Some Programming Issues in EasyLanguage®

This document illustrates how TradeStation EasyLanguage may produce counter-intuitive results when calling functions. Alternative code that avoids the problem is provided and each case clearly explains why one method works and the other does not. Lastly, examples are provided showing how to avoid these situations when using studies from Jurik Research.

Contents:

1. Varying function parameter values within a loop

At times one may need to call a particular function a large number of times, whereby each call is different only with regard to the function's parameter values. For example, the following block of code …

```
Value1 = average ( close, 5 ) ;
Value2 = average ( close, 10 ) ;
Value3 = average ( close, 15 ) ;
Value4 = average ( close, 20 ) ;
```

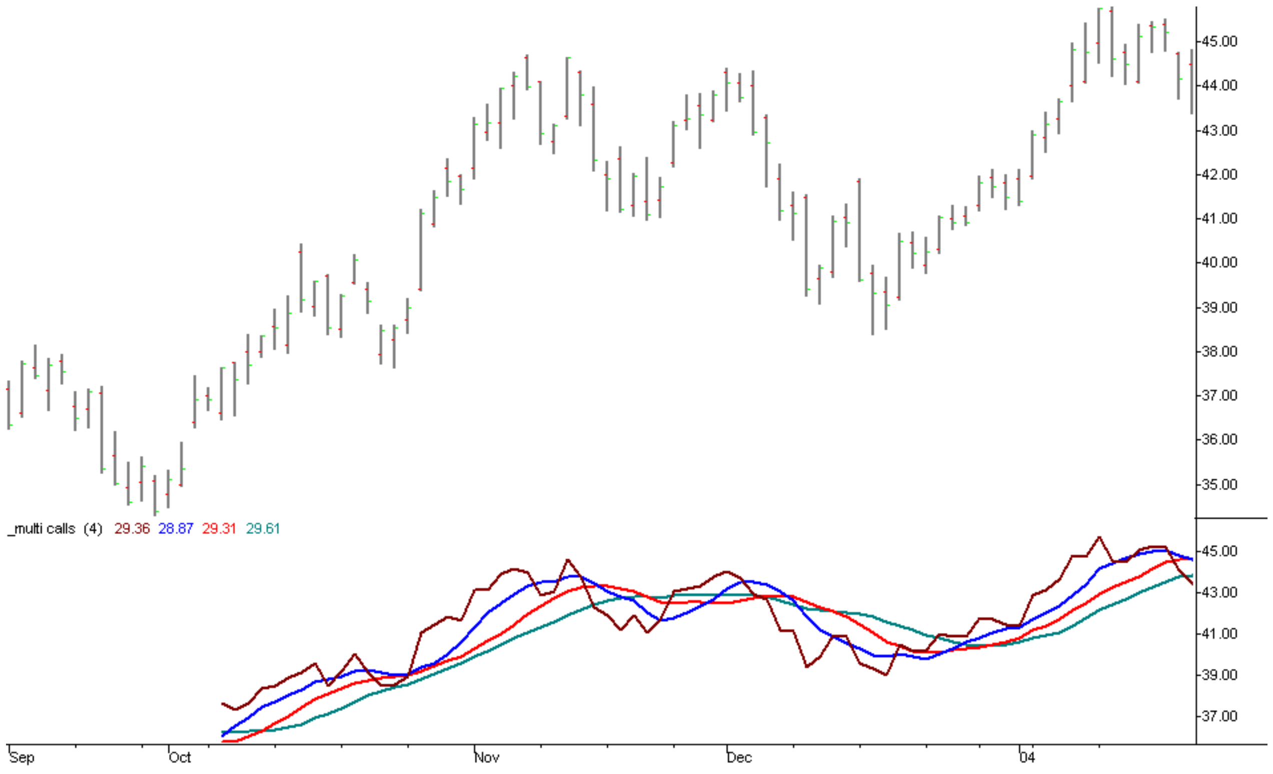
… can be rewritten in a more compact form, using an array, as follows …

```
For k = 1 to 4
      begin
      MyAvg[k] = average ( close, 5*k ) ;
      end ;
```

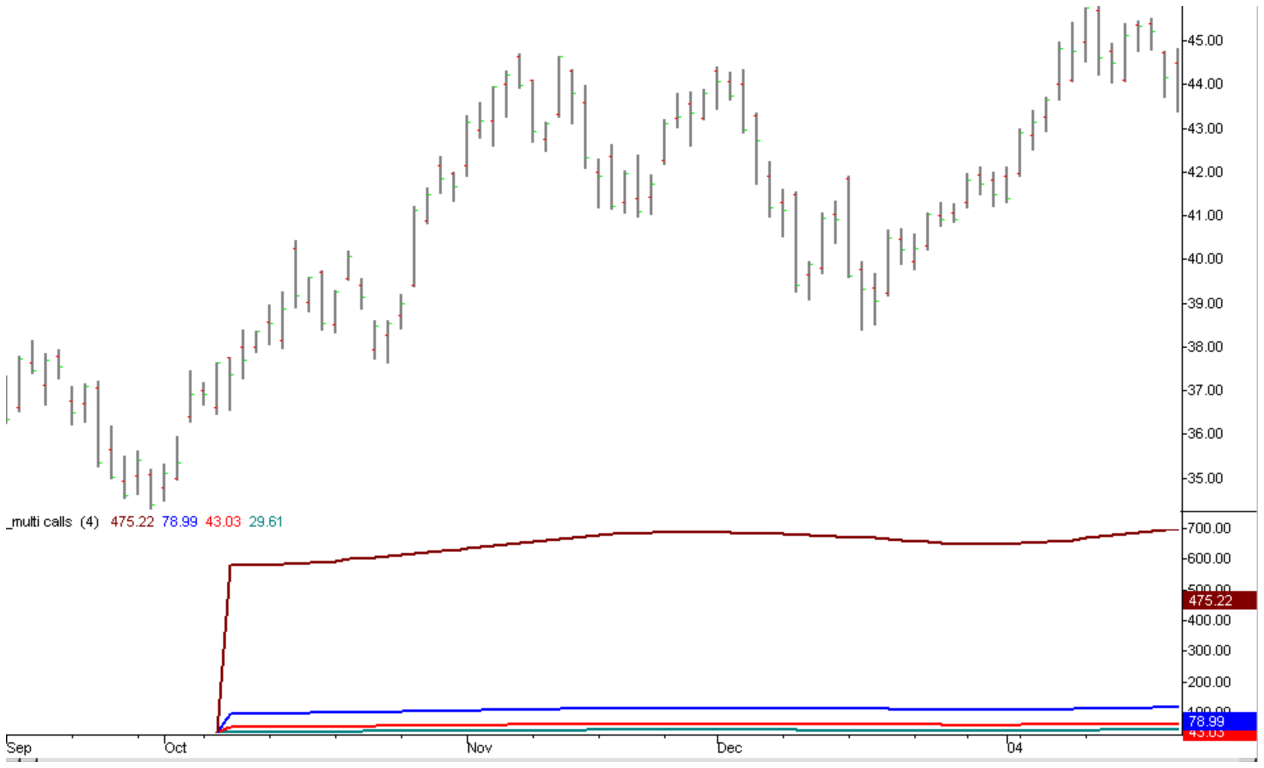
This looping method is very powerful, because it can be made to call the function N times, where N is dynamically determined. In the following sample code, an input parameter (TotalFuncs) specifies this value …

```
inputs: TotalFuncs(4) ;
array: MyAvg[](0) ;
vars: ArraySize(0), k(0) ;

if ArraySize < TotalFuncs then begin
      ArraySize = ceiling ( TotalFuncs ) ;
      Array_SetMaxIndex ( MyAvg, ArraySize ) ;
      end ;

For k = 1 to ArraySize begin
      MyAvg[k] = average ( close, 5*k-4 ) ;
      end ;

plot1(MyAvg[1],"array 1");
plot2(MyAvg[2],"array 2");
plot3(MyAvg[3],"array 3");
plot4(MyAvg[4],"array 4");
```

This produces the desired result as illustrated on the following price chart. Note that the moving averages are placed in a separate subgraph below the price bars.



When the above code is modified so that the function “**average**” is replaced with “**averageFC**”, one might expect the same results. After all, both calculate the simple moving average. The only difference between them is that **averageFC** uses an algorithm that calculates results faster than the function **average**. However, when the change is made, the result is surprising …



Two problems are immediately apparent: The lines do not resemble those in the previous image, and the y-axis values on the right are an order of magnitude larger than the original series. Why this catastrophic failure? It has to do with how the function **averageFC** performs its calculations.

Whenever the function **average** is called, it sums the most recent N price values and divides the result by N. It does not consider how the function **average** was used previously. Therefore you can call it within a loop, repeatedly using different parameters, and any call to **average** will not interfere with how **average** calculates on the next call.

Not so with **averageFC**. How does **averageFC** work?

Note that when taking the average of N price values, the calculation performed for the current price bar is almost the same as the calculation performed for the previous bar. When summing the N price values, only one value in the sum will be different than what was used in the calculation on the previous bar. Therefore, one way to speed up the calculation is to use the same sum that was calculated on the previous bar, and then make a quick adjustment to it. This completely avoids the need to sum all N price values, each and every time the averaging function moves from one price bar to the next. This is how function **averageFC** gets to be faster than function **average**.

The problem we see in the second chart stems from the fact that when **averageFC** is called, it needs to know what its previously calculated summation was. Unfortunately, when **averageFC** is called from within a loop, the summation value is being overwritten by the next call to **averageFC**, whose parameters are different. Consequently, the summation value is not being preserved from one price bar to the next, and the resulting calculation will be incorrect.

In general, any function that requires the use of values that it has calculated on the previous price bar is called a RECURSIVE function. Recursive functions should not be used within a loop. Otherwise, consecutive calls within the loop will violate the recursive algorithm’s assumption that its variables were unaltered since the previous price bar.

Other functions in TradeStation are also recursive; **XAverage** is another example. <u>All functions by Jurik Research are recursive.</u> Consequently, you should not use Jurik Research functions within a loop. The proper method is to write out the separate function calls explicitly. Here is an example …

```
value1 = JRC.JMA.2k ( close, 5 ) ;
value2 = JRC.JMA.2k ( close, 10 ) ;
value3 = JRC.JMA.2k ( close, 15 ) ;
value4 = JRC.JMA.2k ( close, 20 ) ;
```

2. Multiple calls to a series function

EasyLanguage allows the user to explicitly set the property type of a new function to be either SERIES, SIMPLE, or AUTO-DETECT. In Auto-Detect, TradeStation chooses SERIES if any variable refers to past values, (e.g. MyValue[5]), otherwise it should default to SIMPLE. (Selection is attained by right-clicking on the function text and selecting the "Properties" command on the menu.

A type **series** function has a very complex memory structure to allow access to past values of declared arrays and variables. For example, the following code is permissible in a **series** function…

```
variable: MyVal(0);
MyVal = MyVal + MyVal[7] - MyVal[20];

array: MyArr[3](0);
MyArr[2] = MyArr[2] + MyArr[2][7] – MyArr[2][20];
```

In this example, the code is looking back no farther than 20 bars into the past. We can then say the "*Look Back Horizon" or LBH* is 20.

In order to guarantee that all past values within the look back horizon will be available at any time, **series** functions must be called on every bar. Since the user can code a series function within a conditional statement, such as the following:

```
if (X > Y) then Z = _mySeriesFunc(close)
```

EasyLanguage guarantees **series** functions are called on every bar by "pulling out" all **series** functions from any conditional statements and calling them unconditionally. Therefore, in the above code, the condition "if (X>Y)" only controls whether or not Z is assigned a value. In contrast, a **simple** function, as in the example below, is called only when the condition is true:

```
if (X > Y) then Z = _mySimpleFunc(close)
```

The size of the memory structures of **series** functions depends on the study's LBH. If the user specifies a study (indicator or strategy) will look back at most 100 bars, then every **series** function called within that study will provide memory structures for supporting an LBH of 100 for each and every variable found within all those functions. That can produce an amazingly large number of memory structures for the computer to maintain!

To speed up the calculations, EasyLanguage employs various ways to circumvent this bottleneck. The first method is to define a function that has no LBH memory structures. This is called a type **simple** function. Because **simple** functions cannot support LBH, any reference to past values of a variable will be rejected by the EasyLanguage compiler. Therefore the following code is not acceptable in a **simple** function:

```
var: Myvalue(0);
Myvalue = Myvalue + Myvalue[7] - Myvalue[20] ;
```

This is not to say a **simple** function is blind to all of its past. When a **simple** function is entered, the values of all the variables are found to be exactly the same as they were when the function was exited on the previous time it was called. This concept needs some explaining …

Each time your code expresses the name of a **simple** function, EasyLanguage will create a unique copy of that function in memory. These copies are separate and do not interact, which is good because the last thing you want is data from one function to leak into another. Here is an example where EasyLanguage will create four separate copies of a **simple** function:

```
value22 = _MySimpleFunc(close,20);
value33 = _MySimpleFunc(close,15);
value44 = _MySimpleFunc(close,10);
value55 = _MySimpleFunc(close,5);
```

When this block of code is run, each specific copy of `_MySimpleFunc` is called once per bar. Therefore, when each of these four **simple** functions are entered, the values of all the variables within them are found to be exactly the same as they were when each function was exited on the previous bar.

If, on the other hand, you called the same **simple** function in a loop, such as in the following code:

```
For k = 1 to 7
      begin
      MyArray[k] = _MySimpleFunc ( close[k] ) ;
      end ;
```

then each time `_MySimpleFunc` is called in the loop, the value of the variables upon each exit will be retained for the next time it is called in the loop. After the last time it is called in the loop, the values will remain the same until the entire loop is called again on the next bar.

The fact that a **simple** function remembers what it did the last time it was called gives it enormous power. That's because it can support recursive algorithms, such as the exponential moving average (EMA).

Here is a **simple** function, called `_MySimpleFunc`, which calculates the EMA …

```
inputs:  series(numericseries), length(numericsimple);
vars:    xavg(series);

xavg = xavg + (series-xavg) * 2 / (length+1);
_MySimpleFunc = xavg;
```

Here is a **series** function, called `_MySeriesFunc`, which calculates the EMA …

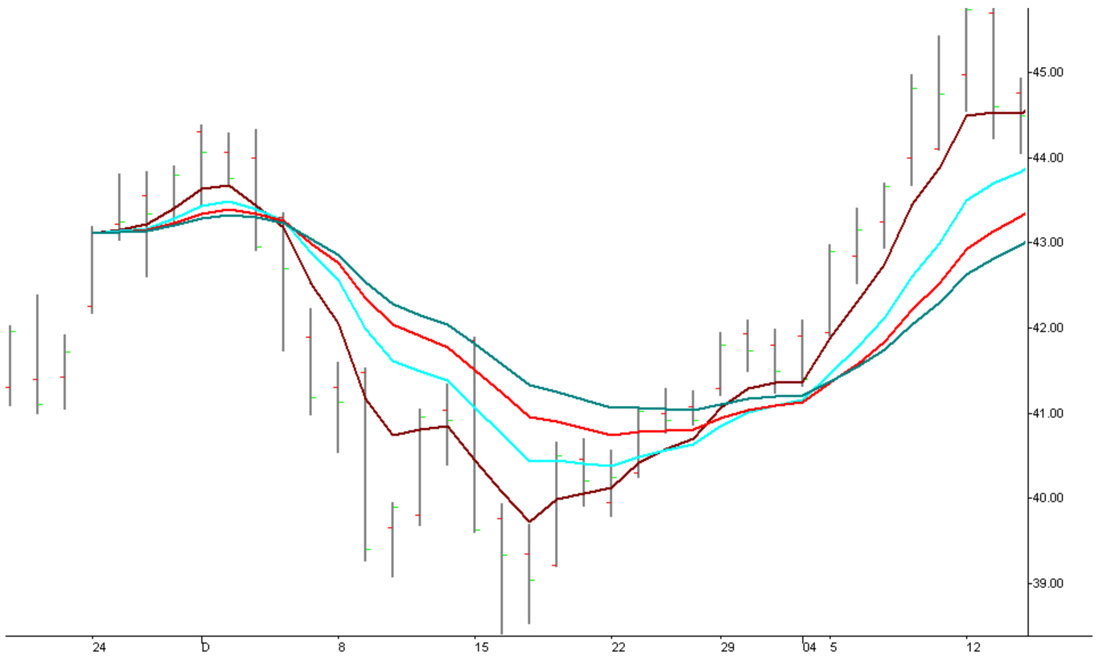
```
inputs:  series(numericseries), length(numericsimple);
vars:    xavg(series);

xavg = xavg + (series-xavg) * 2 / (length+1);
_MySeriesFunc = xavg;
```

The two functions are identical except for their function names and their property types.

Suppose you make a sequence of identical calls to `_MySimpleFunc` and plot the results, as shown in the following code …

```
var: len(0);

len = 20;   value22 = _MySimpleFunc(close,len);
len = 15;   value33 = _MySimpleFunc(close,len);
len = 10;   value44 = _MySimpleFunc(close,len);
len = 5;    value55 = _MySimpleFunc(close,len);

plot1(value22,"Line 1");
plot2(value33,"Line 2");
plot3(value44,"Line 3");
plot4(value55,"Line 4");
```

You would get a chart showing the four moving averages, as illustrated below …



If you simplified the code by removing the variable “len”, …

```
value22 = _MySimpleFunc(close,20);
value33 = _MySimpleFunc(close,15);
value44 = _MySimpleFunc(close,10);
value55 = _MySimpleFunc(close,5);

plot1(value22,"Line 1");
plot2(value33,"Line 2");
plot3(value44,"Line 3");
plot4(value55,"Line 4");
```

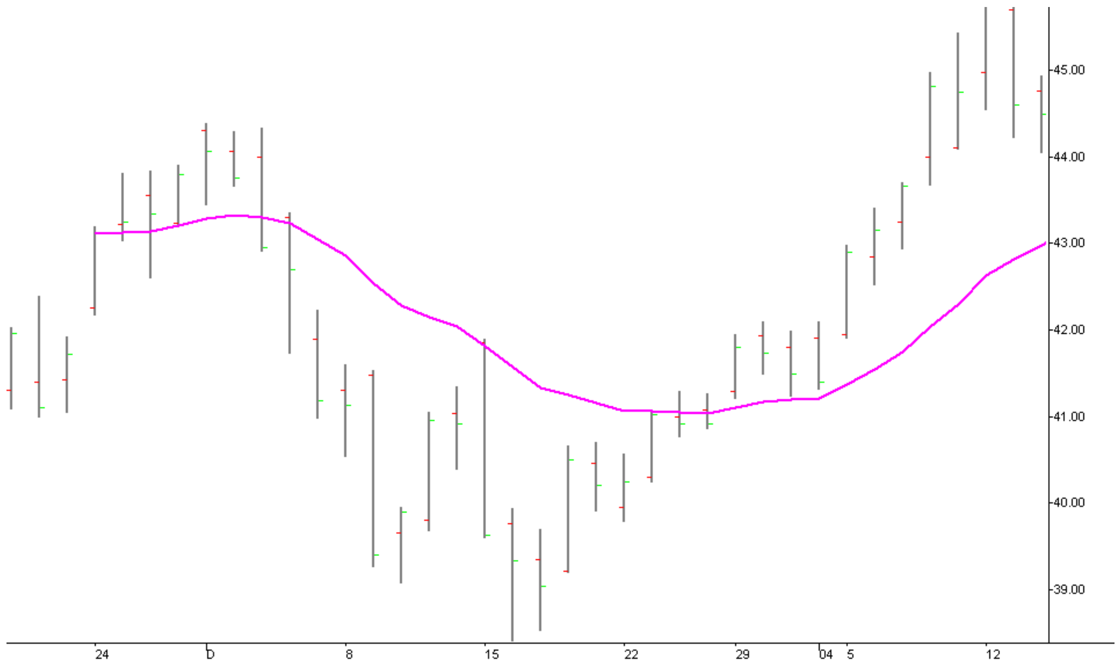
… the plot would look the same, just as you would expect. Now let's replace "MySimpleFunc" with "MySeriesFunc".

```
value22 = _MySeriesFunc (close,20);
value33 = _MySeriesFunc (close,15);
value44 = _MySeriesFunc (close,10);
value55 = _MySeriesFunc (close,5);

plot1(value22,"Line 1");
plot2(value33,"Line 2");
plot3(value44,"Line 3");
plot4(value55,"Line 4");
```

This produces a chart identical to the one above. So far, so good. However, when the code is modified to include the "length" parameter as a variable, as shown here …

```
var: len(0);

len = 20;   value22 = _MySeriesFunc (close,len);
len = 15;   value33 = _MySeriesFunc (close,len);
len = 10;   value44 = _MySeriesFunc (close,len);
len = 5;    value55 = _MySeriesFunc (close,len);

plot1(value22,"Line 1");
plot2(value33,"Line 2");
plot3(value44,"Line 3");
plot4(value55,"Line 4");
```

… the chart produces four plots, all the same, and all you see is one visible plot line!!



What triggers this drastic change in behavior is that all four function calls, on a syntactic level, appear to the compiler as being identical …

```
            value22 = _MySeriesFunc (close,len);
            value33 = _MySeriesFunc (close,len);
            value44 = _MySeriesFunc (close,len);
            value55 = _MySeriesFunc (close,len);
```

Why is this condition so special? During the various version builds of TradeStation 5.0, there were significant changes to how functions operated, so that users did not have to contend with the numerous programming idiosyncrasies of **series** and **simple** functions that existed back then. In addition, some heuristics were added to the compiler to help it guess what you were doing so that it could take appropriate measures to optimize performance. One of the heuristics appears to be this:

If there are multiple calls to the same **series** function, with the same parameter names, then the user probably meant to call the function only once, store it in a variable and used the result as needed. In other words, if the code contains these lines …

```
            value22 = _MySeriesFunc(close, length) ;
            value33 = _MySeriesFunc(close, length) ;
            value44 = _MySeriesFunc(close, length) ;
            value55 = _MySeriesFunc(close, length) ;
```

… the compiler assumes the user’s code was redundant, and what he really meant to write was this …

```
            temp = _MySeriesFunc(close, length) ;

            value22 = temp ;
            value33 = temp ;
            value44 = temp ;
            value55 = temp ;
```

In effect, the compiler appears to “pull out” only one copy of `_MySeriesFunc`. No regard is given to the fact that the value of “length” had changed between function calls, from 20 to 15 to 10 to 5.

The parameter values used for this “pulled out” function are those that were present during the first time the series function call appears in the code. In our example, the code executes as if the compiler rewrote it as follows:

```
// Create only one version of _MySeriesFunc in memory
//     and call it _SpecialFunc.
// Set its parameters to the values used during the first
//     time _MySeriesFunc appears in code.

len = 20;         value22 = _SpecialFunc (close,20);
len = 15;         value33 = _SpecialFunc (close,20);
len = 10;         value44 = _SpecialFunc (close,20);
len = 5;          value55 = _SpecialFunc (close,20);
```

Clearly this is not what was intended. This is also true for Jurik's **series** functions. For example, the **series** JMA function will not operate correctly when coded in this manner.

```
len = 20;   value22 = JRC.JMA.2k (close,len,0);
len = 15;   value33 = JRC.JMA.2k (close,len,0);
len = 10;   value44 = JRC.JMA.2k (close,len,0);
len = 5;    value55 = JRC.JMA.2k (close,len,0);
```

To avoid this problem, the user has two options: either make the parameter names different, or use the **simple** JMA function instead. The user manual associated with each Jurik tool (JMA, VEL, RSX, CFB, etc.) specifies the EasyLanguage names of the **series** and **simple** functions.

Code for the first approach, using different names for variables, might look like this …

```
len1 = 20;  value22 = JRC.JMA.2k (close,len1,0);
len2 = 15;  value33 = JRC.JMA.2k (close,len2,0);
len3 = 10;  value44 = JRC.JMA.2k (close,len3,0);
len4 = 5;   value55 = JRC.JMA.2k (close,len4,0);
```

Code for the latter approach, using a JMA simple function, might look like this …

```
len = 20;   value22 = JRC.JMA.flex.2k (close,len,0);
len = 15;   value33 = JRC.JMA.flex.2k (close,len,0);
len = 10;   value44 = JRC.JMA.flex.2k (close,len,0);
len = 5;    value55 = JRC.JMA.flex.2k (close,len,0);
```

Either approach will produce the correct results.

3. Series or Simple, know for sure.

Years ago, TradeStation 4 would list all the functions along with its **series**/**simple** designation. In TradeStation 8, the list does not display function type. The only way to really know if your user function will perform as **simple** or **series** is to follow these steps:

1. Explicitly select either **series** or **simple** in the Properties box (which appears when you select the "Properties" menu command after right-clicking on the code.)

2. For **series** functions, add the following extra line of code at the end:

   ```
   var: ForceSeries(0);  ForceSeries = ForceSeries[2];
   ```

   This is needed because some versions of TradeStation will override your **series**/**simple** selection and make its type **simple** if no references to past values (using square brackets) are found.

3. Verify the function. Pressing the F3 hotkey is the easiest way. Verification forces the code to be recompiled in a manner compatible to the specified function type and the user will be alerted to any incompatible code.