

Details of APL64 Project Update May 2019

This report provides detailed information pertaining to the progress of the APL64 Project in the following areas:

- APL64 Project team's approach to evaluating performance
- Operations in which the APL64 Project system outperforms APL+Win
- Improvements and enhancements in the modernized APL64 Project Programmer Session

APL64 Project Performance Tests

In most cases tested, the APL64 Project is as fast or in some instances, substantially faster than APL+Win. The APL64 Project development team was able to take advantage of the state-of-the-art development tools and superior methodology used in the APL64 Project to perform optimizations. In every case where optimization was applied, performance increased such that the APL64 Project speed was as fast or faster than APL+Win. It is our goal to optimize all the APL64 Project primitives to ensure that the performance of the APL64 Project will match or be consistently faster than APL+Win.

The optimization process is on-going. Our experience suggests we can successfully optimize all primitives to make the APL64 Project run significantly faster than APL+Win. At this time we are sharing with you the results of some APL64 Project performance tests. All of the results shown in the charts that follow are for cases where optimized APL64 Project operations run faster than APL+Win. Un-optimized cases are not shown in this report. APL2000 will provide updates on the performance of these cases as additional optimizations are completed.

In the accompanying charts, average elapsed execution time is always shown on the vertical axis in milliseconds. Lower average elapsed time values correspond to faster performance. The "winner" of each comparison is always marked with a yellow star to emphasize the better performing system, with a lower average elapsed time value. Remember in reviewing these charts that "Lower Time is Better".

Argument size is shown on the horizontal axis. In general, this indicates the number of elements in argument arrays. In a few cases this axis indicates the number of elements in the result (for example, the deal primitive, `n?m`, uses scalar values for both arguments but the result size is controlled by the value of the left argument). In all cases the horizontal axis value indicates how much work was done for the operation. As the amount of work (i.e., number of elements) increases, the corresponding time also increases for both APL+Win and the APL64 Project. In most cases there is a linear relationship between the amount of work being done and the execution time. This linear relationship between argument size and execution time is highlighted by a line of best fit on each chart.

However, in a few cases note that APL+Win and the APL64 Project values converge to roughly the same value as the argument size increases and the differences seem to be due to statistical variations. In such cases, the critical factor limiting performance is memory access time. Memory access time is workstation dependent and relatively independent of the APL implementation. The speed observed on different workstations for such cases will be heavily influenced by the amount of L1, L2, and L3 memory cache on the machine. These are hardware limitations that constrain the speed that data can be moved in memory and would be difficult or impossible to transcend for algorithms implemented in any computer language whether compiled or interpreted.

The chart showing **FLOAT+FLOAT** operation is a good example of the effect of memory access time for large argument sizes. Up to about 75,000 elements the APL64 Project is faster than APL+Win, but by 100,000 elements in the argument sizes the systems run at roughly the same speed because the time it takes to access memory has become the dominate factor constraining calculation throughput.

Performance results are calculated by executing a test expression over thousands of iterations and averaging the execution time. This is done to reduce statistical variations and to make sure “Garbage Collection” costs are also factored into the run times. The performance test workspace executes the expressions for varying size argument arrays of different types. For example, for the **INT+INT** operation, integer vectors are added together. Argument arrays are calculated using the following data values:

```
float ← count ρ 0.08296339080035192 2.570828471411902 0.8041284604922564
Δfloat ← ϕ float
floatScalar ← ⍵first float
int ← count ρ 2 3 4
Δint ← ϕ int
intScalar ← ⍵first int
bool ← count ρ 1 0
Δbool ← ϕ bool
ones ← count ρ 1
Δones ← count ρ 1
zeros ← count ρ 0
Δzeros ← count ρ 0
```

The variable **count** in the expressions above varies in a **FOR** loop that repeats each test over the following set of argument sizes: **1 2 5 7 10 25 50 75 100 250 500 750 1000 2500 5000 7500 10000 25000 50000 75000 100000**.

When performance is measured for dyadic scalar primitives such as **INT+INT** where both arguments have the same shape and data type, the left argument uses the data variable with the value matching the data type (**int** for integer data, **float** for floating point data, etc.) and the right argument uses the data variable with a **Δ** prefix (**Δint** for integer data, **Δfloat** for floating point data, etc.). This is done so that the calculation is accessing data at different memory addresses for the left and right arguments. This is important so that memory caching effects on performance are properly taken into consideration. While the chart for adding two integer arrays is labeled as **Operation: INT+INT** that calculation is done using the expression:

```
int + Δint
```

For operations that mix data type such as **INT+FLOAT** we execute the non-delta variable names such as:

```
int + float
```

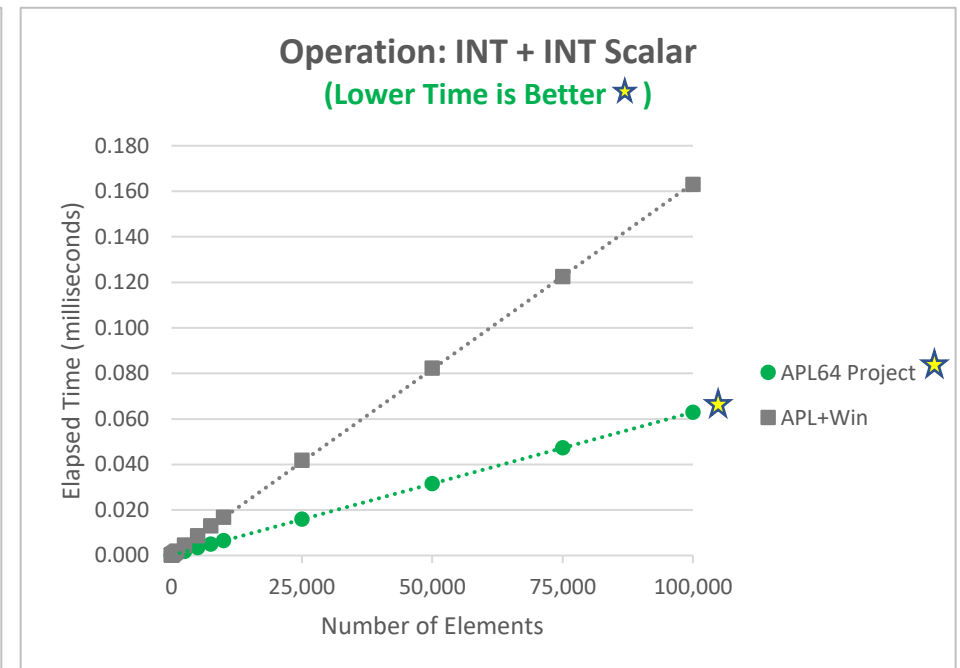
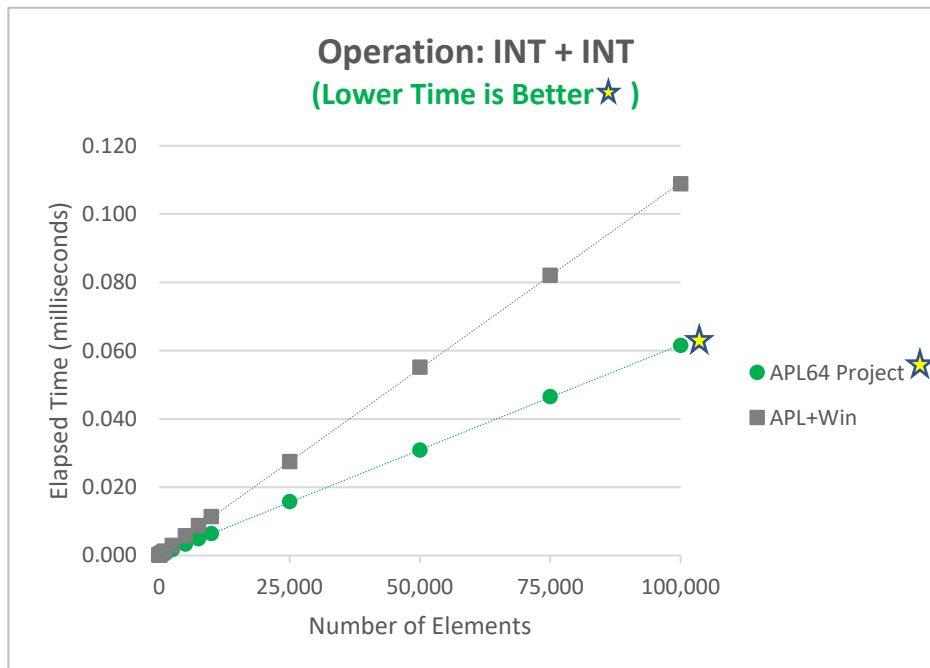
Operations that indicate one of the arguments is a scalar use the corresponding **Scalar** suffixed variable name. For example, the chart for **Operation: INT+INT Scalar** is computed using expression:

int + intScalar

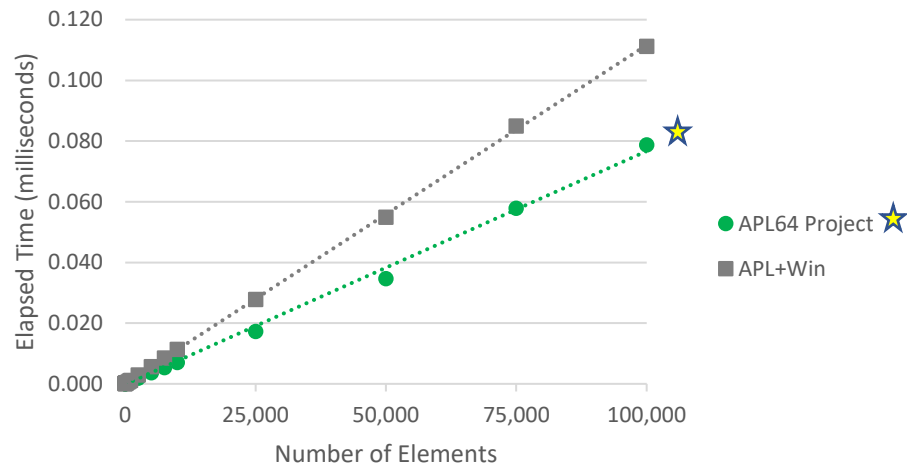
The optimizations achieved in the APL64 Project are largely dependent on the efforts of the APL64 Project team members to utilize the significant improvements in the tools and techniques available in the development environment of the APL64 Project.

Thus far Optimized APL64 Project vs. Production APL+Win Performance results

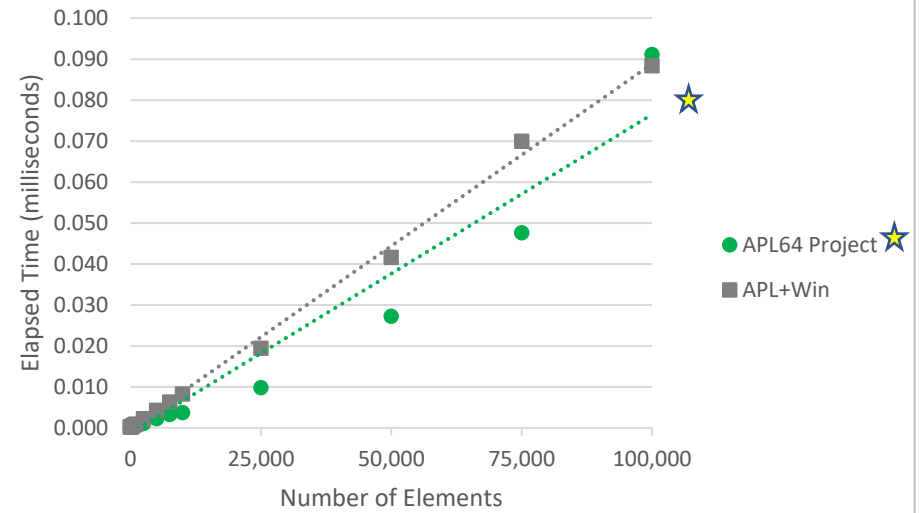
The charts below are representative of system performance for some of the APL primitives that have been optimized in the APL64 Project. They illustrate performance of the APL64 Project relative to APL+Win version 19. APL+Win data points use gray squares and APL64 Project data points use green circles in these charts.



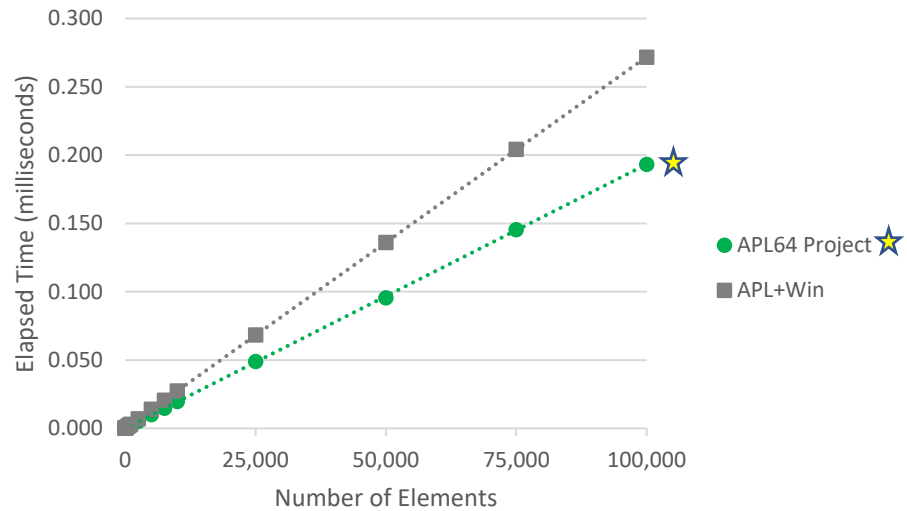
Operation: INT + FLOAT
(Lower Time is Better★)



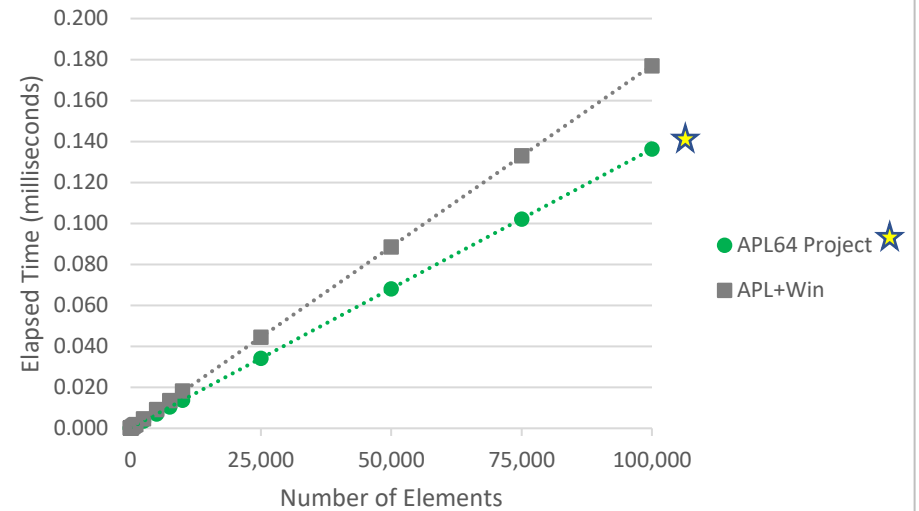
Operation: FLOAT + FLOAT
(Lower Time is Better★)



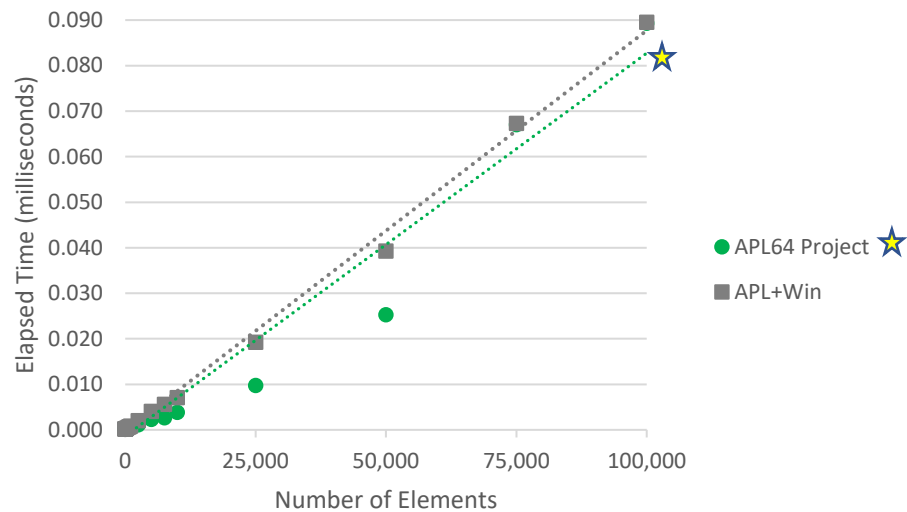
Operation: BOOL + BOOL
(Lower Time is Better★)



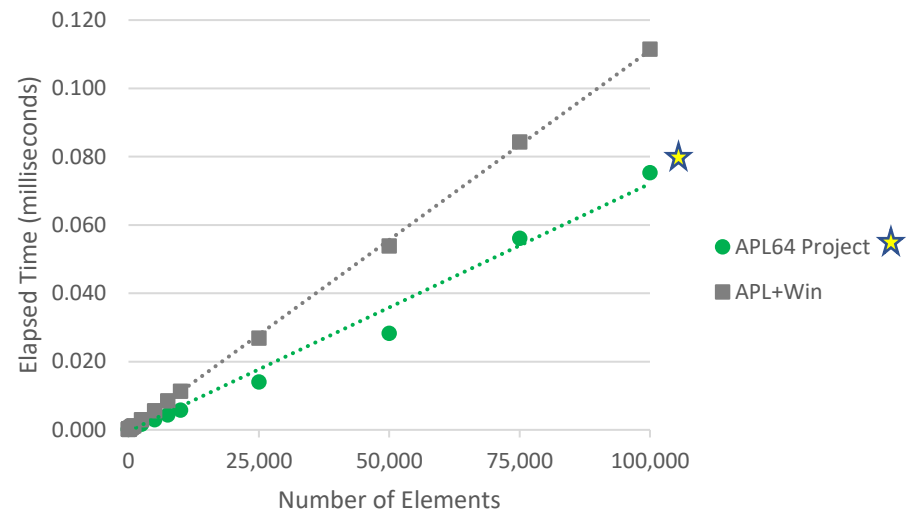
Operation: INT + BOOL
(Lower Time is Better★)



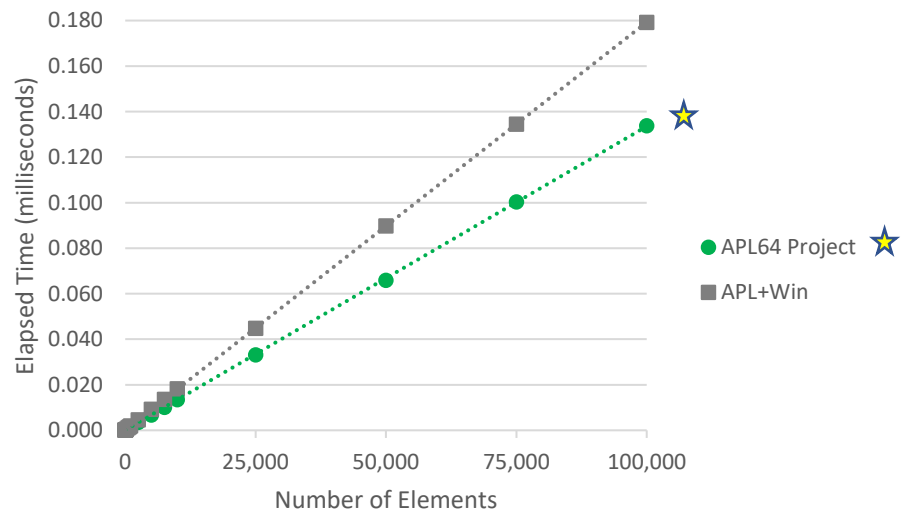
Operation: FLOAT × FLOAT
(Lower Time is Better ★)



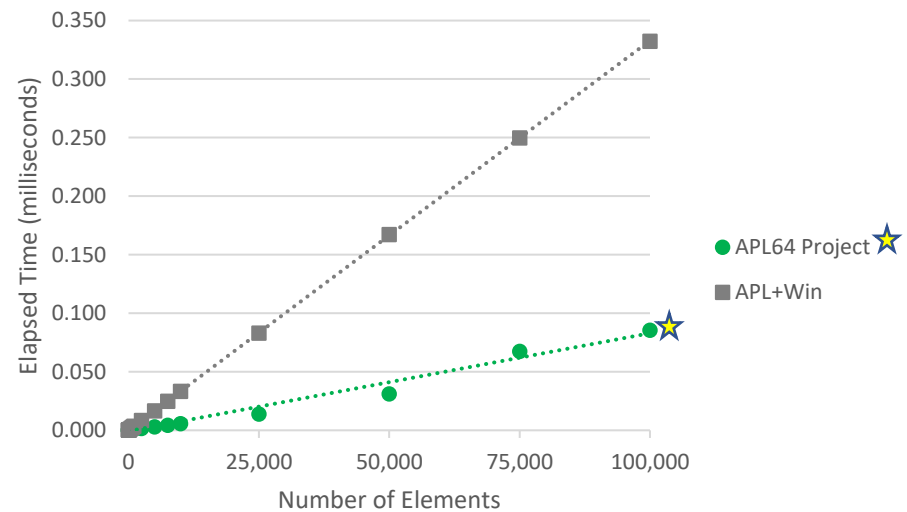
Operation: INT × FLOAT
(Lower Time is Better ★)



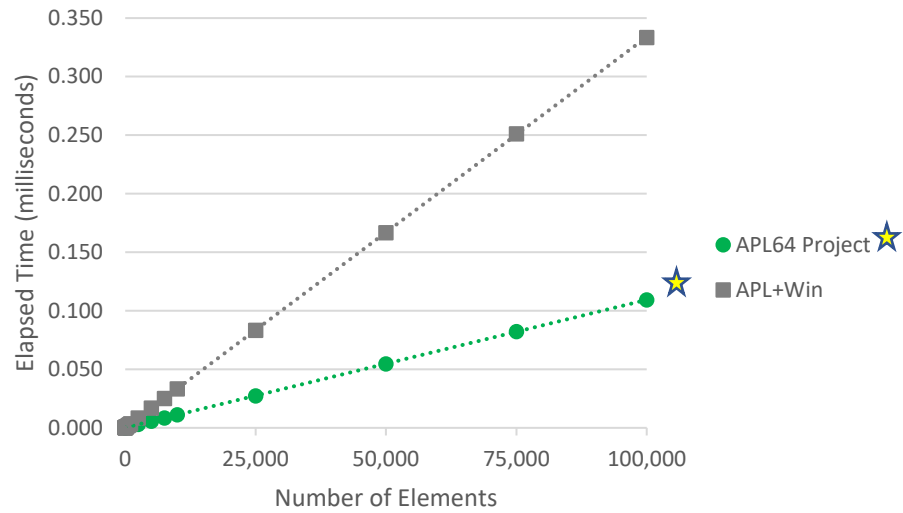
Operation: BOOL × FLOAT
(Lower Time is Better ★)



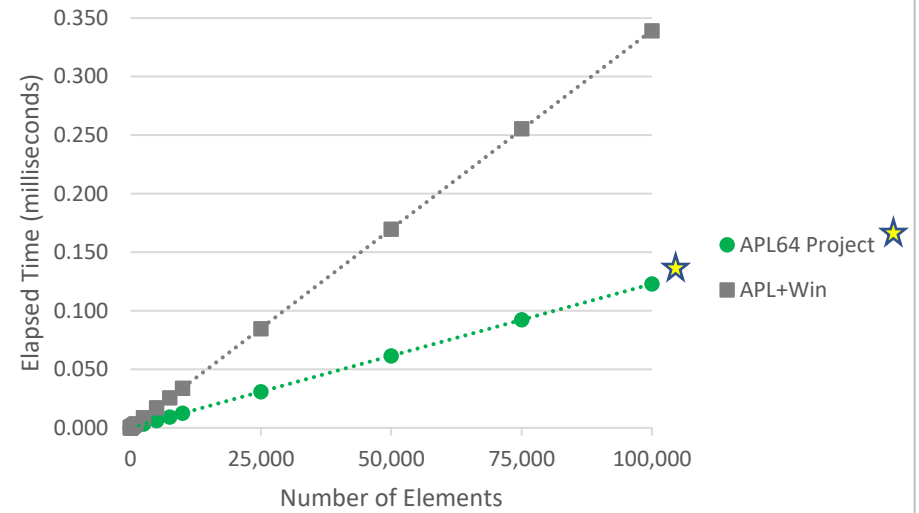
Operation: FLOAT ÷ FLOAT
(Lower Time is Better ★)



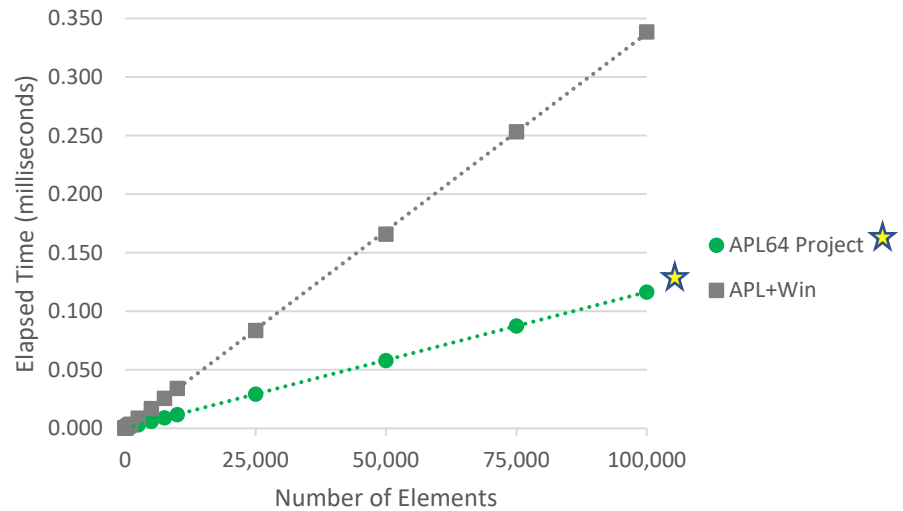
Operation: FLOAT ÷ INT
(Lower Time is Better ★)



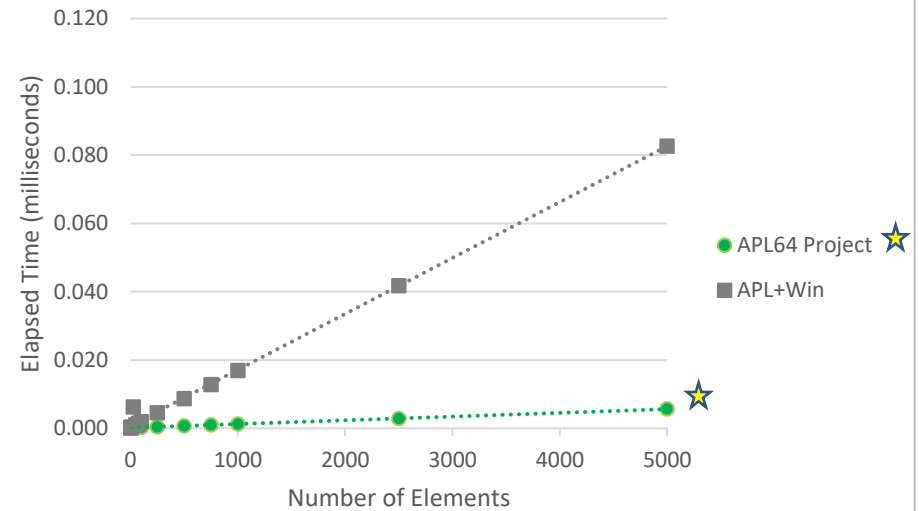
Operation: INT ÷ FLOAT
(Lower Time is Better ★)



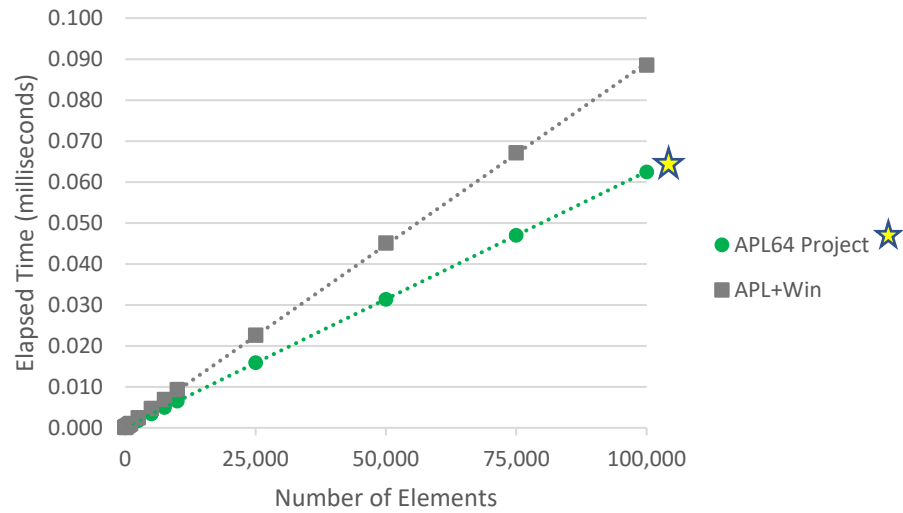
Operation: INT ÷ INT
(Lower Time is Better ★)



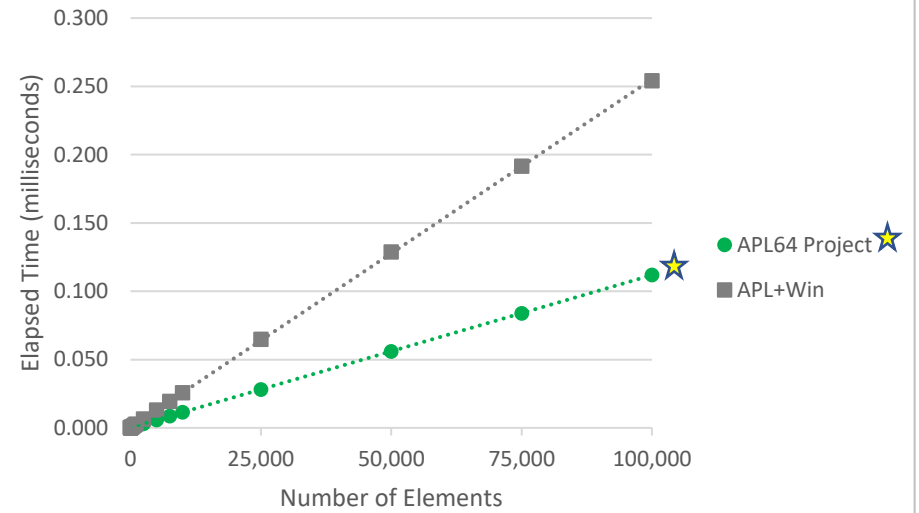
Operation: BOOL ÷ BOOL
(Lower Time is Better ★)



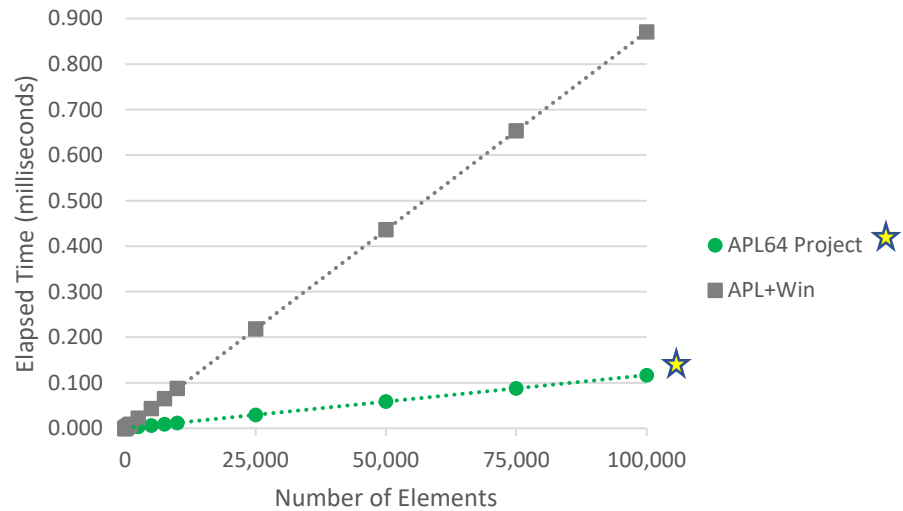
Operation: +/-INT
(Lower Time is Better★)



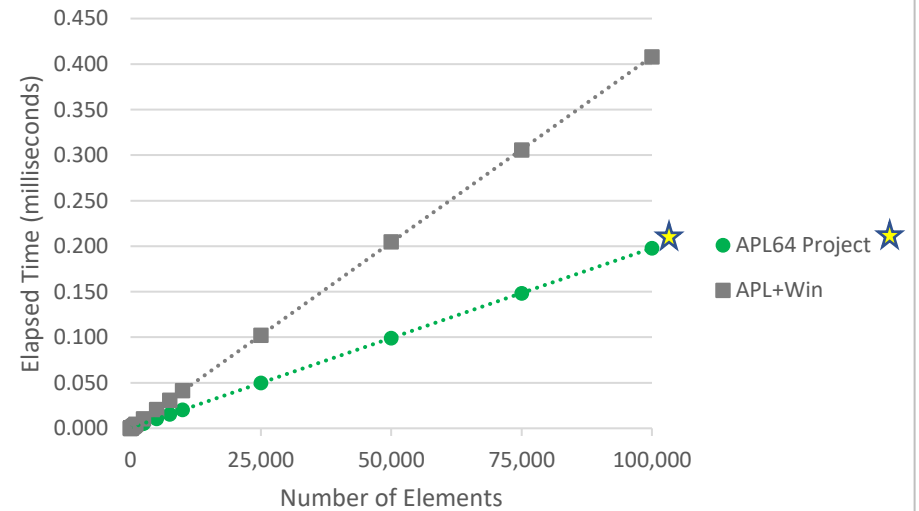
Operation: +/-FLOAT
(Lower Time is Better★)



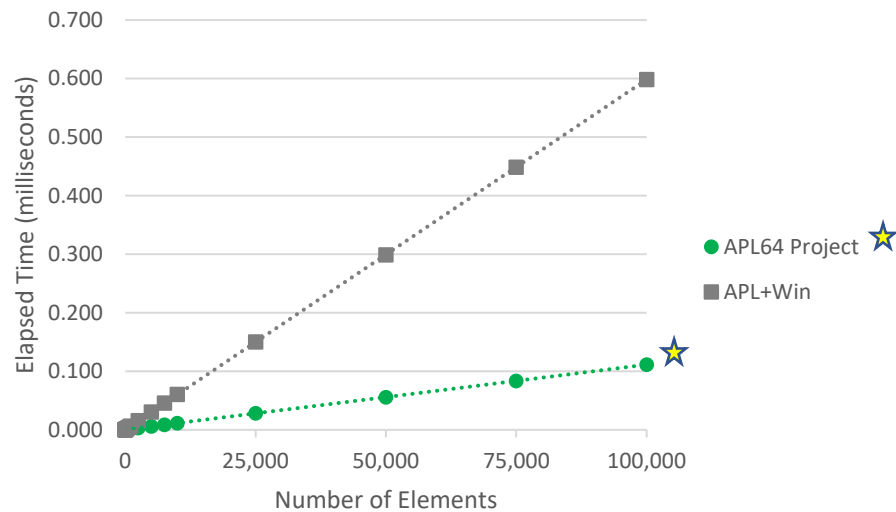
Operation: INT +.× INT
(Lower Time is Better★)



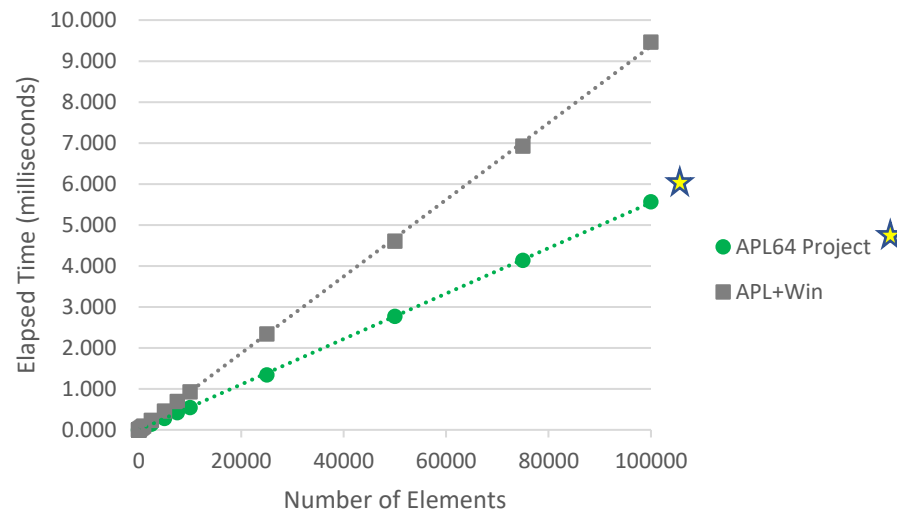
Operation: FLOAT +.× FLOAT
(Lower Time is Better★)



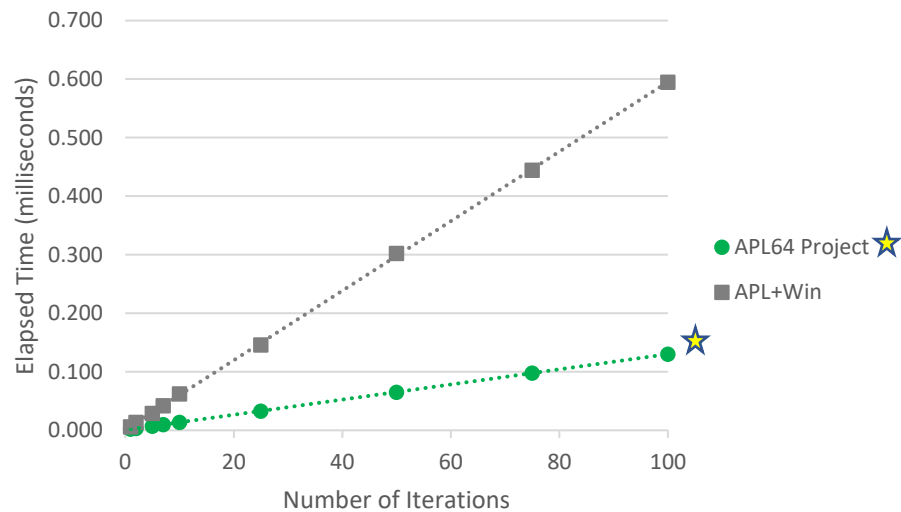
Operation: INT $\cdot$ INT
(Lower Time is Better★)



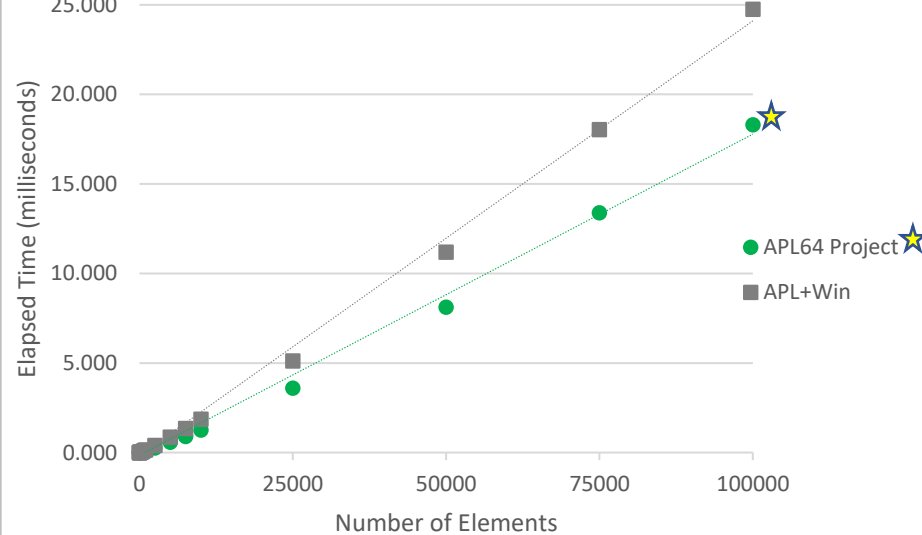
Operation: FOR Loop
(Lower Time is Better★)

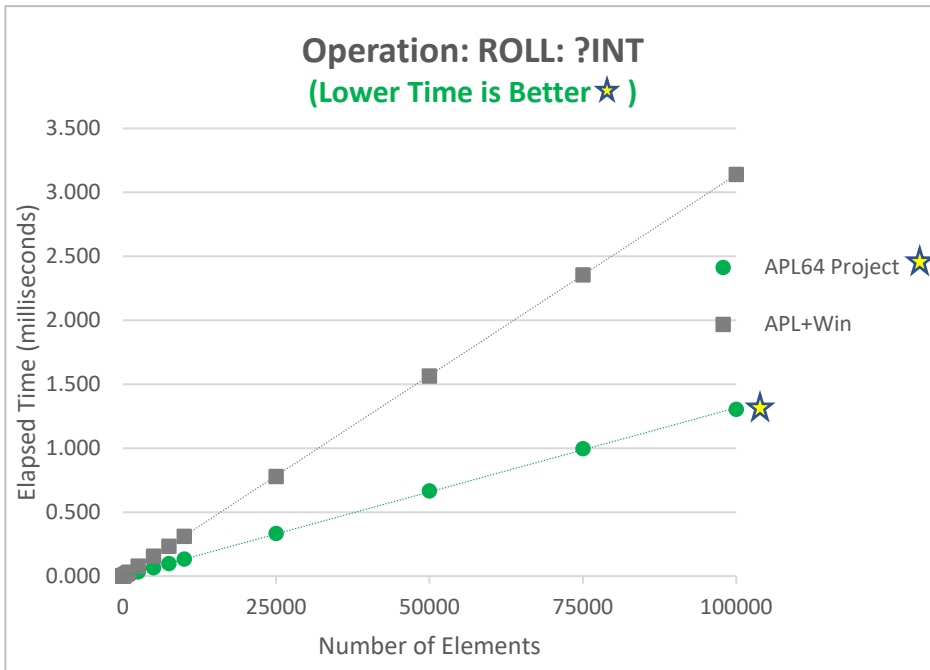


Operation: BRANCH Loop
(Lower Time is Better★)



Operation: DEAL: N?10000000
(Lower Time is Better★)



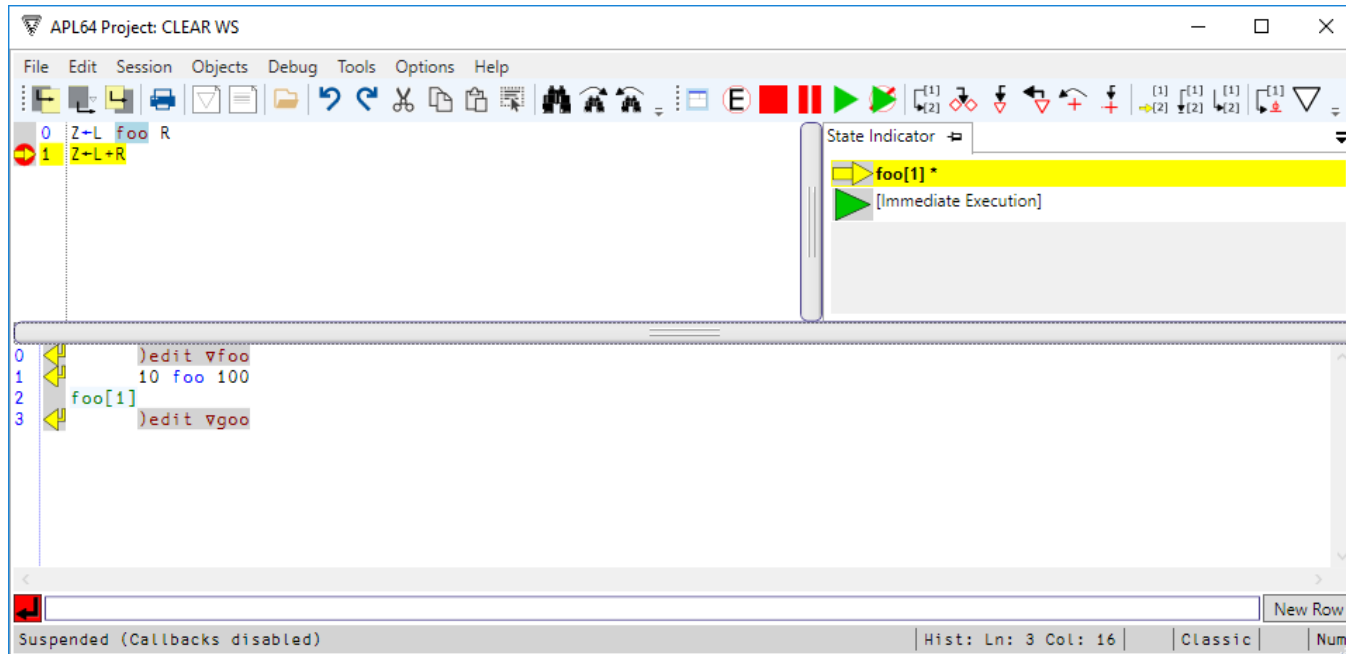


APL64 Project Programmer Session Improvements and Enhancements

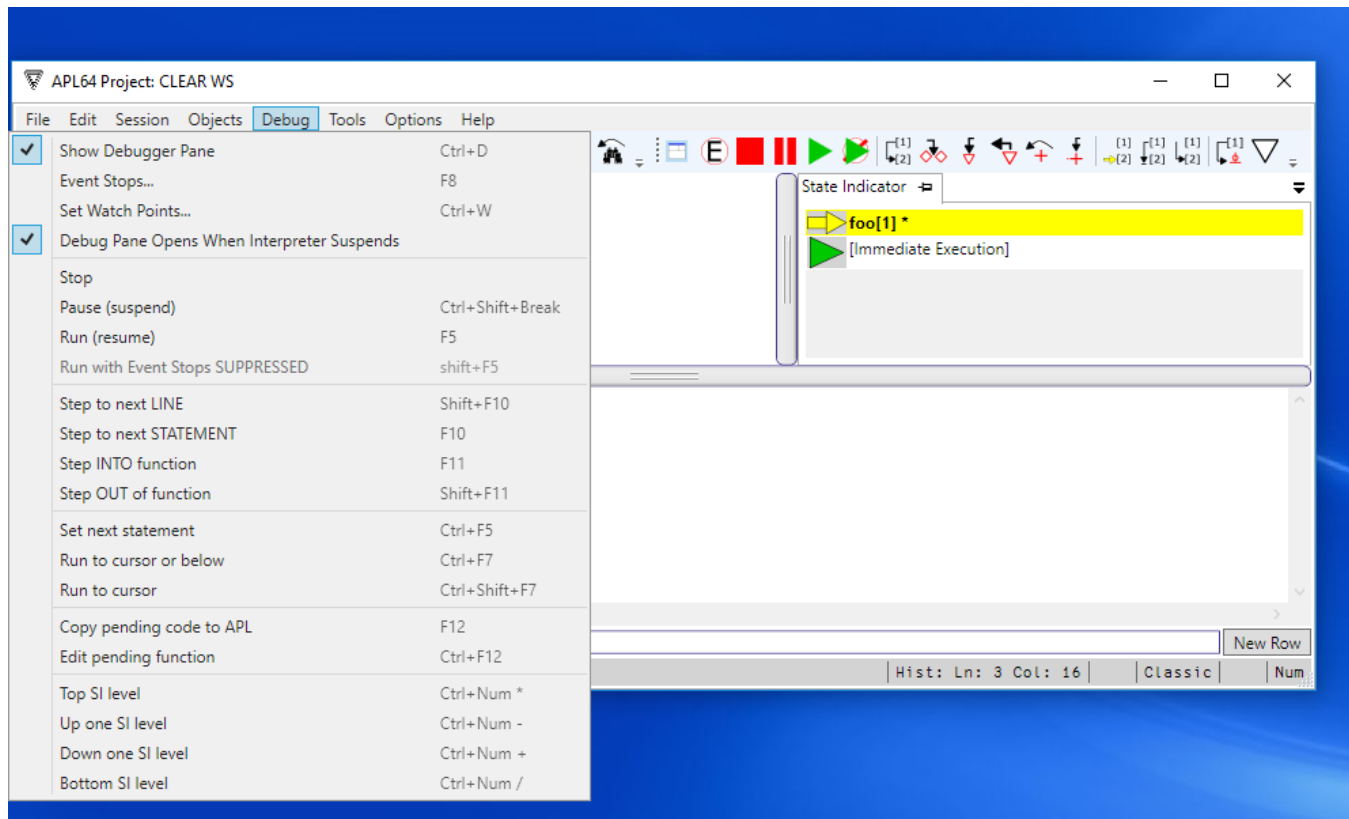
The APL64 Project Graphical User Interface (GUI) includes all the features available in APL+Win and more. This section highlights areas of the Programmer Session which have been improved by taking advantage of the up-to-date APL64 Project implementation environment.

Debugger

- The APL64 Project Debugger has been implemented:



Debugger menus and tool bar items have been implemented:



Robust Syntax Checking

Syntax checking and syntax coloring have been implemented in the APL64 Project for function editors and the session command line.

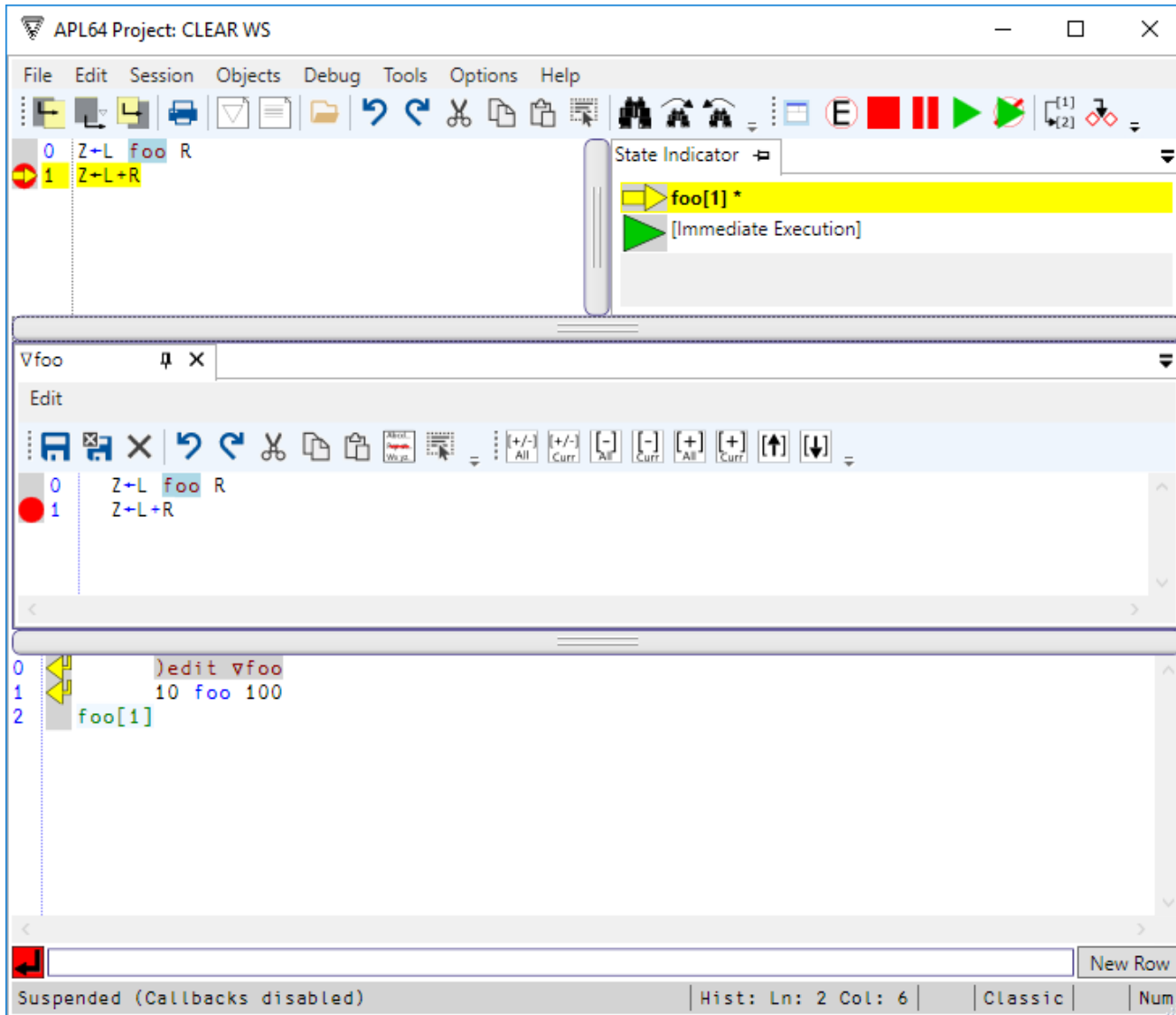
The screenshot displays the APL64 Project: CLEAR WS interface. The main window is titled 'VFOO' and contains a code editor with the following APL code:

```
0 Z←FOO X
1 Z←X+e
2
3
4 )edit VFOO
```

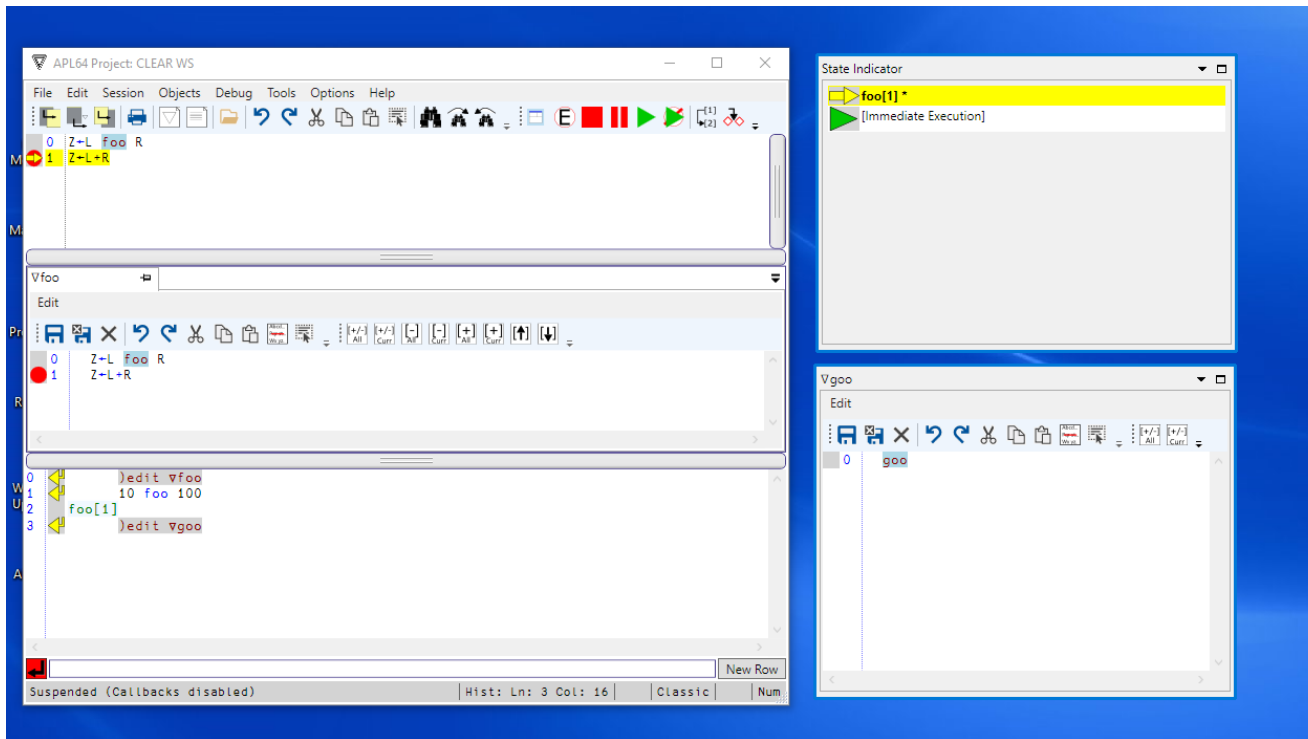
The code is syntax-colored: 'Z←FOO X' is in blue, 'Z←X+e' is in yellow, and ')edit VFOO' is in red. A 'LENGTH ERROR' message is displayed on line 1. The status bar at the bottom shows 'Ready', 'Cmd: Ln: 0 Col: 13', 'Classic', and 'Num'.

Flexible Session Panes Layout

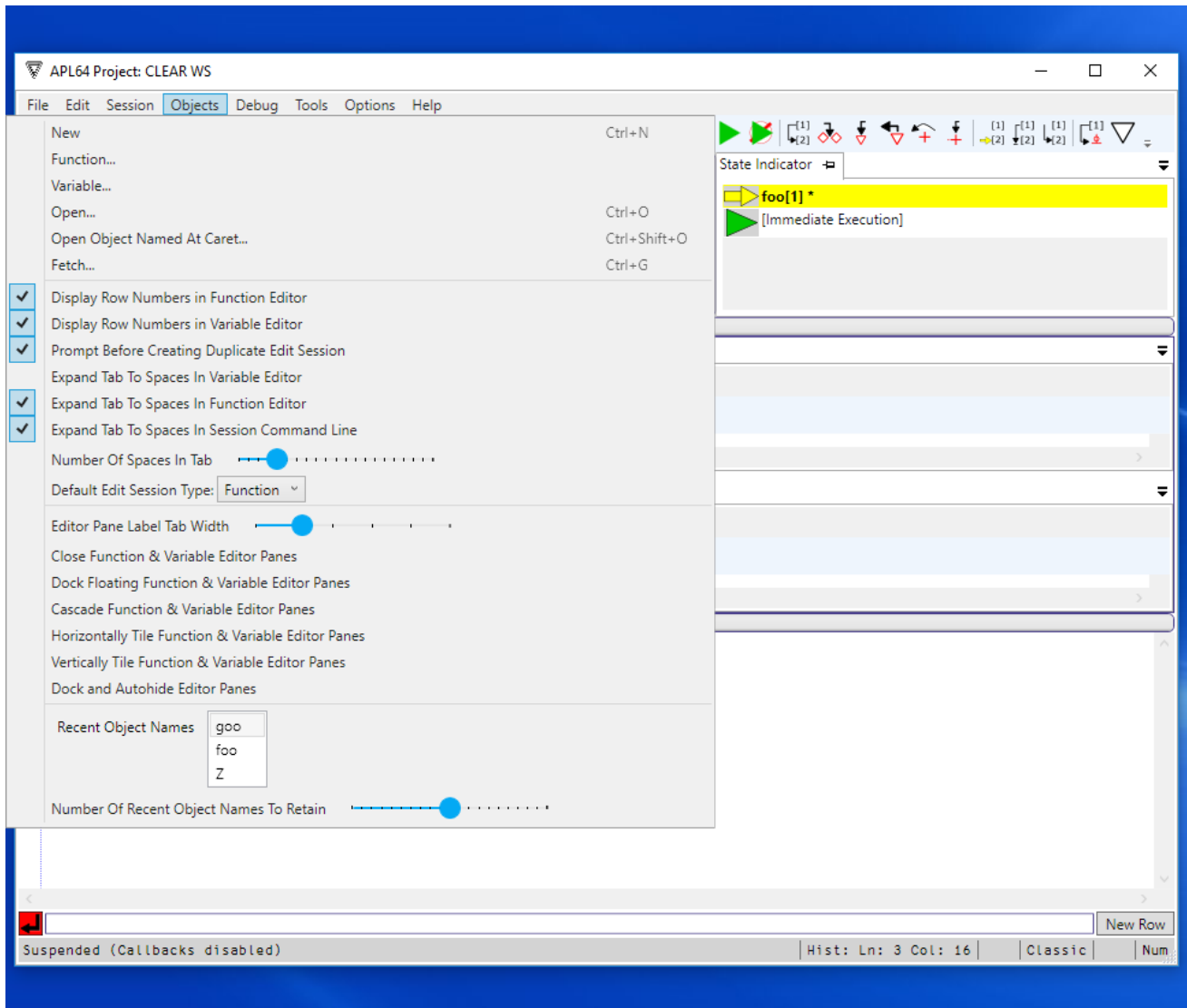
- The user may float object editor and state indicator panes anywhere on any connected display, independently of the pane containing the session history, session command line and debugged function listing.
- All panes docked with object editors vertically tiled:



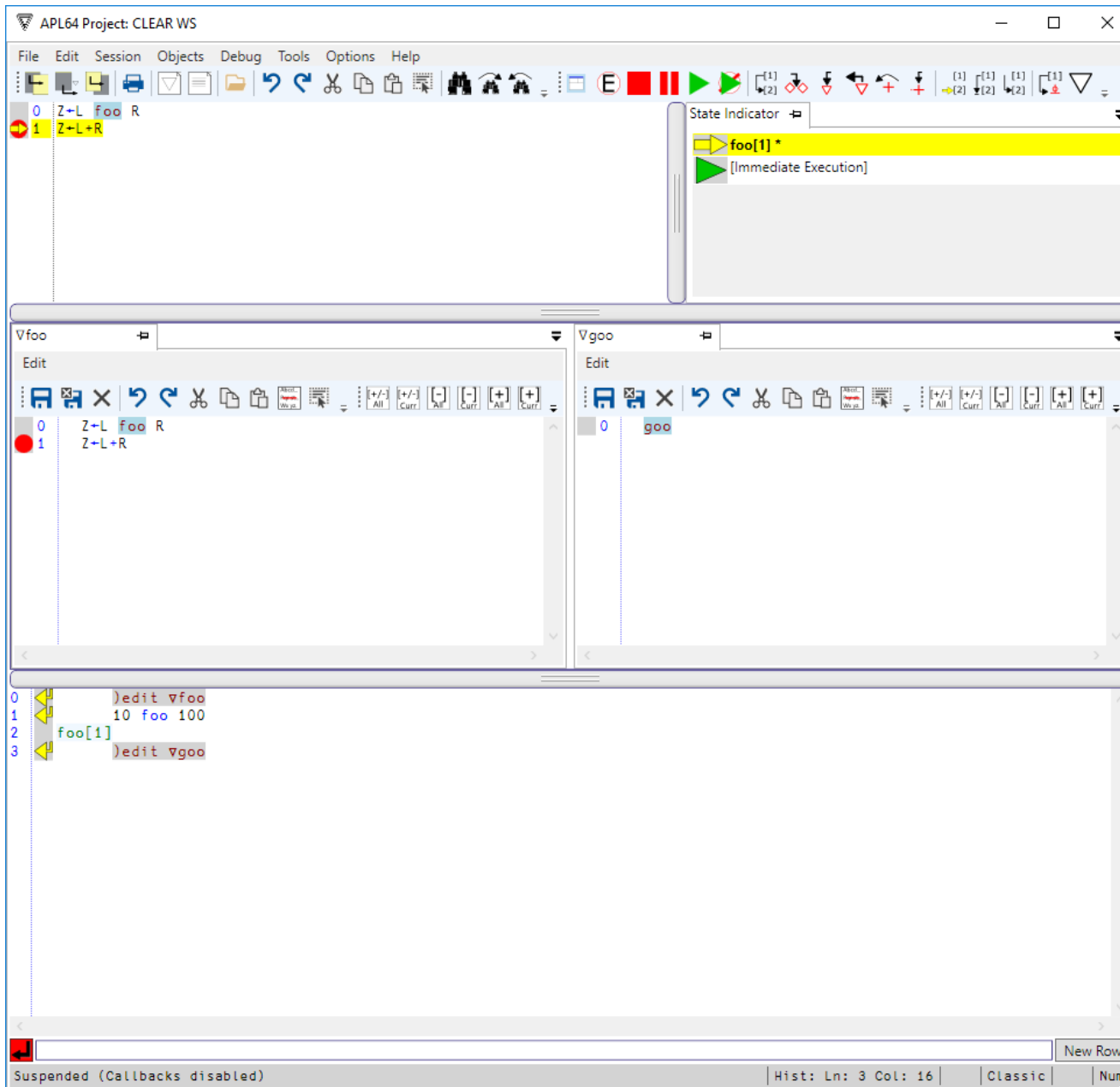
- Some panes Floating:



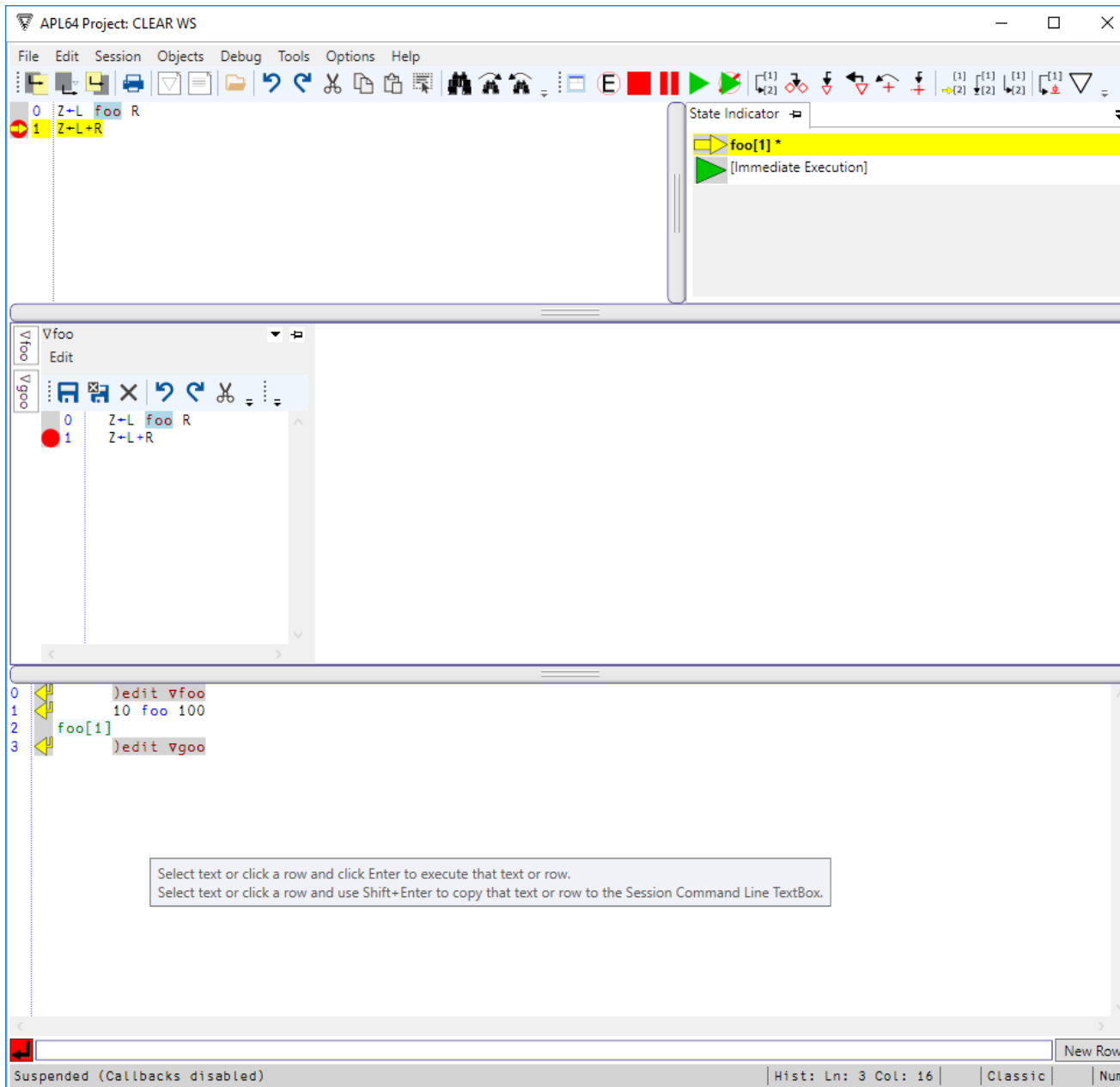
- The Objects menu provides convenient options to close, dock, cascade, horizontal tile, vertical tile and autohide object editor panes.



- Horizontally tiled object editors:



- All Panes Docked and Object Editors in Auto-hide mode:

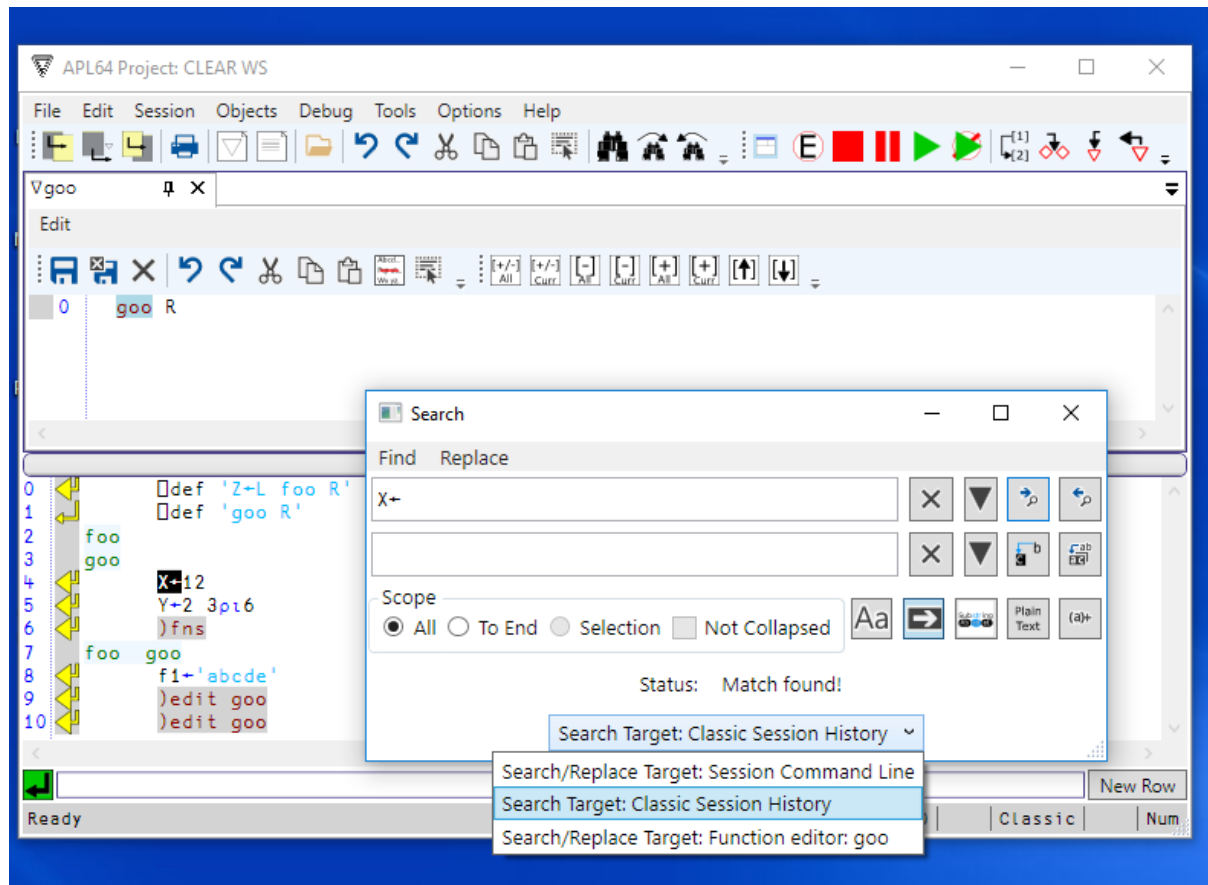


Tool Bars

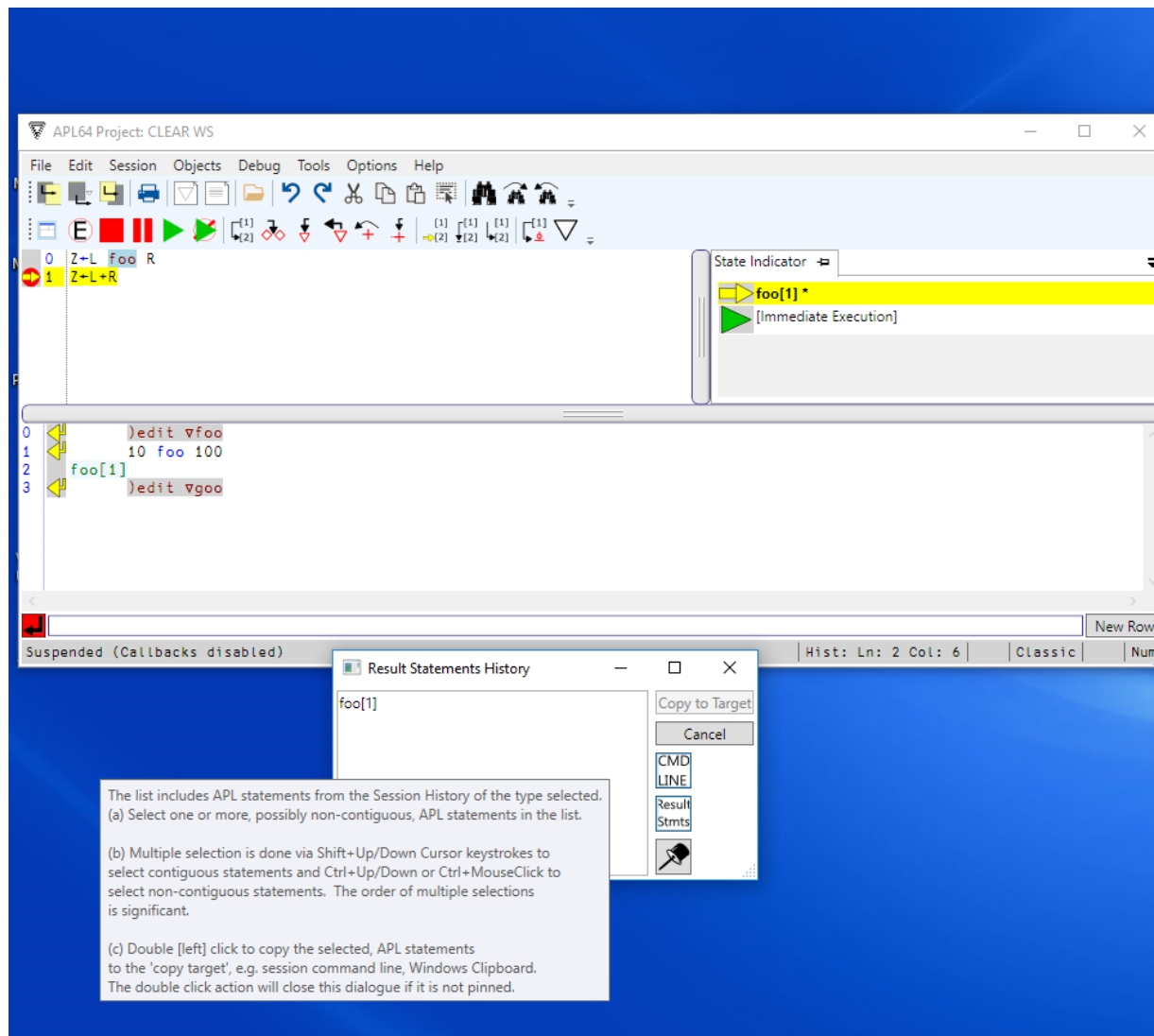
- Debugger and Editor tool bars have been implemented

Menus

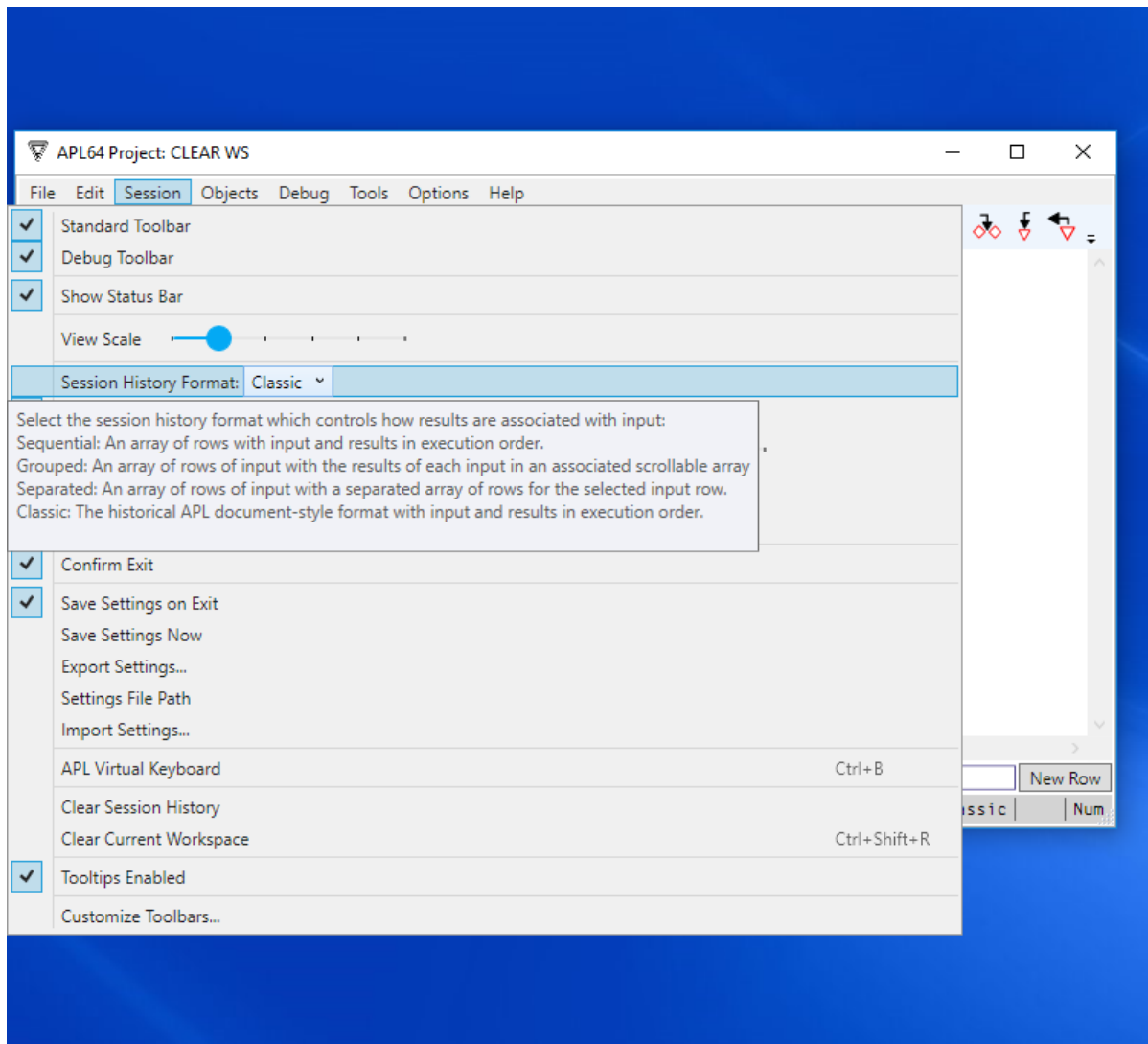
- The Load and Xload menus will transparently load the APL64 Project workspace and a previously saved APL+Win workspace.
- The Find and Replace dialogs have been unified providing the search target option and searched text format options for plain text, wild card and regex.



- The Edit > Gather menu provides an enhanced dialog to select and use APL statements from the Session History pane. Selections can be filtered by Executed, Result and all Session History statements. The copy target can be the Session Command Line or the Windows Clipboard. The dialog may be pinned and located anywhere on a connected display.

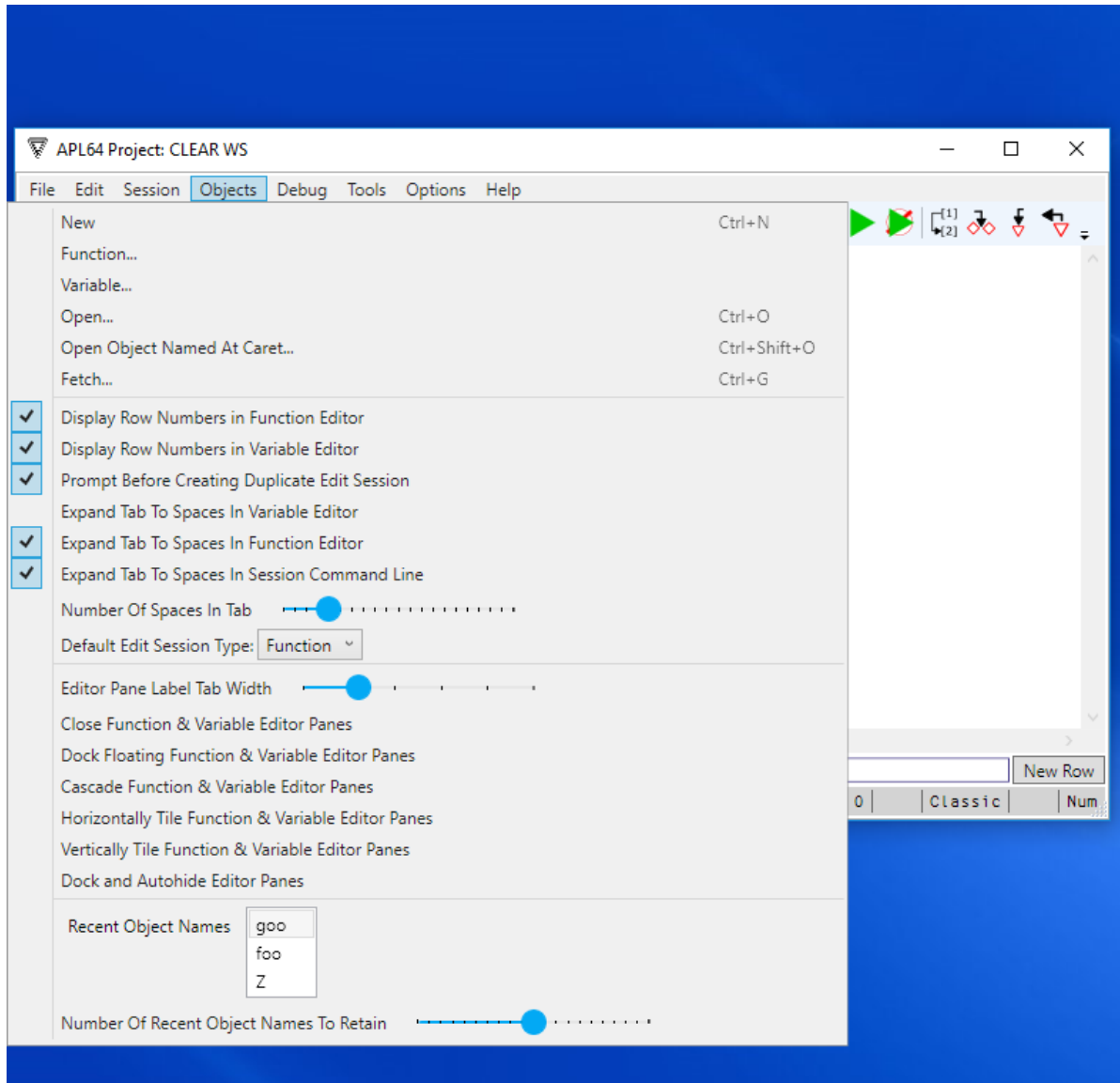


- Session > Session History Format: The session history formats have been renamed to Classic, Sequential, Grouped, Separated:

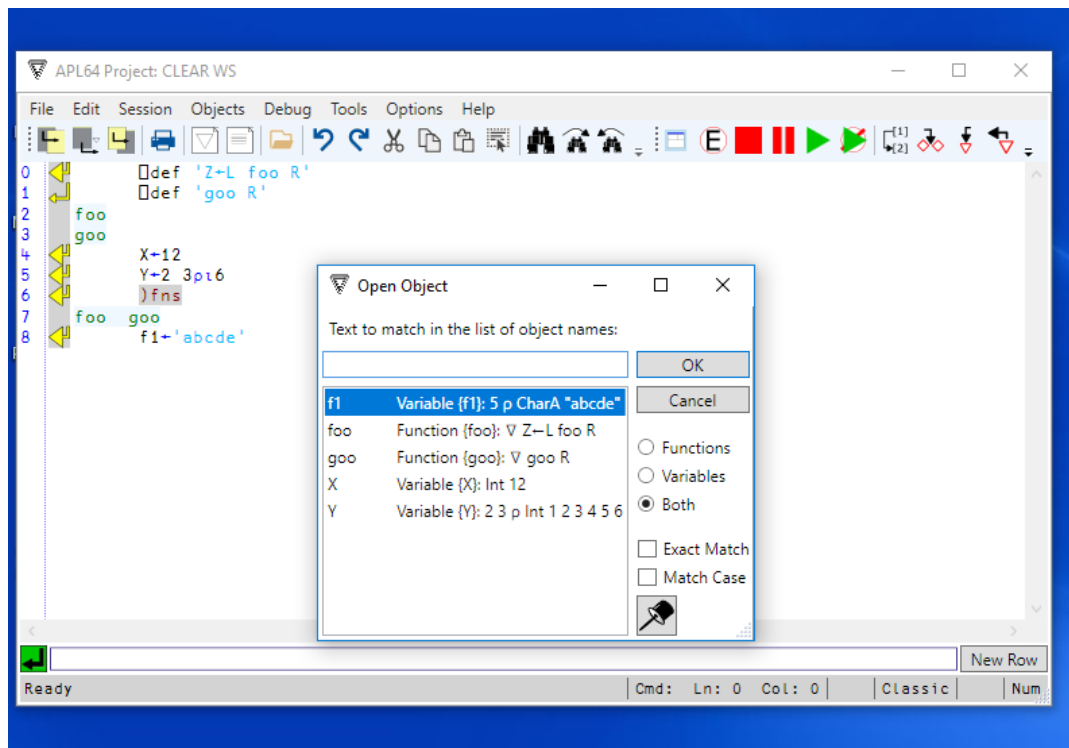


- Because the APL64 Project Session supports multi-row entries in the Session Command Line, the Session > Session Command Line Maximum Height menu items is available to control when vertical scrolling of a multi-row text in the Session Command Line will occur.

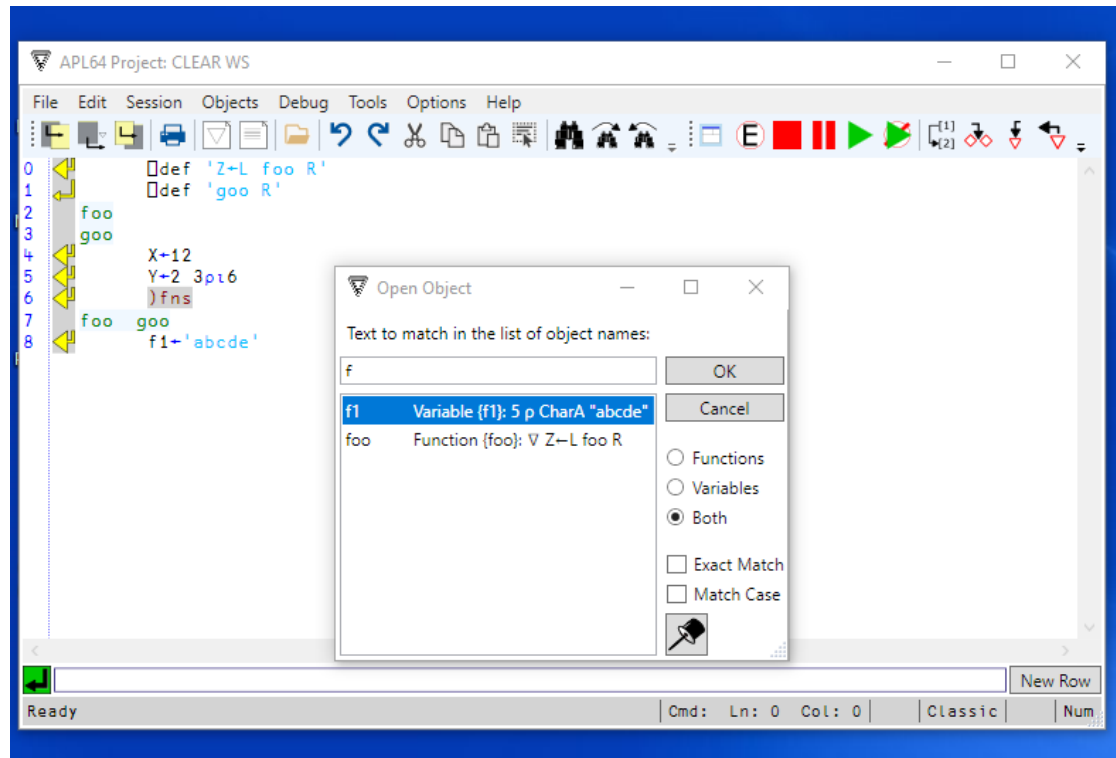
- The Session > Tooltips Enabled menu option has been implemented to activate/de-activate tooltips which are now available for most APL64 Project Session features.
- Numerous Objects menu features have been implemented:



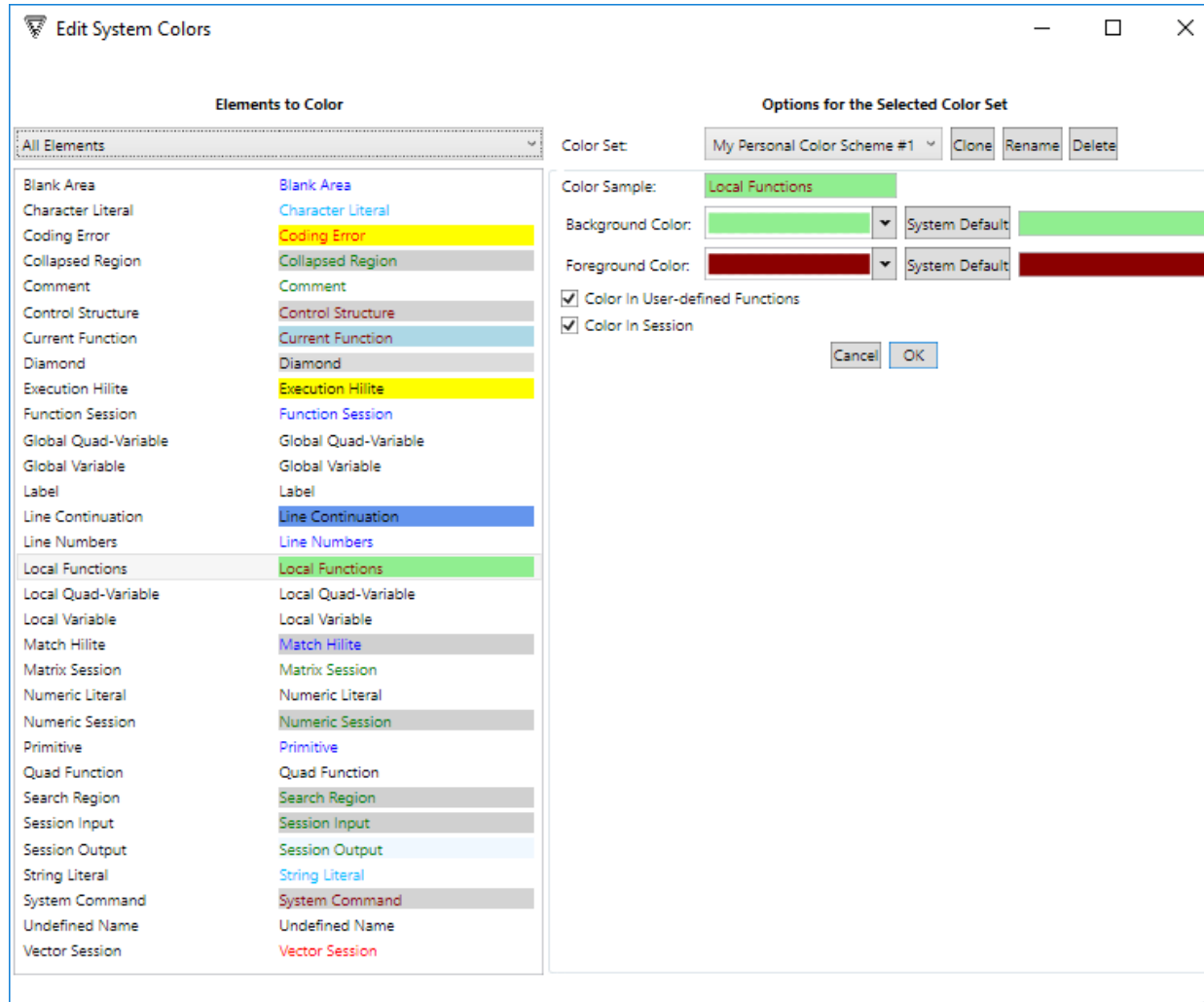
- The Open Object dialog is enhanced:
 - The list of available objects includes the objects name and value tip information.



- The object list is automatically filtered as the user types in the selection text box where filtering can be user selected among Exact Match and Match Case.



- The Options > Colors ... dialog has been enhanced:
 - Multiple stored color schemes are supported
 - Cloning of existing color schemes is supported
 - Current color scheme list includes Element Description and currently-selected color selections



- The APL Session Log Options dialog is enhanced:

The screenshot shows the 'APL Session Log Options' dialog box. It has a title bar with a minus, maximize, and close button. The dialog is divided into four sections: 'Logging State', 'Logging Threshold', 'Log File', and 'Log Import Options'. In the 'Logging State' section, there are two unchecked checkboxes: 'Currently Logging' and 'Commence Logging at Session Start'. The 'Logging Threshold' section contains a label '#Unlogged Session History Statements Threshold:' followed by a text box with '20000' and a spinner button. Below this are two checkboxes: 'Overwrite Log File' (checked) and 'Issue Alert when %Threshold is Exceeded' (unchecked). A label '%Thrtshold for Alert:' is followed by a text box with '90' and a spinner button. The 'Log File' section has a label 'Log File Prefix:' followed by a text box with 'AplSessionLog'. Below that is a label 'Log File Path:' followed by a text box with 'C:\Users\Joe\AppData\Local' and a 'Browse' button. At the bottom of this section is a label 'Recent Log File:' followed by 'N/A'. The 'Log Import Options' section contains four checked checkboxes: 'Import Command Line Executed APL Statements', 'Import Result Type APL Statements', 'Import Callback Type APL Statements', and 'Import All Other APL Statement Types'. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

APL Session Log Options

Logging State

- ☐ Currently Logging
- ☐ Commence Logging at Session Start

Logging Threshold

#Unlogged Session History Statements Threshold: 20000

- ☒ Overwrite Log File
- ☐ Issue Alert when %Threshold is Exceeded

%Thrtshold for Alert: 90

Log File

Log File Prefix: AplSessionLog

Log File Path: C:\Users\Joe\AppData\Local Browse

Recent Log File: N/A

Log Import Options

- ☒ Import Command Line Executed APL Statements
- ☒ Import Result Type APL Statements
- ☒ Import Callback Type APL Statements
- ☒ Import All Other APL Statement Types

OK Cancel

- The Keyboard Definition dialog has been implemented to support user specification of the Unicode glyphs associated with keyboard keys when the user has installed the associated Windows language preference component on the target workstation.

