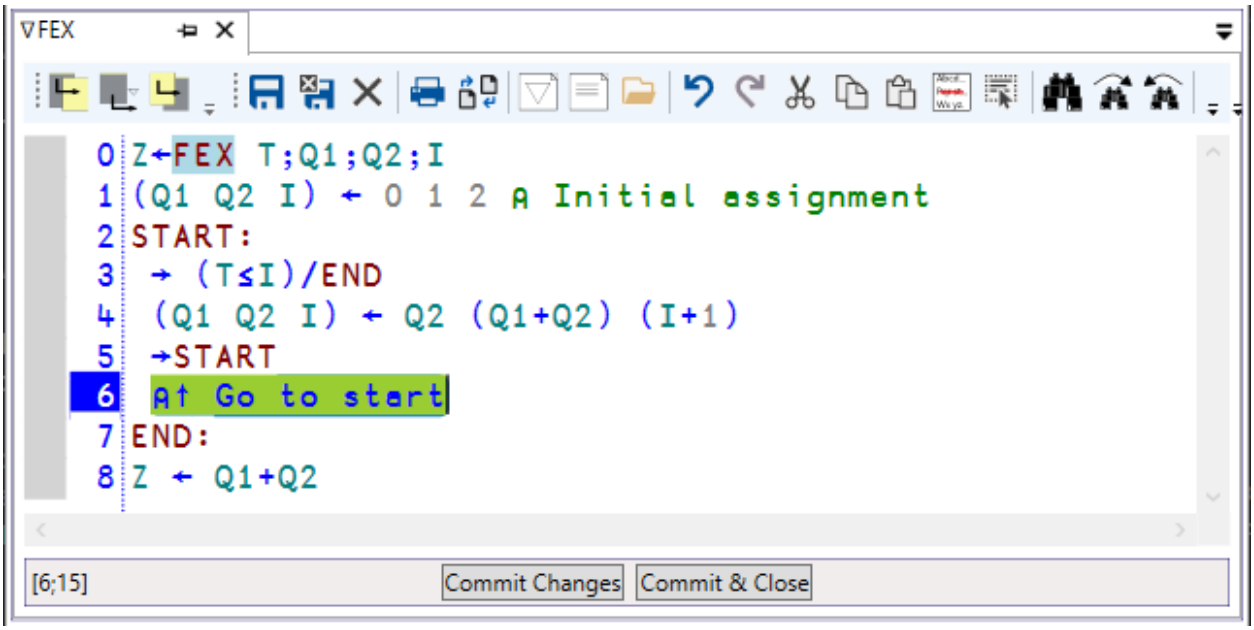
Using Statement Separators

Multiple statements may be placed on the same line of the function separating them with the APL diamond glyph (⋄) and executing the statements from left to right starting to the left of the first diamond glyphs on the function line, e.g. stmt1⋄stmt2⋄…

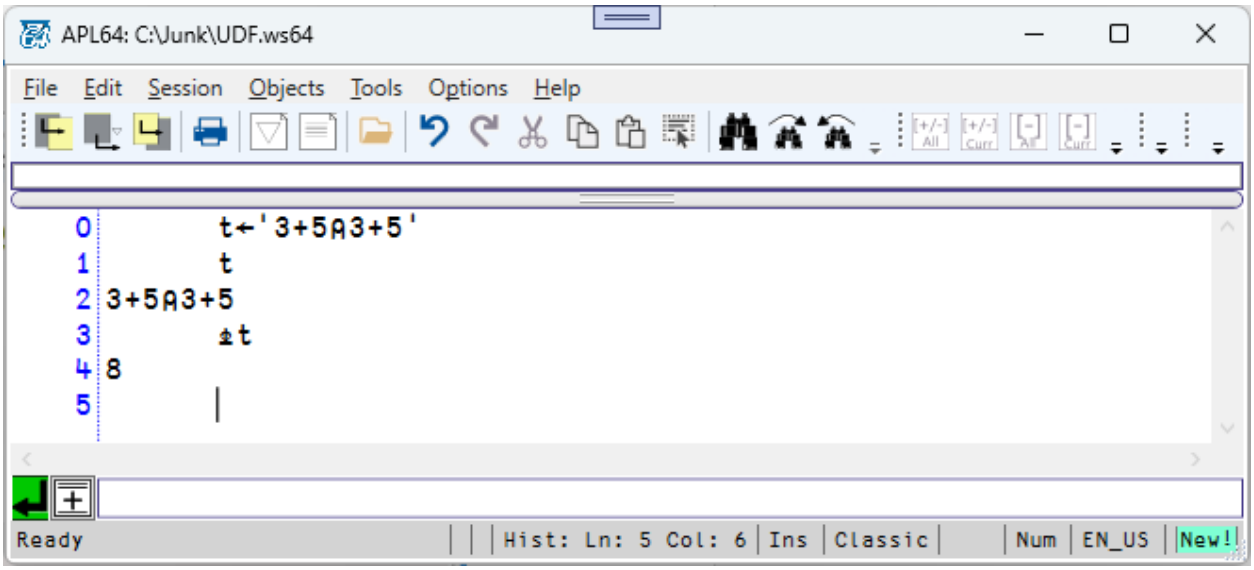
Comment Statements

Function lines can be comment statements of can contain suffixed comment statements. A comment statement begins with the APL lamp glyph (⍝). Any text which comes after the APL lamp glyph on the function line is a comment which is not executed.





A comment within a text vector enclosed in single or double quotes, is treated as part of the text vector and not a comment unless the text vector itself is executed:



Branch Statement

A branch statement can alter the execution sequence of a function when it is evaluated. A branch statement begins with the APL branch glyph (→) followed by:

- A line number, e .g. →23
- A line label, e.g. →Lable1
- An APL executable statement which results in a line#, line label or zilde (⍬), e.g. →(V=10)/Lable1
- Nothing after the branch glyph

If the branch points to a destination which is line number zero or a function line which does not exist, the function evaluation ends, i.e. the function is exited.

If the branch occurs, the execution order of the APL statements in the function does not continue past the branch statement, but instead starts from specified line number or line label.

The branch statement is placed on the function line where the conditional branch should occur.
A line label is programmer-selected contiguous text followed by the colon glyph (:) placed at the beginning of the specified line. Using a line label rather than a line number avoids potential problems is function lines are subsequently inserted into the function. The line label is placed on the function line which is the destination of the branch.

Zilde is shorthand for the numeric value 0⍴0. If the executable statement after the branch glyph results in zilde, no branch occurs when the function is evaluated.

If the expression to the right of the branch arrow results in a vector, the system uses only the first element of the vector, e.g. →label1,45,label2 will branch to label1 without consideration of line number 45 or label2.

The branch statement can be used to select from multiple destinations in the function, e.g.
However, you can use a vector in the expression and calculate an index to that vector. This allows you to branch to different line labels as easily as to different line numbers. For example: →K⊃Lable1 Lable2 will branch to Lable1 or Lable2 depending on the value of K and the current value of the index origin (⎕IO).

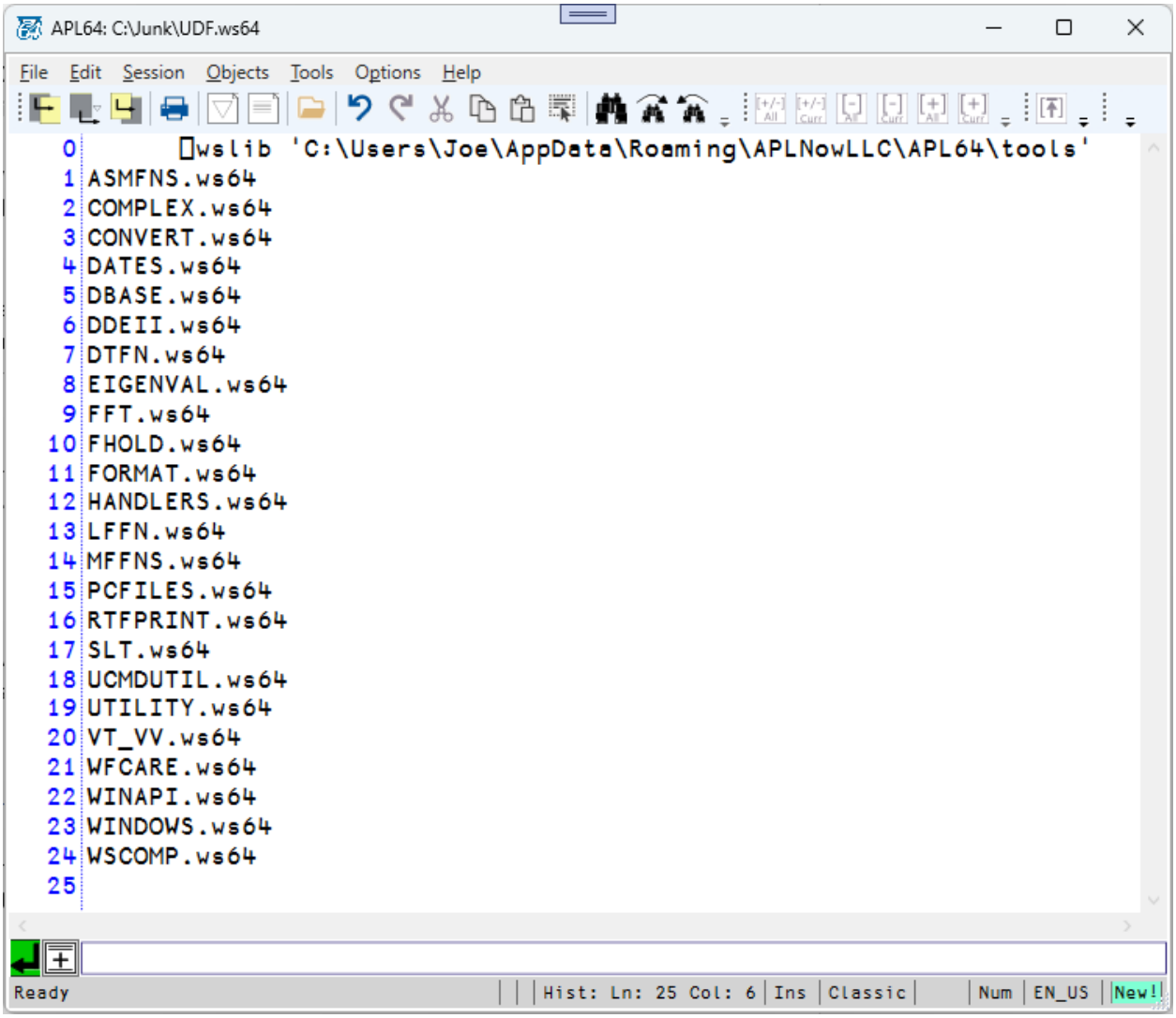
If the branch has no APL statement after the branch glyph, a naked branch, all execution stops and the system returns to immediate execution mode. If another function calls the function that uses the naked branch, you cannot restart the calling function at the point of interruption.

Control Structures

Control structures are APL statements which change the execution order of statements in a function. Control structures include key word APL statements such as :FOR … :ENDFOR, :IF … :ENDIF.

TOOLS WORKSPACES AND FUNCTIONS

When the APL64 developer version is installed, the Tools folder is deployed and contains several workspaces each containing APL programmer-defined functions. If the default installation path is used, find the Tools folder here: C:\Users**YourUserName**\AppData\Roaming\APLNowLLC\APL64\tools.



For example, the DATES.ws64 workspace contains these programmer-defined functions: