Design

# Graphical User Interface

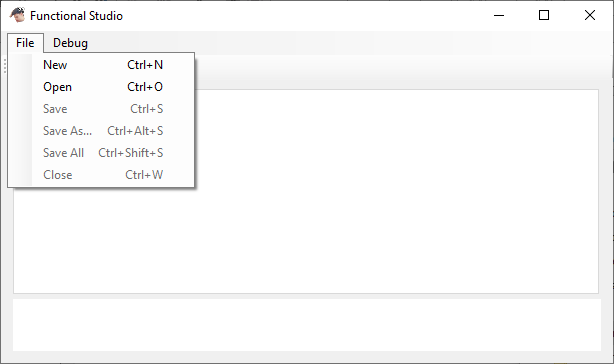
## Primary functionality in menus and buttons

As this app is an IDE, with style intended to lie between those of Visual Studio and Notepad++, the most visually important and striking features are those that lend to the app’s base functionality:

* Menu strip with dropdown options
* Toolbar buttons

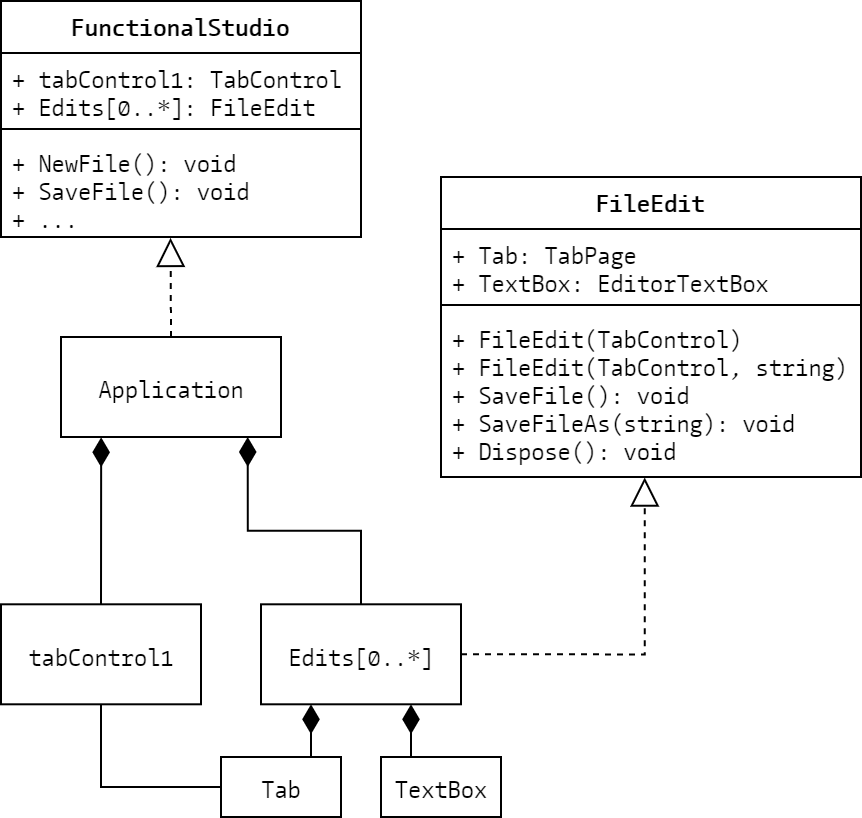
Within the File dropdown are the options: New, Open, Save, Save As, Save All, and Close. Within the Debug dropdown is the option Start. The Toolbar buttons correspond to the New, Open, Save, Save All, and Start options. All of these functionalities have working shortcuts provided by the built-in tool in Visual Studio (available in Properties for dropdown control).

New Open Save Save All Start



Primary functionality provided by menu dropdowns and the toolbar which adapt to context

For every functionality there is an associated event handler method as part of the form class. They interact with the underlying file editing implementation, as shown in the FunctionalStudio form class in FunctionalStudio.cs

All of the methods interact directly with the instances of the FileEdit class which handle each open file, including its tab as part of the tab control.

FileEdit implementation in FunctionalStudio

Filled diamond-head arrows indicate composition, dotted-line hollow triangle-head arrows indicate implementation i.e. Application is an instance of FunctionalStudio and therefore implements it

Note for the following class descriptions:

* Light blue indicates a class or structure type
* Dark red indicates an object or structure
* Cyan indicates an enumerator structure or value
* Orange indicates a method
* Dark yellow indicates an event
* Dark blue indicates a variable with a base C# type e.g. int, string etc.

## FileEdit

FileEdit is a class which encapsulates most of the file interaction functionality i.e. opening, saving, etc. while also controlling tab instantiation and disposal in the tab control it is associated with. It is important to note however that FileEdit does not itself provide functionality for accessing file directories, choosing save locations etc. such as is provided using file dialogs; this is handled at the form level and in the methods that handle the button click events of the appropriate options.

### Construction and primary properties

FileEdit has two constructors: one for initialising a new blank file, and one for opening an existing file. Both take a TabControl object as an argument, but the open file constructor needs a file path as a string to be passed. Both follow this sequence of procedures:

* Instantiate a new TabPage object called Tab with Text set to “Untitled” for a new file, or the file name from the file path if opening a file
* Instantiate a new EditorTextBox object called TextBox (custom control class, addressed at …) with Font set as a new Font object representing the Consolas font, size 9. If opening a file, assign the result of the static method ReadAllText from the built-in File class when passed the file path to the TextBox.Text property. This sets the contents of the text box to the contents of the file at the given file path. FileEdit throws an exception if the file path is invalid
* Add TextBox to the Controls collection of Tab i.e. put the text box control inside the tab, then set TextBox.Dock to DockStyle.Fill i.e. force the text box control to fill up the whole tab
* Assign the TextChanged event handler method to the TextBox.TextChanged event i.e. when the text changes in the text box, call the event handler. This is primarily used to update the saved status of the file
* Add Tab to the TabPages collection of the TabControl object i.e. add the newly created tab to the control, so it now appears as a clickable tab that can be selected
* Finally if opening a file, the update method for the text box is called to properly adjust for size. The reason why is explained in the custom control section

It’s important to note how the TabControl object is never assigned to a variable in FileEdit. The class doesn’t deal with the tab control and instead only interacts with the individual tab page it is associated with.

FileEdit contains two other properties Saved and FilePath, both of which are publically accessible (where Saved is effectively used as a flag to indicate unsaved changes), but both can only be changed from within the class. Saved always starts when a FileEdit object is constructed as true, and FilePath either starts as an empty string for a new file, or is assigned the string of the file path argument passed to the constructor.

The private TextChanged method controls the state of Saved, and also drives a subtle but integral UI feature of drawing an asterisk next to the title of an unsaved file. If Saved is true, then set it to false and append an asterisk to Tab.Text i.e. “Untitled” becomes “Untitled\*”.

### Primary file interaction

The SaveFile method does what it describes. If Saved is false, the static method WriteAllText from the built-in File class is called and passed the value of FilePath and the value of TextBox.Text, which is the contents of the text editor. This sets the contents of the file at the given file path (which is created if it doesn’t exist yet) to the text in the editor. Saved is set to true, and Tab.Text is reassigned the file name from the file path (this serves two purposes: first it resets the title to remove the unsaved asterisk indicator, and second it changes the title to that of the new file if this was called after a save as prompt). Given the chance that the File static method fails because the file path is invalid, FileEdit throws the exception to the next handler. In FunctionalStudio there isn’t any exception handling for SaveFile or the exception case for the FileEdit constructor as all file path construction is handled by the built-in file dialog feature for Windows, which either returns a valid file path or indicates that none was chosen.

The SaveFileAs method effectively wraps changing the value of FilePath with calling SaveFile. FilePath is assigned the new file path, Saved is set to false, and SaveFile is called. Setting Saved to false is necessary if the file was unedited before saving as and so considered saved, in which case SaveFile wouldn’t actually commit saving to the new location. Logically, assigning false to Saved here makes sense, as though the file may have been saved for its old location, it wouldn’t be saved for its new one.

The Dispose method is one of good practice, allowing for any FileEdit instance to be disposed of which automatically handles its components’ disposal. It calls the Dispose methods of both Tab and TextBox; disposal of a TabPage object actually automatically removes it from the TabControl object it is associated with.

### Highlighting syntax errors

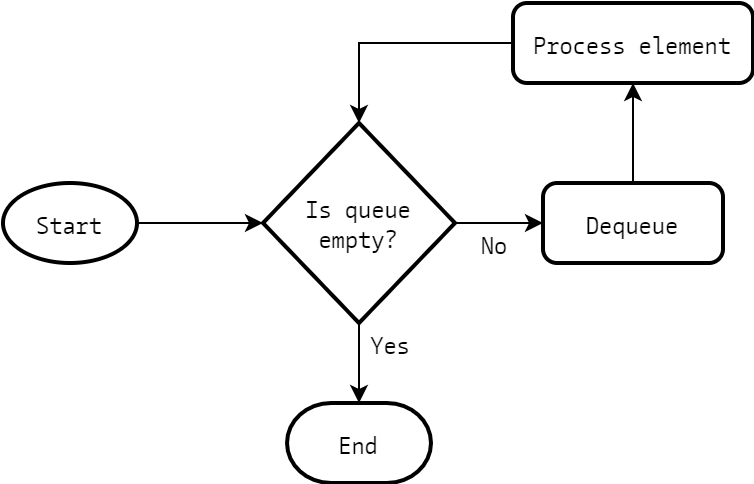
The final public method is HighlightErrors, which takes the return errors of the tokeniser (addressed at …) and appropriately underlines text in a red wave style. This algorithm is complex by virtue of having to use window procedure calls with the imported function SendMessage in order to achieve the desired result with the rich text in the text box, since the basic rich text box functionality doesn’t provide it. Information on the method and the structure used to control the character formatting was found here: [bit.ly/rtbFormatting](http://bit.ly/rtbFormatting) *(Extending RichTextBox - Part II)*

First the CHARFORMAT structure needs to be defined, which is the data structure that pairs with the message EM\_SETCHARFORMAT (integer value 0x0444) to manipulate rich text in controls that support it. The whole message that needs to be sent effectively translates to: “draw a red wavy underline under the selected text”. The CHARFORMAT instance used has to apply a mask to itself so that the control only processes underline information; this is done by assigning CFM\_UNDERLINETYPE (integer value 0x800000) to dwMask. The instance then needs to specify what the underline format is; this is done by assigning the bitwise OR combination of the values for the wave underline style and the red underline colour (0x8 | 0x50) to bUnderlineType. Also the size of the structure needs to be found and handed back to it in cbSize to properly function (a consequence of implementing a C++ function in C# with variably sized arguments).

With the CHARFORMAT instance ready, SendMessage is called and passed the handle of the text box, the message EM\_SETCHARFORMAT to change the rich text, the value of SCF\_SELECTION (integer value 0x1) to indicate applying the format to selected text, and the CHARFORMAT instance to specify the actual formatting. This applies a wave-style red underline to the selected text in the text box.

A very similar window procedure call involves setting the character format to have no underlining, so that any parts of the text currently error underlined get cleared of it first. This behaves the exact same as the call which sets the red underline, but instead the bUnderlineType variable in the CHARFORMAT instance is set to 0x0, which means no underline.

How the error highlighting works is by retrieving the tokeniser errors from the static method GetTokeniserErrors from the class Translator (addressed at …), from which the errors in the source code passed as an argument are returned in a queue. First all underlines are cleared using the window procedure call that removes underline formatting. The errors are then looped through (looping through a stack or queue that you intend to deplete usually functions by popping/dequeueing at the start, and looping until the number of elements is zero) and for each, the start of the selection for the highlighting is set to the index of the error, then the indexer for the source code is incremented until whitespace or the end of the string is reached (only selecting one word. This is where the selection ends, so the selection length is recorded as the number of increments up to that point.

**  
Loop for queue when depleting contents

Once the appropriate text is selected the window procedure call is sent through SendMessage to underline the text in wave-style red; the process is repeated for each error returned.

#### *Suitability for real-time highlighting*

The HighlightErrors method was the first attempt of applying rich text formatting to the text box, and demonstrated that targeted portions of the code could be formatted whatever way appropriate. However, the selection and formatting process was in fact very inefficient and clunky, so much so that you could see the text being highlighted and formatted across the file as it ran. Preventing this yet again went beyond the basic functionality of Windows Forms, as there is no way to stop displaying information in a control without using a window procedure call. The appropriate message in this situation is WM\_SETREDRAW (integer value 0x000b), which sets whether a control draws new changes to itself or not, simply taking a 0 or a 1 to indicate as such. So before the error looping and selection process, the SendMessage call to disable redraw of the text box is processed, and redraw is enabled once all the errors are highlighted. Also, previous selection start and length are recorded before and reassigned after to ensure everything stays the same, and the TextChanged method is unsubscribed from the TextBox.TextChanged event before and re-subscribed after so the changes made while highlighting don’t cause knock-on effects.

Another unfortunate effect of the clunkiness of the rich text formatting was that it made real-time syntactic highlighting incredible user unfriendly. At one point HighlightErrors was implemented to be called after a short delay of no user input, so that errors could be highlighted effectively real-time. However, because of how slow the formatting process was, the method presented significant reduction in feedback and responsiveness, since the entire text box would disable redrawing for a fraction of a second if the user didn’t type for some given amount of time. A partial solution to such an issue would be raise the delay time for the method call, but that would result in simply less responsive highlighting which doesn’t trigger until a user has stopped typing for some more significant amount of time; HighlightErrors is now called at runtime if there were syntax errors caught by the compiler. There lacked a solution that maintained user friendliness by keeping the app as responsive as possible while typing, which made the feature of real-time syntactic highlighting infeasible, and applying such highlighting at runtime like the errors defeats the purpose.

## FunctionalStudio

The FunctionalStudio form always begins with an empty tab control and an empty list of FileEdit instances. Of the IDE functions, the only ones that are enabled are those that create a new file or open a file. A flag called running exists as a variable to indicate whether or not code is being executed, which is used to appropriately configure the form to hold operations i.e. prevent code changes and disallow changing tabs, primarily for the potential implementation of debugging where code may be left running for a long time, and the user may attempt to interact with the form when they shouldn’t.

Which buttons and options should be enabled or disabled when is handled by the method UpdateUI, which is called after any processing occurs that involves changing the state of the form i.e. changes to the FileEdit list, or when code starts or stops running.

* If there are no open files i.e. the list is empty, only the options for creating a new file or opening are file are enabled
* If there are files open i.e. the list is not empty, and no code is running i.e. running is false, then all the options are enabled
* If there are files open i.e. the list is not empty, and there is code running i.e. running is true, then all the options are disabled

### FileEdit implementation – instantiation

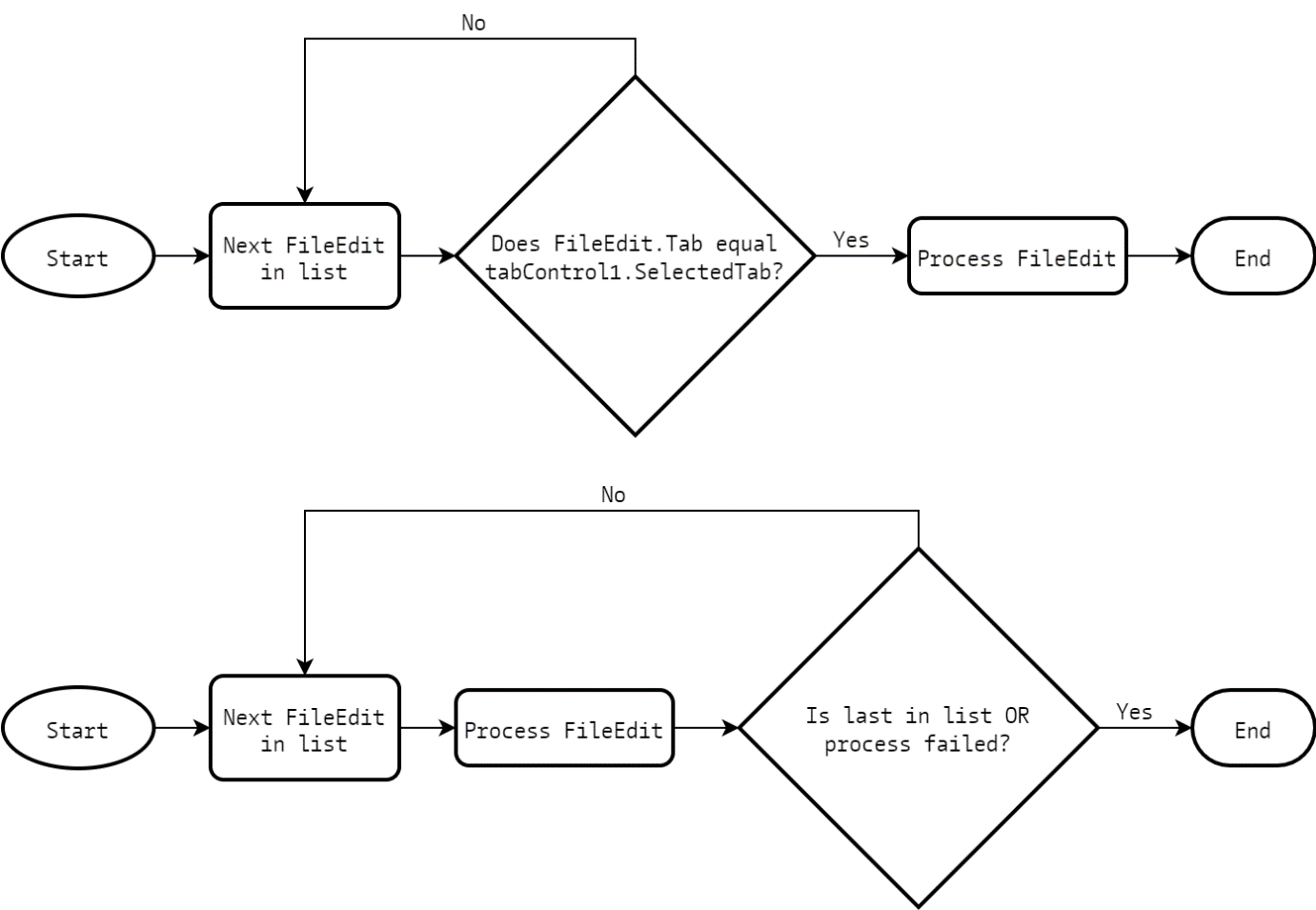
The new file button and the new file option in the File dropdown (Ctrl + N) call the NewFile method. It instantiates a FileEdit object, passing the constructor tabControl1 so it creates a tab for the control in the window. The new FileEdit object is added to the list, and the text box in the tab is auto-focused by selecting the tab from the tab control and then selecting the text box. UpdateUI is called.

The open file button and the open file option in the File dropdown (Ctrl + O) call the OpenFile method. An OpenFileDialog object is instantiated with Title set to “Open file”, Multiselect set to false, Filter set to “Paskell files (\*ps)|\*ps”, and InitialDirectory set to the MyDocuments folder of the user. The resulting dialog will be a file selecting dialog starting from the user’s MyDocuments folder with the title “Open file”, where you can only select one file of type .ps, and indicated in the bottom right is “Paskell file (\*ps)”. The filter behaves by displaying the text before the pipe in the dialog and filtering out files to match the condition after the pipe. The dialog is displayed; if it returns an OK, a new FileEdit object is instantiated, passing the constructor tabControl1 and the file name given by the dialog, it’s added to the list, the text box is auto-focused, and UpdateUI is called. If the dialog doesn’t return an OK, nothing happens. The form is re-enabled.

For both of these cases, the OnResize event handler method in the FileEdit object is subscribed to the Resize event of tabControl1 so when the control is resized such as when maximising the window, the event handler is called. This is primarily used to cue an update in the text box to adjust for size, explained in the custom control section. The handler is subscribed from here rather than from within FileEdit, as when the object is disposed, it cannot track back to the TabControl it had its handler assigned to and unsubscribe from it – as such, whenever trying to resize the window after having closed a file, an attempt to call the method of the disposed object would cause a crash.

### FileEdit implementation – saving and closing

A key process repeated for the remainder of the methods that interact with the FileEdit instances involves selecting the correct instance from the list, or in fact selecting and processing them all. For saving, saving as, and closing, the instance of the FileEdit associated with the tab currently selected must be found by looping through the list until the tab in a given FileEdit object matches that of the selected tab in the tab control before being processed. For saving all and closing all, the list is looped through and each FileEdit object is processed until one process fails i.e. file wasn’t saved or file didn’t close, at which point the loop breaks and the rest aren’t processed.

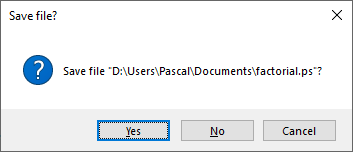
Loop to select correct FileEdit instance, and loop through all FileEdit instances until failure or end

Note that, despite in code the loop not necessarily occurring without elements in the list, logically it is sound here not to consider if the list initially is empty, as the methods that use these algorithms will not be available if no files are open

#### Primary saving and closing methods

The processes which accommodate failure cases for the methods that use the second algorithm are SaveTabFileAs and CloseTabFile. These methods are also used in every other save and close method using the first algorithm where returning failure isn’t necessary.

The SaveTabFileAs method takes a FileEdit object as an argument and instantiates a SaveFileDialog with Title set to “Save file as”, and Multiselect and Filter set to the same as for the OpenFileDialog when instantiated in the OpenFile method. The resulting dialog is much like the dialog for opening a file, however the user writes a file name to save to, or selects an existing one to overwrite. The dialog is displayed and if it returns an OK, the SaveFileAs method of the FileEdit object is called, passing it the file name given by the dialog. For an OK dialog return, the method returns true for success. If the dialog did not return an OK, nothing happens and the method returns false for failure.

The CloseTabFile method takes a FileEdit object as an argument. If the Saved property of the FileEdit object is true, then the object is removed from the list of FileEdit instances and disposed with its OnResize event handler method unsubscribed from tabControl1.Resize, with the method returning true for success. Otherwise the static Show method from the built-in class MessageBox is called, passing two strings for the text and caption of the message box, and the values MessageBoxButtons.YesNoCancel and MessageBoxIcon.Question. The text is set to “Save file \*\*\*?” using the whole file path or “Untitled” if a new file, and the caption is just “Save file?”. The result is a message box titled “Save file?” with a question mark and the question to save the current file, with the options “Yes”, “No”, and “Cancel”. If the message box returns Cancel, nothing happens and the method returns false for failure. If the message box returns Yes, the tab of the given FileEdit object is focused, and the SaveFile method from FunctionalStudio is called (addressed at …). For either a Yes or No return from the message box, the FileEdit object is removed from the list of FileEdit instances and disposed, and the method returns true for success. The reason why the correct tab needs to be focused first is because the SaveFile method functions by processing the currently focused tab, and though when CloseTabFile is called from a direct user interaction the correct tab should already be focused, in the case such as for the CloseAll method described later where all the FileEdit instances are looped through and close, the FileEdit instance intended to be closed may not have its tab focused.

The remaining file interaction methods are direct event handlers of the IDE interaction events and use one of the two looping algorithms specified in the flowchart diagram above. All of them reference either SaveTabFileAs or CloseTabFile.

#### Methods using the first looping algorithm

The save file button and the save file option in the File dropdown (Ctrl + S) call the SaveFile method. The current selected FileEdit instance is found using the first looping algorithm, and if it has been previously saved (indicated by whether or not its FilePath property is empty) then its own SaveFile method implemented in the FileEdit class is called. If it hasn’t been previously saved, SaveTabFileAs is called and passed the FileEdit instance. The result is previously saved files simply get saved, while new files prompt for Save As dialog, which only saves if the user successfully provides a save location.

The save file as option in the File dropdown (Ctrl + Alt + S) calls the SaveFileAs method. This method is equivalent to the SaveFile method above, but calls SaveTabFileAs regardless of if the current file has been saved or not.

The close file option in the File dropdown (Ctrl + W) calls the CloseFile method. The current selected FileEdit instance is found using the first looping algorithm, then CloseTabFile is called and passed the FileEdit instance.

#### Methods using the second looping algorithm

The save all button and the save all option in the File dropdown (Ctrl + Shift + S) call the SaveAll method. Using the second looping algorithm, the same algorithm used in the SaveFile method is processed on each FileEdit instance, the difference being that the return value of the SaveTabFileAs call is used to indicate the process success or failure, and therefore indicates if the loop breaks before ending.

The close button of the FunctionalStudio window calls the CloseAll method. Using the second looping algorithm, CloseTabFile is called and passed each FileEdit instance. The return value of the CloseTabFile call indicates if the process failed, and if it does then the window close event is cancelled and the loop breaks before ending. Important to note is that the list of FileEdit instances is copied to a static array before being looped through, as the CloseTabFile call changes the contents of the list by removing a FileEdit instance; this causes crashes in loops in C# without using a static collection like an array.

### Program execution

The start button and the start option in the Debug dropdown (F5) call the StartProgram method. To start with, running is set to true and UpdateUI is called to configure the UI properly. Using the first looping algorithm used primarily by the saving and closing methods, the FileEdit of the selected tab has its text contents retrieved and its text box disabled, and its HighlightErrors method is called to highlight syntax errors in the code. The output window is cleared to prepare for compile and execution.

The static method Compile from the Translator class is called and passed the source code, and it provides a return state indicating success or failure, and the errors if failure, and it also provides an expression that can be evaluated, the return of which would be taken as the code output. If there are errors, they are looped through and displayed in the output window; for syntax errors, only one error indicating syntax failure is displayed and the errors are visible from the HighlightErrors method previously called.

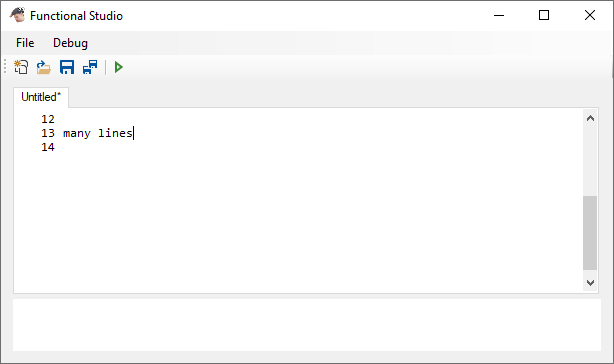
Provided the compilation is successful, the main expression is evaluated (the process of which is addressed at …) and its output is displayed in the output window. If a runtime error is encountered, it is displayed in the output window and it indicates in which function it failed.

The interface needs to be reconfigured for editing so running is reset to false, the text box is re-enabled, and UpdateUI is called again.

As part of execution, not only should the user interface be configured appropriately for execution, but the user should not be able to change tabs either. The method CancelChangeTab handles the Selecting event thrown by tabControl1 when trying to change tab, and forces the event to cancel if running is true, thereby preventing the event from causing an update to the control.

## EditorTextBox

The EditorTextBox class in EditorTextBox.cs is a custom control that provides a Visual Studio style text editor featuring line numbers and scrolling functionality more appropriate for an IDE i.e. the window can be scrolled line by line as far as having the last line of the file be the top line in the window, allowing for effectively a blank canvas of lines on the screen for clarity.



Multi-line Label control

RichTextBox control

VScrollBar

### Design

The control features a multi-line Label control called lineNumbers to indicate line number, a RichTextBox control called textBox which is where the code editing occurs, and a VScrollBar control called vScrollBar which is used for custom scroll functionality for the control. The choice of a label is over a text box is because the user should not be able to interact with the line numbers at all; they simply act as an indicator in the margin and don’t contribute to the editing functionality of the control. The scrollbar is necessary to properly implement the custom scrolling, made clear how it functions later. Both lineNumbers and textBox are kept together using a TableLayoutPanel object container called container, with lineNumbers set to a fixed width while textBox just scales with the rest of the control.

### Pass-through properties, methods, and events

Initially the text editor component of the FileEdit class was just a RichTextBox object, so in building the EditorTextBox class, the custom control was intended to mimic the behaviour of a RichTextBox as closely as possible. Most obviously necessary was an overload Text property to interface with the text from textBox, but also a Font property (which would also match the font of the line numbers), the SelectionStart and SelectionLength properties for manipulating selections (primarily used by the HighlightErrors method), and a custom TextHandle property which provides the control handle of the textBox (necessary for HighlightErrors to pass its messages). Enabled adjusts the ReadOnly property in textBox to disable modifications, which is used when executing code to prevent modifications while running.

Also the relevant events from textBox are subscribed through an EditorTextBox object’s TextChanged, MouseWheel, and KeyDown events (MouseWheel also subscribes to the event in lineNumbers). Finally the SelectAll and TextUpdate methods pass directly on to the SelectAll and Update methods of textBox, and are used in the HighlightErrors method.

In the constructor, any events that could cue a scroll (addressed further on) have handlers subscribed to them; most events are the pass-through events for textBox, and the Scroll event of vScrollBar subscribed to directly by the primary scrolling method ScrollTextBox, which is addressed below.

### Custom scrolling

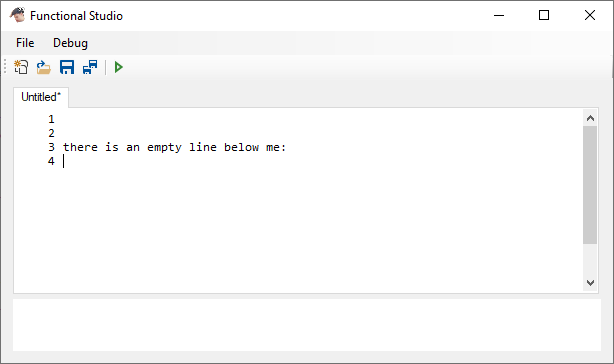
The default scroll functionality of controls in Windows Forms is unfortunately very limited and rather frustratingly on-rails. One potential solution was to implement a window procedure call to force scroll the text box to a desired point, but this kind of solution pushed the API to unnatural places, and the controls would behave in glitch-like, undefined ways. Likely the most difficult component of such a solution was aligning the scrolling of both lineNumbers and textBox, but regardless the solution was ugly, flickering at any given scroll, and prone to try to reset itself to some default scrolling state where the scroll bar only scaled how it liked.

The final solution ended up being effectively a fake scroll, simply adding a separate custom scroll bar vScrollBar whose properties could be changed comfortably at will, and scrolling the control by actually shifting the entirety of container up and down. This provided much more control over how the scrolling behaved, which allowed for creating a scrolling solution that was pleasant, practical, and smooth.

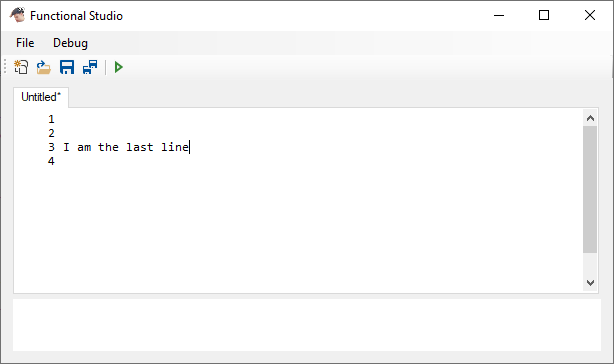
#### Line numbering and configuring the scroll bar

Firstly it is important that the line numbers are clear and useful so that position in a file can be easily tracked. I intended for there to be as many line numbers as there are lines, plus one more if the last line has code in it. This means:

* If the final line is blank i.e. there is no code and there is only one empty line, or the final character in the editor is a newline character, line numbers should only go up to that final line:



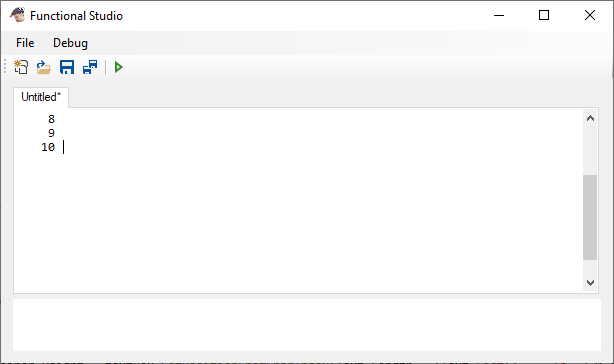
* If the final line contains code, line numbers should go up to one line beyond that final line:



Visually the intended effect is to always have one more line number displayed below the last line of written code; the distinction is important to make however to properly identify what that should be. For example, if the code contains multiply blank lines, the line numbers must still go down to that bottom empty line, and no further.

The desired effect is achieved by writing as many line numbers as there are lines in the editor, as given by textBox.Lines.Length. If there are no lines i.e. a blank file, or the line character was a newline which indicates the final line being empty, then another line number is appended.

Configuring vScrollBar is in fact a rather peculiar combination of property adjustments, as the scroll bar feature in Windows Forms appears to be rather poorly implemented and even more poorly documented. The first point to note is how the physical scrollbar interaction controls the Value property of vScrollBar. The interplay between the Maximum, Minimum, and LargeChange properties define how the scrollbar actually looks in a form: the ratio of LargeChange to the difference between Maximum and Minimum defines the proportional size of the scrollbar thumb in the form, but not quite exactly.



Maximum - Minimum

Minimum is normally 0, therefore this value is normally just Maximum

LargeChange - 1, oddly enough

Though understandable the relationship between LargeChange and Maximum (logically if LargeChange represents a page up or down change, as I have implemented it, then the proportion of the scroll thumb in the bar matches that of the page size within a whole document), it is unclear, and apparently undocumented, as to why the thumb size had to have the proportion of LargeChange *minus 1* to the difference in Maximum and Minimum, rather than just LargeChange. Worse yet, it turns out that Maximum does not in fact indicate the maximum value that the scrollbar can reach; the maximum value is given by how far down in the bar the thumb can reach, which due to the odd thumb size in relation to LargeChange causes the actual Maximum value reachable by the scrollbar to be Maximum - Minimum - LargeChange + 1. Official documentation on the issue appears to be none-existent. Only a StackOverflow thread was of use: [bit.ly/vsbMaximum](http://bit.ly/vsbMaximum) *(.NET Vertical Scrollbar not Respecting Maximum Property)*

*After some research, I've found that a scroll bar can only go up to its maximum minus the size of the scrollbar's slider.*

*And the size of the slider appears to be equal to (LargeChange - 1).*

*Doesn't seem very intuitive to me but there you go.*

Thankfully the configuration of the scrollbar properties, including its peculiar implementation of LargeChange, happen to align well with the vision for the custom control. The entirety of the scrollable contents of the editor can match the maximum scroll of the bar, the large thumb size as associated with LargeChange can represent the page size of the editor, and then the limitation of the scrollbar up to its strange true maximum can indicate only being able to scroll up to the final line in the code, where the remaining blank page made available matches the remaining thumb-sized gap in the scrollbar beyond where it has to stop.

In the UpdateLineNumbers method, where also the contents of lineNumbers is updated as previously described, vScrollBar is configured to match with the current size of the document and the size of container is adjusted, as is explained afterwards. If there are fewer than two lines in the document, the scrollbar is not enabled. If there are two or more lines in the document, then it is enabled, and the properties of vScrollBar are set as follows:

* Maximum is set to textBox.Lines.Length - 2 + Height / Font.Height
* SmallChange is set to Math.Min(3, textBox.Lines.Length)
* LargeChange is set to Height / Font.Height

It is worth noting that the division is an integer division and so introduces rounding, though this isn’t an issue for these values in particular. The first thing to note is the scale; the value 1 in all these properties indicate one line in the editor. LargeChange is set to the number of whole lines that can be shown in the editor. SmallChange is either 3, or however many lines are actually in the editor if there are fewer than 3. Minimum isn’t changed, so remains always at its default 0 value, and most importantly Maximum is set to the number of lines in the editor plus the whole numbers of lines that the editor can show, minus 2.

This oddly specific manipulation of the value accounts for two things. Firstly, given the amount of lines visible in the text editor, you are supposed to only scroll past that number of lines minus one; in order for the last line to be visible, you should not be able to scroll far enough to go past that line, hence the minus one. Secondly, considering the previous note on how the practical maximum that can be reached by scrolling in the form is actually Maximum - Minimum - LargeChange + 1, then the actual value of Maximum needs to be set to the maximum you intend to be able to reach, in this case textBox.Lines.Length - 1, plus the Minimum, in this case 0, plus LargeChange, in this case Height / Font.Height, minus another 1. The result is that the true value of Maximum must be then set to textBox.Lines.Length - 2 + LargeChange, where we replace LargeChange here with what we set it to later for good measure (Windows Forms has a tendency to not set control properties to certain values unless other values have been considered first, in this case needing Maximum to be set before LargeChange).

The result is a scrollbar with a thumb proportionally sized with the ratio of the window size to the size of the whole document, which grows and shrinks in a manner that is smooth and intuitively makes sense.

#### Applying the scroll to the container

With the scrollbar configured, it is important then to consider what happens to the editor itself. Its size is associated with the size of container, as both the actual editor and the line numbers control are set to fill container and so scale with it as well. An interesting consequence of configuring control sizes, font sizes, scrollbar sizes and the such using integer values is that those values are not perfectly accurate and do not correctly scale once stacked upon each other several times over. Using the value of Font.Height to dictate scrolling distance in fact leads to gradual drift over several lines, to the point where by 30 odd lines of code, the last line has gone off the top of the screen if scrolled to the very bottom. It is therefore necessary to appropriately control the scrolling by consulting textBox directly for the location of its lines, which is done by using its method GetPosFromCharIndex.

The other need for this method is to appropriately adjust the size of container for however many lines it contains. Though it would require the unlikely amount of lines in the tens or hundreds, calculating the amount of space that the lines of code in the editor would take up using a multiple of Font.Height would eventually lead to a noticeable discrepancy in size of the container. Though an extreme consideration that so many lines could be written, it is sensible to attempt to solve; in fact, by doing so the whole problem can be addressed using the now rigorously accurate size of container to control the scaling and positioning of the scrolling itself.

By using GetPosFromCharIndex in textBox to find the location of the lowest line of code, container can be appropriately sized to fit every line of code plus another window of height, given by the value of Height (here Height, which is the height of the EditorTextBox instance in the form, is added so that the remaining window’s worth of space at the bottom of the editor is still considered part of the control). GetPosFromCharIndex has a curious property of being able to index one position beyond the last character in the text box, allowing it to be passed Text.Length. What this does is enable us to consider where the next character would be if typed, which is in effect where the caret is displayed. The result is that if the last character was a newline character, then the result for the index position one after that would return a position one line lower and reset to the start of the line, rather than the next position on the same line that had just been typed on.

Then actually scrolling the text box would involve correctly position container such that it appears as though it has been scrolled to a certain point. Encapsulated in ScrollTextBox, the scroll is applied by setting the Location property of container such that the Y coordinate is negated appropriately for a scrolled effect. Properly processing the values here is important, given that many of the properties used are discrete values and present rounding errors when used in division. The desired effect is to convert the number of lines scrolled as indicated by vScrollBar.Value into a coordinate value that correctly scales with the size of container. This is achieved by multiplying the value given by the scrollbar by the ratio between the true location of the furthest point that can be scrolled to and the maximum value that the scrollbar can reach. The true location of the furthest point that can be scrolled to is simply given by container.Height minus the height of the window since that cannot be scrolled past; and the true maximum value the scrollbar can reach is given by the previous observation from the StackOverflow thread of Maximum - Minimum - LargeChange + 1. The result is in fact always textBox.Lines.Count - 1, however it has been written as the custom property ScrollMax in the class, and properly uses the values of Maximum, Minimum, and LargeChange for clarity’s sake. The custom property is also used in other places to extend the scrolling functionality.

#### Additional scroll cues

ScrollTextBox is the driver for any scrolling in an EditorTextBox control, and is used for every other scroll implementation as part of the control.

Changes to the text in textBox are handled by the OnTextChanged method. UpdateLineNumbers is called to adjust line count and enable/disable vScrollBar if necessary, and ScrollToLine (addressed further down) is called and passed the line index of the caret position to scroll if necessary.

Mouse wheel scrolling over the EditorTextBox control is handled by the OnMouseWheel method. If vScrollBar is enabled, then the scroll event is effectively passed on to vScrollBar and its own OnMouseWheel method is triggered. This has the effect of treating mouse wheel scrolling anywhere in the control as equivalent to mouse wheel scrolling over the scrollbar itself, which is an expected feature of any scrollable control. Since OnMouseWheel is in fact a protected method, it needs to be invoked using reflection; this is done rather than attempting to mimic the scroll so that it is consistent between scrolling over the EditorTextBox control and over vScrollBar.

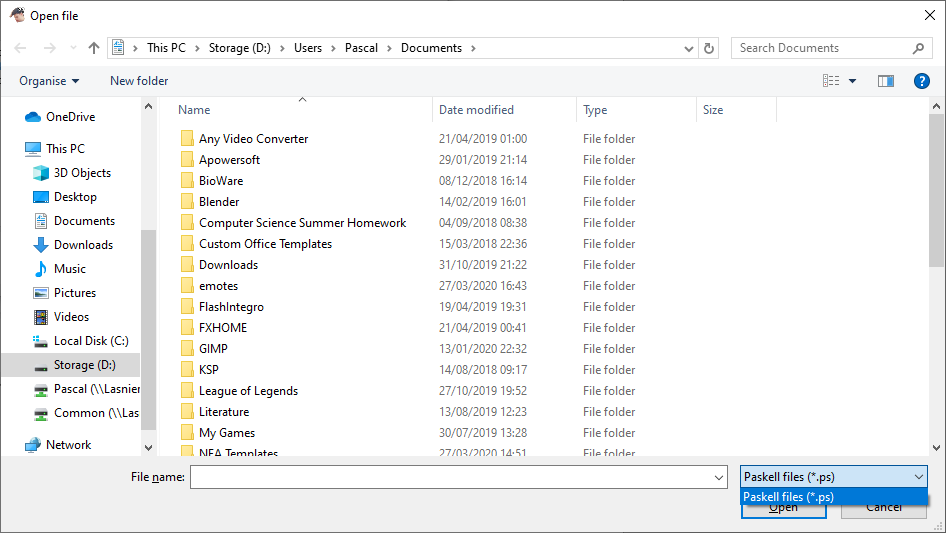
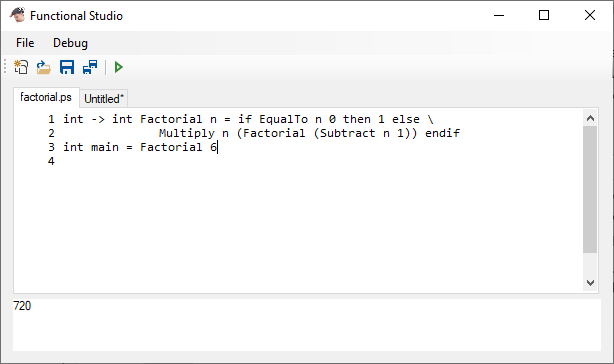
KeyDown events in textBox are handled by the OnKeyDown method. For Page Up and Page Down keys, the Value property in vScrollBar is incremented or decremented by a factor of vScrollBar.LargeChange, but not set beyond ScrollMin or ScrollMax, as controlled by the Math.Min and Math.Max methods. The ScrollTextBox method is then called to update the control appropriately, and the event object e has its Handled property set to true, which prevents it being handled by the control regularly. This has the effect of scrolling up or down one window’s height at a time, only up to as far as can be scrolled normally. For the arrow keys, the line index of which line the caret in the text box would next be is considered and passed to the ScrollToLine method, again using Math.Min and Math.Max to ensure not attempting to scroll beyond the boundaries. Initially the intended solution for the arrow keys was to simply call the ScrollToLine method and pass it the line on which the caret is on. However, the event is always processed by custom event handlers first before being regularly processed by the form, which meant that the method attempted to scroll, then the actual press of the arrow keys took effect, changing the caret position. There was no way to force a different order of process, and so the effect has to be considered within this method first and then the event needs to be left to be handled again by the text box (hence why for the arrow keys, the Handled property for e isn’t changed).

For considering scrolling to a particular line when a change occurs in textBox, the ScrollToLine method is called by the event handlers that process such changes. It checks whether the given line is either above or below the visible boundaries of the control, which is does by taking the difference between the line number and the current value of vScrollBar.Value and comparing it to the boundaries, and then adjusts the Value property of vScrollBar appropriately such that the line would be made just visible. ScrollTextBox is then called to process the change.

## Resulting Graphical User Interface

The culmination of all the components of the UI is a robust and fluid IDE-style programming environment. Though limited, all options for interacting with the app are provided in the toolbar, menu dropdowns, and keyboard shortcuts. These directly interact with the underlying classes that encapsulate an idealistic code editing interface, featuring a line numbered editor window with custom scrolling of which multiply instances can be made in tabs. All instances can be opened, closed, saved, created, and have their code run to produce meaningful output.

The interface design was intended to make user interaction as responsive and easy-to-use as possible, and required some compromise such as sacrificing the real-time syntactic highlighting that was intended to be included as part of the app. The custom-implemented scrolling solution is very smooth and consistent, and generally writing code in the editor feels responsive. File access, including saving and opening functionality, is as standard for Windows and makes the app feel like a fitting addition to any Windows software library.



The majority of the power of the application however resides in its program execution capabilities, which it interacts with through the StartProgram method as called from the Start button in the toolbar and Debug menu.

# Function Programming Paradigm Runtime Environment

## Defining a function

Though its potential can be deep and its applications complex, functional programming is grounded on very simple principles defining how functions should be defined and behave, and how they are called. Some functions can be multi-parameter, highly complex expressions, and others more moderately composed. However, every single function that can be defined falls under the same categorisation in this paradigm: a function takes an argument, just one single argument of a particular type, and returns *something*. This something could be anything; it could be an integer or a float, it could be yet another function that will take another parameter and produce yet another return value. This is the backbone of functional programming, that every single function always takes one thing and returns another; what you do with return is yours to decide (or specify).

From this can be considered how multi-argument functions can be defined. In the traditional sense of one, you can expect a function to take one, two, or more arguments which all fit some standard data type, and it should then return back a result like a number which can be made useful somewhere else. Take for example multiplication; it can be defined as a two parameter function which returns the product of two numbers. How this would fit in the functional programming frame is that a “multiply” function takes a number as an argument, and returns back another function. This new returned function then can take another number as an argument, and finally returns back a final number, which in this case would be the product of the two total arguments passed. This is how any function can be classified; when you call a function to produce a result and pass it multiple arguments, you are in fact passing a function one argument, then the result of that function – a new function – another argument again, which finally produces a useful result.

First function call

Second function call

This frame of thought prompts a key consideration of how functions in the runtime environment should have their function type defined: every single function can be defined by what type of argument it takes and what type of result it returns. From this a simple class type could be defined, one which indicates parameter and return, which both can hold a type again represented by the same class. A base case must be introduced somewhere, so the class should also be able to represent a base type like int, bool etc. This class is what would enable functions and variables alike to be defined in a universal way, and such generalisation would enable indiscriminate consideration of the type of any expression when evaluating it.

### TypeSignature – defining expression type

Implementation of expression type definition

Hollow triangle-head arrows indicate generalisation i.e. Function Type and Variable Type can be generalised as just Type, so a Type could be either of these classifications

This general classification is encapsulated in the class TypeSignature in Runtime.cs. What this class practically represents is the type definition of functions and variables in a program, so for example the signature for describing a function which takes an integer as an argument, and returns an integer as a result e.g. a factorial function. From here onwards, such a type description for any expression, so either a function or a variable, will be called a *type signature*, matching the class name.

How the TypeSignature class represents the type of an expression is by using a bool flag IsFunction to distinguish between a function type and a variable type. If it’s a function type, the two TypeSignature properties Parameter and Return are used to indicate what type the function takes and what type it returns. If it’s a variable type, the built-in class type Type property appropriately called Type is used to indicate the type of the variable using C# type definition. The effect of the structure of TypeSignature is essentially a recursively defined signature for expressions in code, where the base case is a variable type TypeSignature object. There is in fact a condition where IsFunction is false, so indicating the expression is a variable, but Type remains undefined; this represents a generic type for some variable which is primarily used to describe some base functions for the language (as is addressed later), which allows some functions to be able to accept various types. The added flexibility for these base functions loosens the restrictions of the compiler for type matching, however this introduces less safety in code execution as if invalid variable types are passed the runtime environment will break. There exist constructors for all of these possible configurations of TypeSignature.

The implication of such a frame of reference is that more complex signatures such as , for functions such as addition and multiplication, actually represent a nested combination of function mapping from on type to the next. In fact, that type signature is a simpler version of the more detailed full type signature . Natural order of evaluation for a type signature, which is always to take the first function map symbol () that is not nested and assign parameter and return type whatever is on either side, means that the right side doesn’t need to be enclosed in brackets to specify it as a return type. Conversely, if instead using the signature , which represents a function taking another function of type as an argument and giving a result, then the brackets are necessary to indicate the parameter type being that function type.

Considering this, TypeSignature has a ToString method which enables a TypeSignature object to be converted back into a written format of the signature with minimal bracket nesting for clarity. Using the ternary operator as small condition statements, a type signature can be decomposed from a TypeSignature object: if it is a function type, it draws enclosing brackets around it if specified and recursively calls the ToString method for Parameter and Return to write around a function map “->” string, where the parameter type is specified to write enclosing brackets, and the return type is specified to not write enclosing brackets; if it is a variable type, it displays the name of the type associated with Type and ignores any specification of enclosing brackets. The result is a string that conforms to the type signature writing format as specified above. The proper name for a type in the string is provided by the string value of the OperandType enum values, which is retrieved using a Type object and passing it to GetTypeString; by looping through the enum values and finding which one is associated with the C# type provided, its name is provided as a string and matches that which would be used by the user when writing code in this app. OperandType and its association with C# types is addressed at …

### Generalising expressions

With the structure of a type signature now defined, it becomes practical to be able to assign type signatures to actual entities in order to identify them and how they should be treated. Yet again some general classification is necessary to simplify execution; from the generalisation of function and variable types into a single classification TypeSignature, it is appropriate to be able to represent both functions and variables in one type: an expression. The encapsulating class is that of PExpression, and henceforth *expression* is the umbrella term for any entity in a program, including functions and variables.

The powerful concept of functions in this paradigm being single-parameter functions which return other expressions, which could be functions further applied, is well represented in the implementation of expression type signatures and fits well with the syntax of evaluating functions in a functional programming language. However, multi-parameter functions are still best managed as whole, contained entities, though there doesn’t have to be any sacrifice made to the purity of the paradigm. Once passed all the arguments, these functions can evaluate the expression that compose them.

#### Implementation of functions

As an example for how such functions could arise:

It’s important to note that despite being a variable-type expression, it doesn’t necessarily have a value assigned to it by default; *just because something is a variable doesn’t mean it is evaluated*. In order to evaluate , the return value of must be found, which means applying those two arguments to in that order. Ignoring how would be implemented, you can visualise even from reading it the pure functional sequence to entail: is passed , making a new function which can just be represented aswith the type signature . is then passed , and magically a variable of type is returned, assigned to , and finally it has been evaluated.

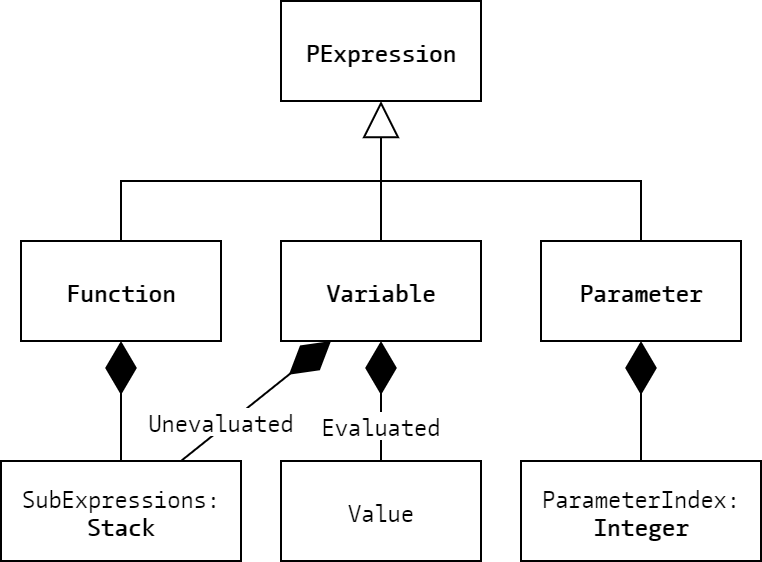
Despite the beauty of the concept, this is still a program on a computer and the code still needs to be run. Practically speaking no function can be evaluated, wholly considered or not, without being passed all of its arguments. The effective way to evaluate such a sequence of expressions as make up the definition of involves parameterising the function, argument by argument, until totally parameterised such that it can then be evaluated and a real result be given i.e. first passed, the instances of the parameter are replaced by ; then passed , the instances of the parameter are replaced by ; then finally the arguments and are processed and is evaluated to a proper value. By no means does such implementation challenge the fundamental basis of function evaluation; the partially parameterised is still in effect the returned function from , with the new type signature . All this does is provide a practical framework for how functions get evaluated, argument by argument.

This consideration of functions practically applied leads to other observations of how functions must be implemented in a runtime environment. The first observation is that so long as an expression is still a function, it has parameters that need to be parameterised. Truly evaluating a function involves passing it arguments of appropriate type until it finally becomes an expression with a variable type and a proper value can be found. The second observation is that any user-defined function or expression, as would be represented by some PExpression object in code, must carry with it its entire definition as matches what follows the equals sign, including whatever parameters it has left to replace in its contents in order for it to eventually be evaluated once parameterised.

The implication of this is that there must be a way to represent the definition of an expression, including the parameters as part of it if it is a function. Parameters in a definition must behave as placeholder entities; they don’t hold a value or represent a function, but they must exist to indicate where a real expression must go and what it must look like i.e. in what part of a function definition they lie (which may be multiple places).

#### Parameters and expression definitions

Firstly there must be a way to represent parameter instances within function definitions such that they can be identified and replaced appropriately with arguments when a function is passed them. These parameter instances must sit amongst other real expressions, so it is sensible to further generalise the PExpression class to encapsulate parameters also. Within the description of a parameter should be information regarding whether for any argument passed the parameter is meant to be replaced; this effectively, for any function with some given amount of parameters, is what order they are in when the function is declared, and so in what order parameter instances get replaced by arguments passed. This can be best represented using a sort of index value.

Addressing the need to describe the definition of an unevaluated expression in some way, the most appropriate solution would likely be a stack of the expressions that compose it i.e. its *subexpressions*. Choosing a stack over any other dynamic array data type would enable Reverse Polish Notation evaluation, which would be appropriate for such a structure.

Implementation of expression

From this, instances of PExpression can be specialised to fit one of three (extended to four) primary expression types:

* Functions, which are defined by their subexpressions which contain parameters
* Variables, which are either evaluated and hold an absolute value which can be obtained, or are unevaluated and are defined by their subexpressions, though they contain no parameters
* Parameters, which indicate within subexpressions where arguments passed will be placed, and have an index to specify which argument passed to a function they are for

#### State of an expression

The distinction between evaluated and unevaluated variables for PExpression in fact associates further with a different type of distinction to be made with expressions. Initially when compiled, all expressions are defined as set objects and, for functions or unevaluated variables, are often defined in terms of each other. Left ignored, this introduces a huge obstacle with code execution wherein attempting to evaluate an expression in one place, either by trying to evaluate an unevaluated variable or parameterise a function, causes those changes to reflect everywhere else that the expression is referenced, and so modify its behaviour in places where it wasn’t supposed to. Other than being a violation of the qualities of functional programming that define it – statelessness and immutability – it prevents expressions being used in multiple places where they may be treated differently i.e. a function being passed different arguments. To this extent, the issue would first be that if a function has already been evaluated somewhere else, you may end up trying to pass arguments to a now-variable, which wouldn’t be correct.

The solution to this is to distinguish the definitions of expressions and explicitly prevent their direct modification; any expression that is about to be worked on must first be cloned if identified as a definition. The distinction of an unevaluated variable from an evaluated one relates to this in that an expression that has been cloned to be worked on is then *being* *evaluated*, and so needs to be represented as such. An unevaluated variable needing to be evaluated would have at one point begun as a definition, became an expression that was being worked on, and once evaluated is evaluated totally, not needing anymore work. Therefore, any expression can lie in any of the three categories of: being a definition, being a worked expression, or being evaluated. These are encapsulated in the PExpressionState enum and its instance in PExpression objects as the variable state; it has the potential values PExpressionState.Defined, PExpressionState.WorkedExpression, and PExpressionState.Evaluated.

The default value of state is PExpression.Defined, as is set in most of the constructors. The exception to this is when constructing a PExpression object representing an evaluated variable, in which case state is to PExpression.Evaluated (this occurs primarily in the compiler, addressed at …) For this constructor, the Value property representing the value of the variable is set to a dynamic value argument (dynamic indicates that the type of the variable is evaluated at runtime, part of C#, allowing any type to be passed and have a dynamic variable represent it) and TypeSignature is set to the type signature representing the variable type. For the remaining constructors with state as PExpression.Defined:

* For a function or unevaluated variable, TypeSignature is assigned the type signature passed, and SubExpressions, which holds the subexpressions of the expression, is initialised as a new empty stack.
* For a function parameter, TypeSignature is assigned the type signature passed, parameterIndex, which indicates which position of an argument the parameter holds its place for, is assigned the index passed, and isParameter, which is the flag indicating the expression is a parameter, is set to true.
* A base function expression instance is defined, which is addressed below.

For all of these constructors, a string can be optionally passed to assign an identifier to the expression, stored in the Identifier property which is defaulted to a blank string.

#### Defining functions – base expressions

As every function is defined in terms of other functions, there must lie a base case to the references, which must include all of the primary functionalities of a programming language. These are the *base expressions*, whose evaluations are beyond the boundaries of the functional programming environment, and instead reference implementation within the runtime environment itself. Using the built-in Func class, it is possible to assign a function as a Func object of a given type to a variable within PExpression and simply call that function from within the class like any other. For the constructor for PExpression objects representing base expressions: TypeSignature is assigned the type signature passed; the function variable, which is what the PExpression object will use to evaluate itself once parameterised, is assigned the function passed; the isFunction variable, which is the flag indicating that the expression is a base expression, is set to true; and the arguments variable, which is the queue where arguments passed are kept, is initialised as a new empty queue.

Functions assigned to a base expression must take a queue of PExpression objects and return a PExpression as a result. They operate on the assumption that all the arguments they use are variables, so attempt to retrieve the Value property of the results of the arguments’ Evaluate call (addressed at …). The arguments are all dequeued from the queue the function is passed, and are used to produce a new evaluated variable PExpression object with its Value property set to the result of the operation.

#### Defining functions – user expressions

When initialising a user-defined function or unevaluated variable, the SubExpressions stack is where the definition resides, and is where the subexpressions as defined by the compiler would be pushed to. Elements in the stack aren’t just the subexpressions themselves, but in fact are nested tuples providing information for execution (the uses of which are addressed at …). Within the stack are, importantly, any of the parameter instances that make up the function being defined. A public PushSubExpression method allows external components such as the compiler to interface with the stack and push all the subexpressions that make up the definition onto the stack, including the parameters.

## Executing functional code

Given the generalised classification of *expressions* in a functional environment, such objects should be able to be evaluated and have their results made useful. For evaluated variables, identified by their state variable holding the enum value PExpressionState.Evaluated, the Evaluate method only returns themselves, where Value is already defined and available to be used. However, the majority of expressions that would be defined would either be unevaluated variables that need to be processed, or functions that need to be parameterised. Before directly working on such expressions, it is important to remember the rule that *definitions* cannot be worked on, and therefore need to be cloned to be made useable.

The CloneWorkedExpression method serves as the most appropriate way to clone the definition of an expression such that a workable instance of that expression can be used to make evaluations. If the expression has a state variable of value of either PExpressionState.Evaluated or PExpressionState.WorkedExpression, then it simply returns the same expression back, as it doesn’t need cloning. In the case of it being PExpressionState.Definition however, then a cloning process entails.

If the expression is either a parameter or a base expression, a duplicate is created simply by calling the appropriate PExpression constructor and passing on the defining arguments, so the type signature, the expression identifier, and then either the parameter index for a parameter or the function for a base expression.

If the expression doesn’t match any of these cases however, then it must either be a function or an unevaluated variable, and as such contains a subexpressions stack that needs to be cloned also. Contained within the nested tuples that make up the elements of the stack are a PExpression object and a stack, both of which need to be properly cloned to ensure there are not multiple instances of either when being evaluated. To clone the stack, the Stack constructor taking a stack as an argument needs to called, but twice – this is because the constructor flips the stack when copying it so the process needs to be repeated to maintain the same orientation. For the PExpression object, it should only be cloned – which is done by recursively calling CloneWorkedExpression on the object – if it is *not* a definition. This is because trying to clone any function definition that calls itself recursively and then clone the definition expression of itself within its subexpressions would create a baseless case of cloning and cause the program to crash. An expression should only be cloned when it is known that it will be worked on i.e. when trying to evaluate it. As such, any definition expression within the subexpressions stack will naturally clone itself when necessary.

Popping all of the stack elements onto a temporary one, the components of the nested tuple are cloned, the new PExpression object and stack are placed back into a nested tuple, and the new tuple is pushed onto the new subexpressions stack. The elements from the original definition get placed back onto their stack in order, and the new PExpression object now has its cloned subexpressions ready to be used freely. A final important process is to set the state variable of the new expression to PExpressionState.WorkedExpression, so that it doesn’t get cloned again.

### Evaluating functions – base expression evaluation

Given that any expression can be manipulated without side-effects, provided CloneWorkedExpression is used, it becomes possible to evaluate expressions. For a base expression, the overload of Evaluate with a PExpression parameter (which is effectively the method to parameterise a function) uses the arguments queue to store arguments passed to the expression. For every argument passed, the argument is enqueued to queue and the type signature of the expression is updated to account for the parameterisation – it is assigned to its own return type signature.

Once the type signature indicates that the expression is no longer a function, so the function has been fully parameterised, the function as given by the function variable is called, passing it the queue of arguments. Its resulting expression is assigned the identifier of the original expression, its state variable is set to PExpression.Evaluated, and it is returned. Any error thrown by the function is caught here and returns a runtime exception specifying that the base expression failed; any kind of failure here would likely be due to the passing of invalid variable types, which many of the base expressions are susceptible to because of the frequency of generic types in their type signatures allowing for any type to make it through the compiler and attempt to be passed at runtime.

### Evaluating functions – parameterising user-defined functions

In the context of a user-defined function rather than a base expression, arguments are not kept in a separate stack and must instead be appropriately placed into the SubExpressions stack for the parameters they must replace. The overload of Evaluate with a PExpression parameter, in the case of parameterising a user-defined function, every subexpression is cycled through (using a temporary stack to transfer from SubExpressions), and checked if it is a parameter. If it is, it is first cloned; it is important for this to occur here as parameter expressions don’t actually get any other opportunity to be cloned, and without it any changes made to the object, in this case its parameter index would reflect in every instance of that parameter elsewhere, an undesirable effect. Then, if the parameterIndex variable of the parameter is 0, the PExpression object is replaced with the argument passed to Evaluate. If the parameterIndex variable is not 0, then it is decremented.

This has the effect of parameterising the function for the current argument passed, replacing every parameter instance whose index was 0 with the argument, and shifting the remaining parameter indices such that the next parameter instances get replaced by the next argument, and so on. The expression is update to account for the parameterisation, so it is assigned its own return type signature.

Once the type signature indicates that the expression is no longer a function, so the function has been fully parameterised, the Evaluate method overload without parameters is called to work on the subexpressions stack and evaluate the sequence to produce a proper value. The resulting evaluated expression is returned.

### Evaluating expressions

For any unevaluated variable expression that exists, either as defined by the user or produced when a user-defined function is totally parameterised, the subexpressions that compose it are required to be evaluated. This encompasses the concept of how expressions are defined and implemented, as written (like described at …), so how they are composed of single-argument function passed an argument, to return another function which is passed the next argument, and so on.

The implementation of subexpression evaluation could easily be imagined simply as having a base subexpression (not to be confused with base expression) which defines the primary function process, and then passing the necessary arguments to it until none remain. For any simple function definition, that would suffice.

However, not all function evaluation follows such simple process. Before considering more complex arrangements, it is first necessary to examine the implication of the structure of the expression definition. Simply considering a definition structure such as – ignoring the considerations of and potentially as parameters – what does such a sequence of expressions mean? Were it not known that is a function taking two arguments, that and are expressions of type appropriate for the parameters of such a function, then how would it be evaluated?

Though strange to consider at first, it is in fact totally correct to operate on the premise of *you just give what’s on the right to what’s on the left*. By human-reads-words logic this totally makes sense: what else in that definition, knowing there exists in it a function, would you consider the function? Furthermore, why else *wouldn’t* be the function; why would you try to pass something to it if it weren’t? This consideration is not just an unexplainable bout of common sense and in fact entirely defines expression evaluation logic.

From here it becomes appropriate to define a sequence of expressions such as like a kind of clause, which is in effect just an expression definition, and these clauses always begin with a function. Furthering this consideration, clauses can be made nested within others using, most appropriately, brackets: would you want to pass the result of a clause to a function, or the components that the clause is comprised of? For example, the distinction between and . The first appears to be passing the arguments and , whereas the second appears to be passing the result of passing the argument . Clauses – whose shapes would be defined by entire expression definitions, bracket clauses, and likely other types of syntax which could be considered – are the natural form of structuring expressions to be evaluated. Every clause could effectively be collapsed into a single expression by evaluating it, which requires considering the base subexpression – effectively the function at its start – and its arguments.

Encapsulating a long, multi-argument sequence of expressions such into a single clause is logical, in much similar a way as how long type signatures can be written as un-nested sequential parameter types up to a final return. Consider instead that clauses can only be a single function expression and its argument – this is taking the premise of functional programming back to its purest form. The expression definition would then become to accommodate this requirement. It then becomes evident that by natural evaluation of the terms (which is from left to right, to clarify) that this nesting of clauses is equivalent to simply considering the longer chain of arguments for in only one clause.

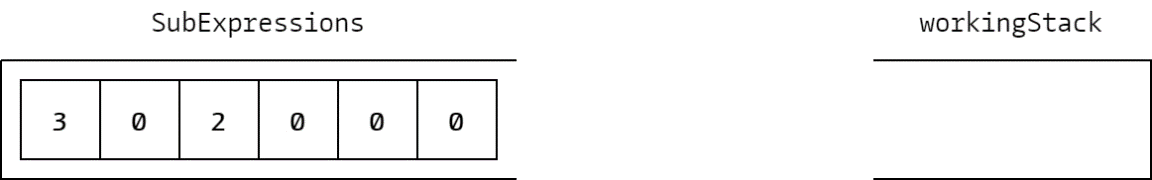
These clause and nesting considerations mostly apply to the compiler which is addressed later, but considering them now provides a better understanding of how such evaluation would actually occur at runtime; it should be reminiscent of writing functions nested in an algebraic calculator. It is well known that there exists a more efficient and memory-friendly application of such calculations, and this functional programming evaluation concept’s similarity to it allows for the valuable conclusion: subexpressions should be evaluated using Reverse Polish Notation implementation.

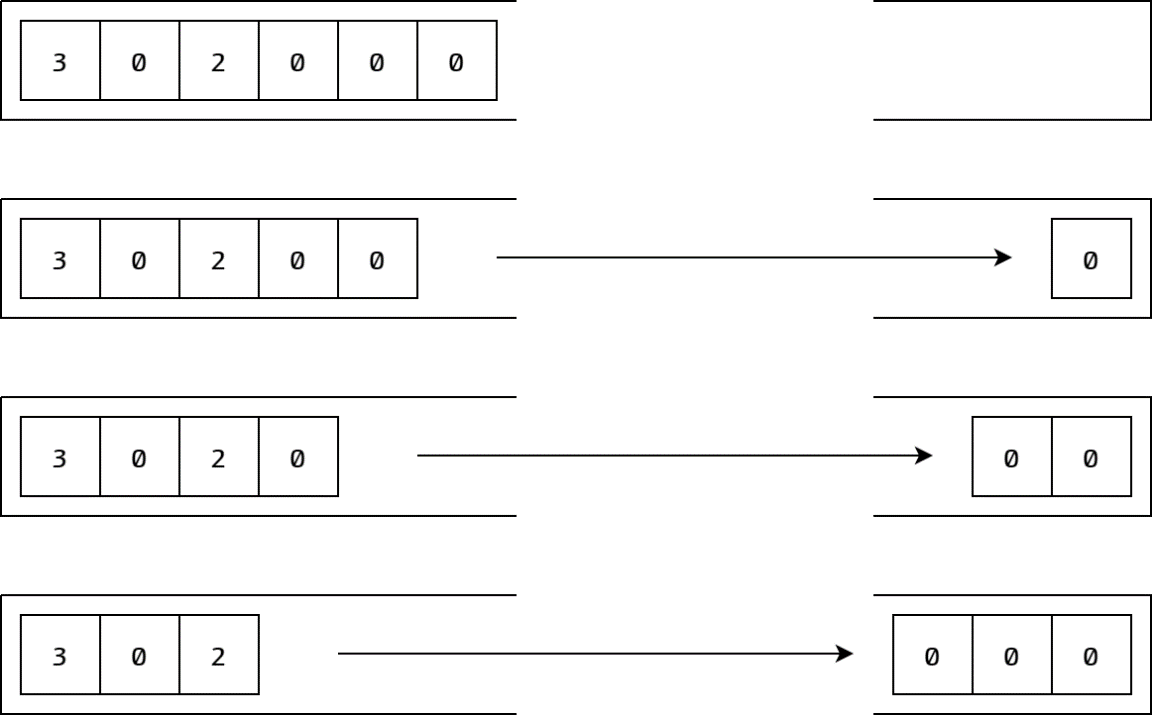
#### Evaluating SubExpressions in RPN

The reason why SubExpressions is declared as a stack is in fact to accommodate for the use of RPN. It is important to note though that this is not the stack that the RPN evaluation actually operates on, and the reason for using the stack is that, since RPN evaluates expressions right to left, SubExpressions must have the rightmost subexpressions be the first out. This happens totally naturally when using the compiler, which processes right to left, to push to the structure which uses a first-in last-out system. Evaluation of expressions occurs in the Evaluate method overload without parameters, which itself calls EvaluateSubExpressions to process the subexpressions in an expression.

Within the tuples that compose each SubExpression element is an integer value, primarily referred to as argumentCount. A non-zero argumentCount value associated with a subexpression effectively has the implication that it is the base subexpression which is at the start of a clause, though that implication itself holds no particular significance in evaluation. What this then provides is information to the runtime environment of how many arguments it should be passed, so in the context of RPN, how many of the expressions pushed to the RPN evaluation stack should be popped back and passed to that base subexpression as arguments. For anything else, so subexpressions representing arguments for which the argumentCount value is 0, they simply get pushed onto the RPN stack so that they then can be used as arguments when the base subexpression that will evaluate them comes.

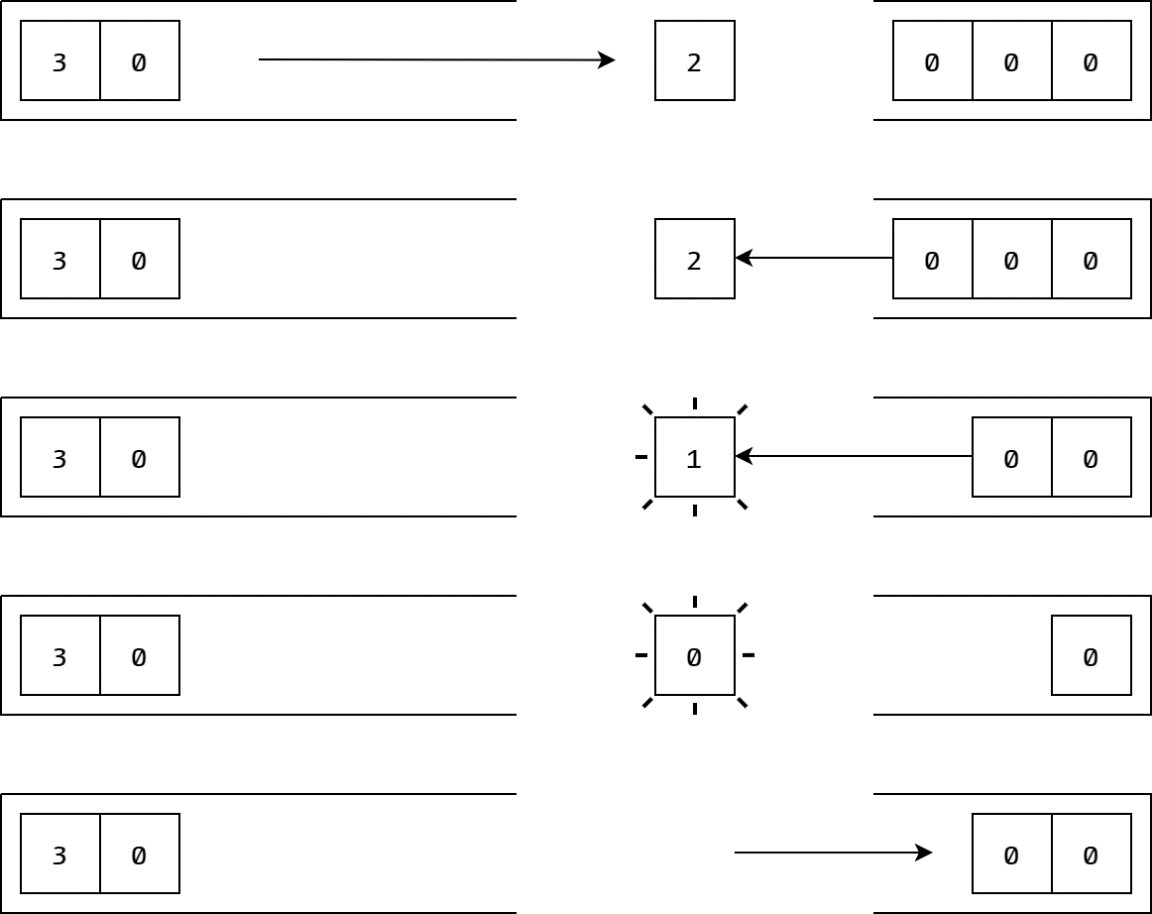
A very important thing to note is that the value of argumentCount for a base subexpression does *not* just match the total argument count for that function. Take for example passing an argument to a higher order function; to properly match the type signature of a parameter, it may be necessary to only pass some arguments to a function to produce a partially evaluated function e.g. where takes two arguments. Here is explicitly only evaluating *one argument*, not two, and this is what is specified by argumentCount when evaluating subexpressions.

  
The arrangement of SubExpressions in RPN configuration can be visualised as above, where each block represents a subexpression, and the number in each block represents its argumentCount value. The process of operation would be as follows:

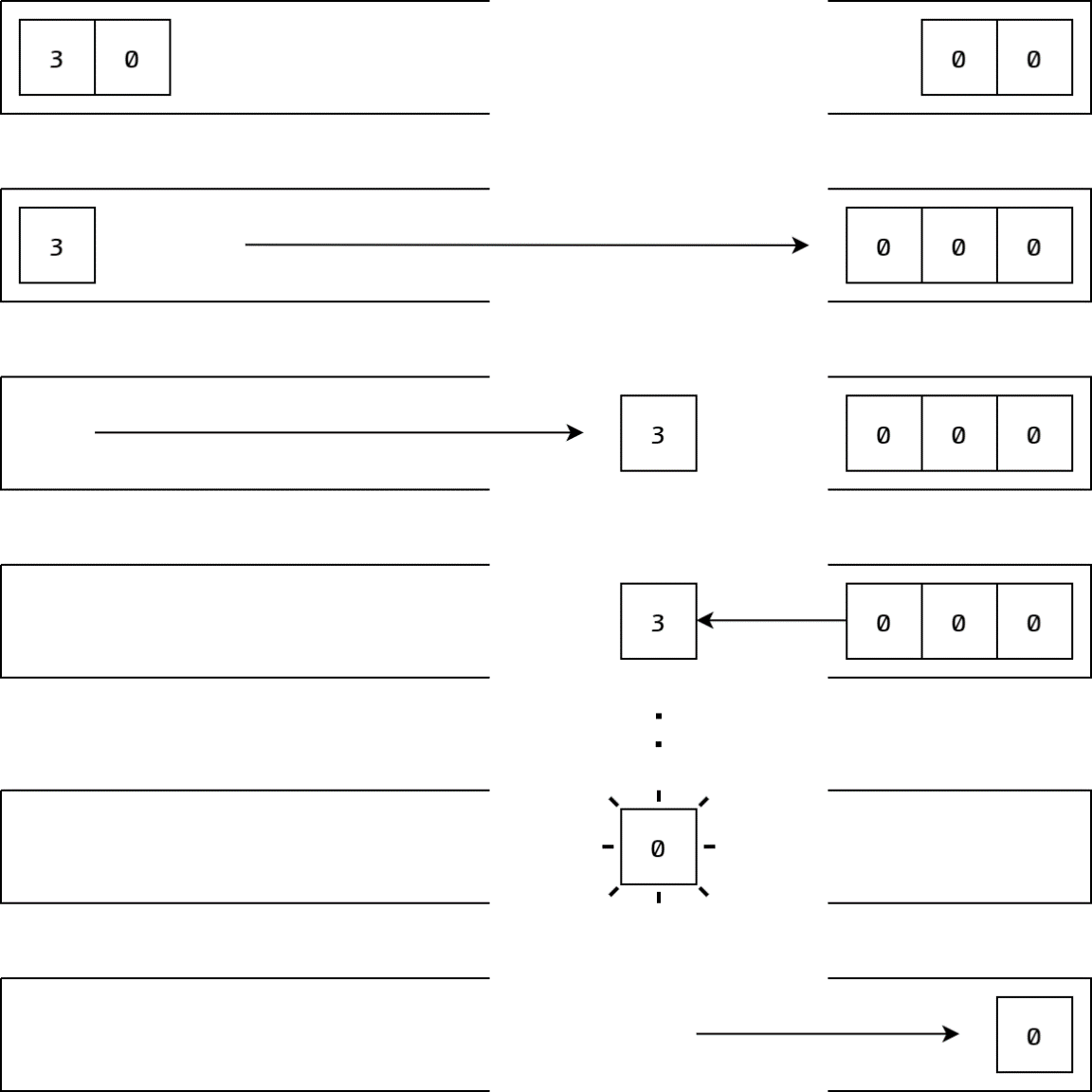
* The subexpressions from the stack are moved to the working stack and nothing is done with them as they have argumentCount values of 0, so the first two arguments are moved to the pushed to the working stack.

Arguments are pushed onto the working stack for RPN evaluation

* Then when the expression with a positive argumentCount value is popped – the positive value indicating it will take arguments – arguments are popped back out of the working stack and passed to that expression using its Evaluate overload taking a PExpression argument. One by one, they are passed to the expression, its argumentCount value is decremented, until it reaches zero and no longer requires arguments. The new evaluated (potentially only partially) subexpression gets placed onto the working stack and will be treated like any of the other expressions that are worked on there. In this case, the expression takes two of the arguments from the working stack, then its result is pushed onto the working stack.

  
Arguments popped back off the stack to be used in expression evaluation, the result pushed

* The whole process continues until final the SubExpressions stack is empty and the final base subexpression has been evaluated. At this point, since an expression should only be having its subexpressions evaluated if it is totally parameterised and therefore an unevaluated expression, the final, singular object left remaining in the working stack *must be* the final evaluated expression. For this example, the last subexpression besides the base subexpression is popped and pushed onto the working stack, the final base subexpression is evaluated using the three arguments in the working stack, and then its result is finally pushed onto the working stack as the final result.

  
The process repeats until the SubExpressions stack is empty and only the result   
remains in the working stack

This process of evaluation of subexpressions is the driver of all evaluation of user-defined expressions in the runtime environment, applying the RPN evaluation process to functional programming at execution. There are, however, additional processes that occur within evaluation that manipulate the flow of subexpressions from the SubExpressions stack to the working stack where they are processed. These involve the final part of the nested tuple used for the stack.

#### Implementing selection – specifying conditions

With a proper environment to allow expressions to be evaluated and so effectively have function programs be run, it felt as though the runtime environment was set to be able to tackle any problem within its limitations. However, a subtle flaw as part of the logic in expression evaluation meant that there still remained an issue with a particular type of function, which in fact is commonly used in function programming; this type of function is that where the function uses recursion.

Given the standard evaluation procedure, one way to implement selection using some kind of condition statement would be to define an IfThenElse function as a base expression in the compiler. How this would operate is it would take a condition, an expression for if true, and an expression for if false as arguments. It then returns either of the expressions depending on the value of the condition, so allowing selection in the language. One downside of such implementation was that, in order to allow for expression arguments that weren’t necessarily variables, the generic type specialisation of TypeSignature had to be made to encapsulate functions as well as variables, which introduced even less certainty in the types of arguments passed to base expressions.

The main problem with this implementation, however, was how it shaped function evaluation. Despite the purpose of the IfThenElse function, the evaluation process wouldn’t discriminate it or its arguments from anything else; it would treat them just like a regular function. What this meant was that as part of the RPN evaluation of subexpressions in an expression, both of the expressions that would be passed to the IfThenElse function would be fully evaluated.

Now consider a function which operates on recursion. This type of a function references itself to produce some value, and the end of the recursion would be provided by some base case as chosen appropriately using selection. Using the IfThenElse base expression implementation, it in fact would not matter that a base case exists; the evaluation process would always evaluate both of the expressions in the statement. What this means is that for a recursively defined function, when attempting to evaluate it, the runtime environment would actually recurse through its self-reference infinitely, regardless of whether for any particular call of the function, its base case should be reached.

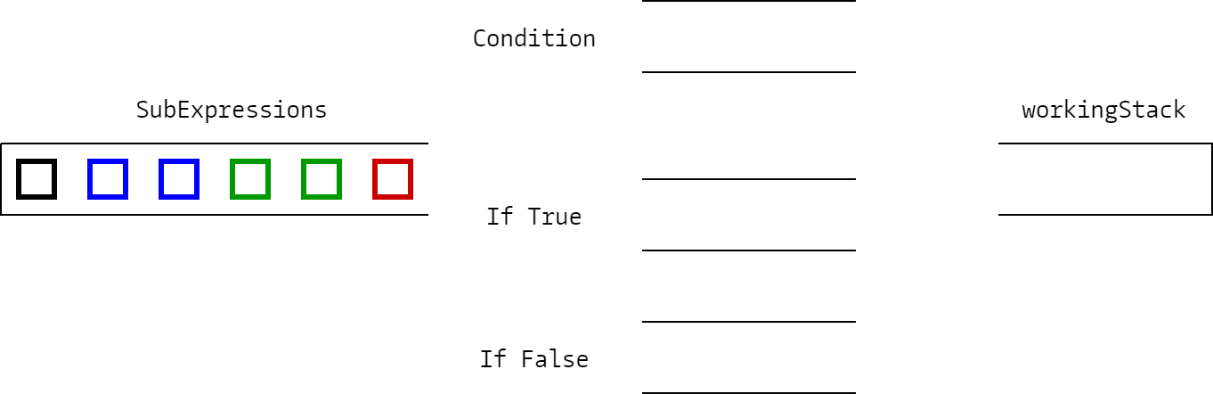
This presents the need implement a selection process in the language which actively does not evaluate certain subexpressions until required to do so; this would be the only way to implement selection and not introduce infinite recursion for self-referencing function definitions, and so requires implementation within the expression evaluation process itself.

How this is addressed is by including another evaluation specifier of sorts to the nested tuples which make up the subexpressions. Using a stack (the choice of which over a queue simplifies copying at compilation) of “condition specifiers” to indicate what purpose a particular subexpression serves, the evaluator can selectively evaluate expressions according to a condition, where the stack enables nesting of selection clauses. These specifiers are encapsulated in the ConditionSpecifier enum object, as the enum values ConditionSpecifier.Condition, ConditionSpecifier.IfTrue, and ConditionSpecifier.IfFalse.

#### Implementing selection – selective evaluation

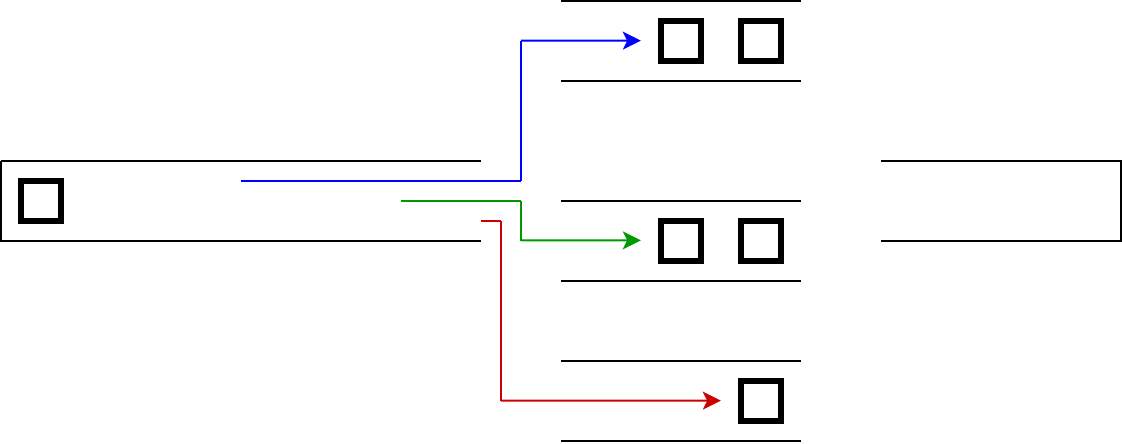
Under the premise of only evaluating what’s necessary, there has to be a way for the evaluator to temporarily store subexpressions to be evaluated on demand. The use of the ConditionSpecifier enum simply enables subexpression clauses to be identified as different parts of a selection statement: ConditionSpecifier.Condition specifies a clause which evaluates a condition to check, ConditionSpecifier.IfTrue specifies a clause to evaluate when the condition returns true, and ConditionSpecifier.IfFalse specifies a clause to evaluate when the condition returns false.

Given the most natural order of writing selection blocks, a group of subexpressions representing such would start with a condition, directly followed by the clause if true, directly followed by the clause if false, and then the selection block is complete. In EvaluateSubExpressions, the start and end of such selection blocks are easily identified; after having encountered a clause specified as part of a condition (encountered last because of the RPN order of evaluation), the next subexpression it encounters with a different specifier, or potentially just the bottom of the SubExpressions stack, indicates the end of the block. The start is indicated by the first subexpression specified as evaluated if false, though registering the start isn’t necessary; there will be no interruptions until the end of the condition clause.

The way the evaluator temporarily holds subexpressions within the method is using queues. They act as a hold for those clauses which comprise the different parts of the selection block, and important aren’t stacks so that the order of the subexpressions is maintained through them; they still get evaluated the same way.

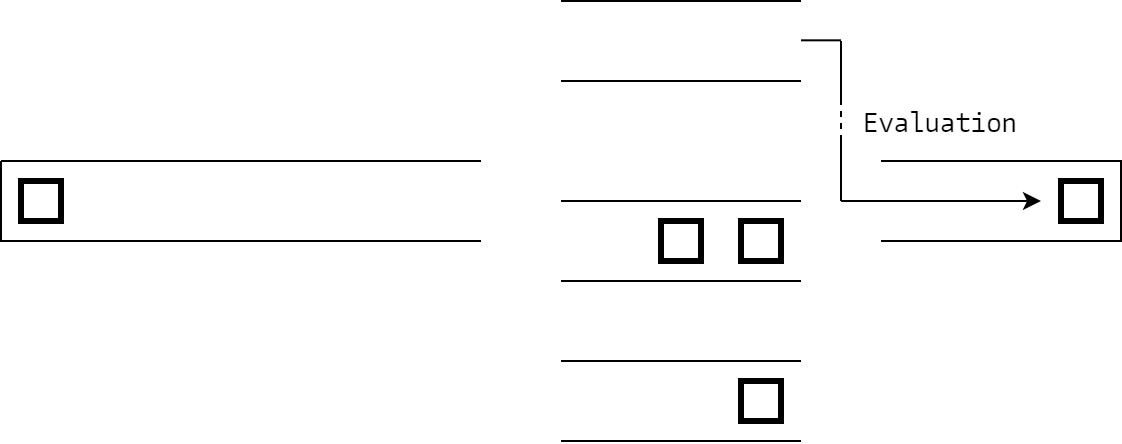
Consider the arrangement as visualised above. Blue subexpressions represent those in a condition clause, green represent those in an if-true clause, red represent those in an if-false; such specification is provided by the first ConditionSpecifier enum value popped from the stack primarily referred to as conditionSpecifiers in the nested tuple subexpression element.

For any subexpressions without a given specifier simply bypass the entire process and are just evaluated in RPN fashion using the working stack on the right. Any subexpressions that do have a specifier are instead placed into the appropriate queues rather than directly evaluated.

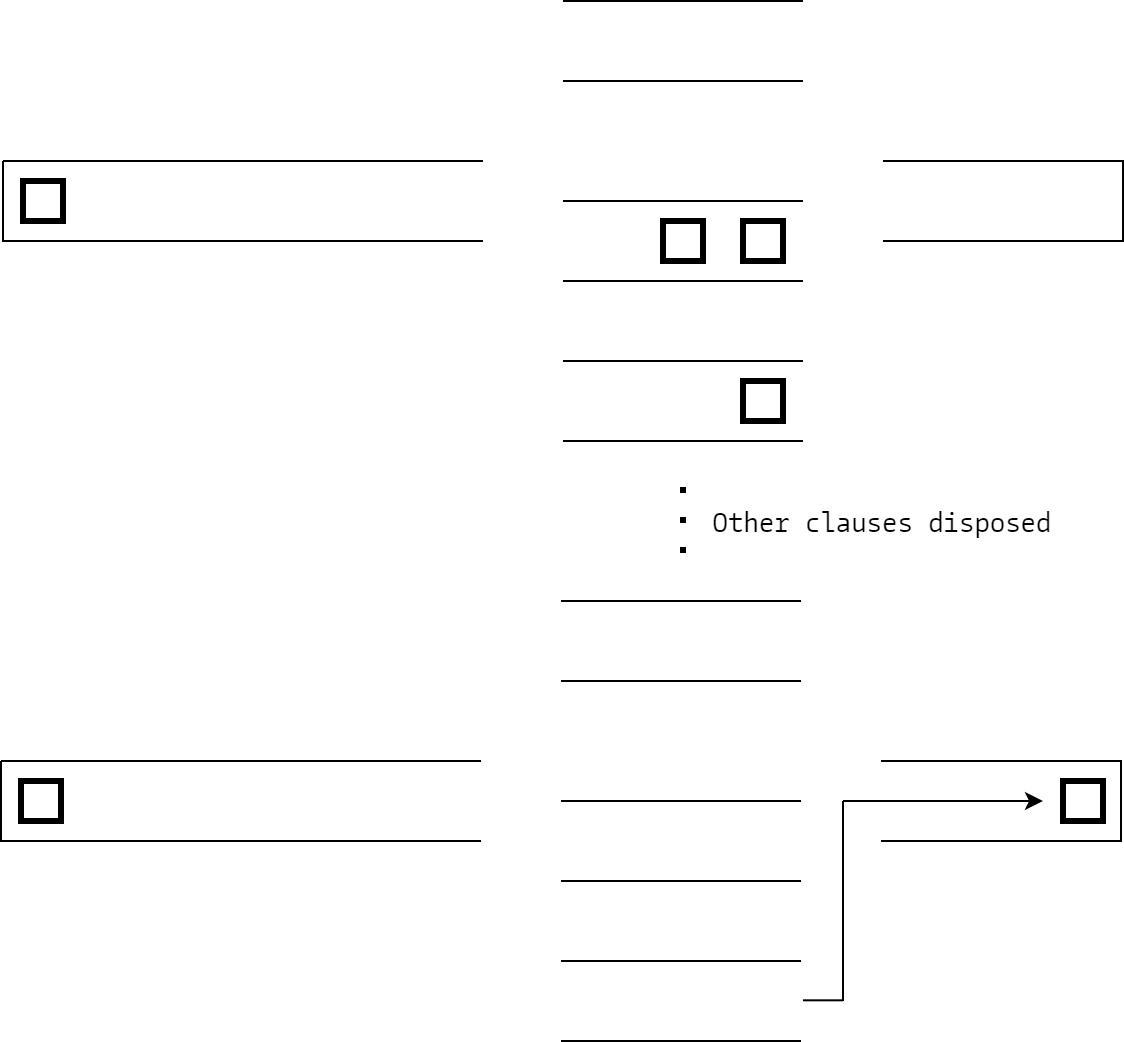
  
  
Subexpressions are queued into the appropriate queue and their top specifier discarded

Once the entire selection block has been retrieved – which in this case is indicated by the next subexpression not being a condition, though it could be indicated by there being no more subexpressions – then the condition clause of subexpressions is to be evaluated. It is important to note how the condition specifier previously indicated by the colour is discarded (in the context of the ConditionSpecifier stack, the top value is popped to identify what the specifier is and isn’t pushed back on.)

For the evaluation of any of the clauses, EvaluateSubExpressions is recursively called and passed the queue of the clause to be evaluated and the working stack. Something to note is that the primary working overload of EvaluateSubExpressions which encompasses the RPN evaluation process actually takes a queue, not a stack, and a working stack to evaluate with so that every recursive call of EvaluateSubExpressions pushes subexpressions to the same place. The overload of EvaluateSubExpressions called by Evaluate wraps this primary overload by initialising a queue from the SubExpressions stack (which maintains orientation) which enables it to evaluate effectively as though popping from the stack directly, but actually providing the option of a queue to enable evaluation of the temporarily stored condition clauses. A new working stack is also initialised in the wrapper.

Using the recursive call of EvaluateSubExpressions, first the condition clause is evaluated.

Condition clause evaluated using working stack

The value of the condition clause is popped from the working stack, and if true, then the if-true clause is evaluated, and if false, then the if-false clause is evaluated. In this case the evaluation result remains in the working stack, since it is the result of the selection that is supposed to be being used, while the unused clause is discarded so the queue can be used again for the next selection block. It is also worth noting that evaluating these condition clauses doesn’t necessarily have to include proper function evaluation i.e. if it’s just an argument that the selection block determines, then evaluating the appropriate clause will simply push the argument and that’s it. Importantly however, any clause will always evaluate to a single subexpression, as is such for clauses in subexpressions regardless of a condition.

Appropriate clause evaluated, queues empty for further use

With everything totally evaluated, and only what is necessary being evaluated, the subexpression of an unevaluated variable can be totally processed and a value retrieved. The EvaluateSubExpressions overload used by the parameterless Evaluate overload pops the final expression in the working stack and returns it, which Evaluate then also returns after assigning the state variable to PExpressionState.Evaluated. The returned PExpression object would then have a value assigned to its Value property, and can then be made useful in any other context in the runtime environment, including the output of the program. This evaluation process is what underpins the entire system.

# Paskell Translator

In order to make the runtime environment useful, there must be a compiler of sorts to enable user-written source code from the code editor to be converted into PExpression objects which can be evaluated to produce some result. This is achieved in two steps: Tokenisation and Compilation, handled in Translator.cs by the Tokeniser method and the Compile method respectively.

## Tokenising source code

A language can have its primary written components split up into different types. For Paskell (the language used here, which is my own) this includes equality, function mapping, condition statements, words (representing functions, arguments, etc.), operands, and brackets. All of the chunks of code these can represent are called tokens, and their types are encapsulated in the TokenType enum, with respective values for each. Included is a token for “line breaks”, which – since Paskell is compiled line by line – allows a line in Paskell code to span across multiple lines in the editor. The tokenising process is done using the Tokenise method.

The way that these tokens are identified in code is by Regular Expressions associated with the token types. They can be associated to them using a string pattern attached to the enum type using a custom Attribute called RegexPattern. They define the tokens as such:

* Brackets are either a ‘(’ or a ‘)’
* Equality is a ‘=’
* Function mapping is a ‘->’
* Words are alphanumeric strings that cannot start with a number and cannot be operands or condition statements
* Operands include bool operands i.e. true or false, integers, floats, and characters, represented using the single quotation marks surrounding a single character
* Condition statements include if, then, else, and endif
* Line breaks are newlines, not including those that follow a backslash

### Retrieving enum attribute information

The Regex patterns for each of the token definitions, as given by the RegexPattern attribute, is retrieved using a custom enum extension method called GetPattern. It operates by taking the MemberInfo object associated with the enum type and retrieving its attributes. They are returned as type-less objects, so must be cast; knowing the attribute will be of type RegexPattern, they are cast to RegexPattern objects from which the Pattern property, as is assigned when constructing the attribute for the enum type, can be accessed and so the pattern retrieved. This function is defined to be called like a method from the TokenType enum object. The solution to this problem is provided here: [bit.ly/enumtostring](http://bit.ly/enumtostring) (*Enum ToString with user friendly strings*)

### Matching code and producing tokens

In order for the tokeniser to be of use, it must end up providing an array of tokens which can be used by the compiler to produce expressions. These manifest as Token objects, where Token is a structure primarily containing properties that cannot be modified once instantiated. These include: the string property Code, meant to store the code associated with the token such as the string of an operand, which is effectively what the Regex match would return; the TokenType property TokenType, which just indicates what type of token it is; and the integer property Index, which indicates in what position in the source code the token is representing code for. The Token constructor initialises all of these properties using arguments passed. Importantly, Token inherits the IComparable interface which enables tokens to be compared to each other, in this instance by their indices. This is primarily to enable the tokens to be sorted once found, which is addressed at …

Before attempting to tokenise the code, first a bool array codeMatched of length matching that of the source code is instantiated. What this is used for is to identify what parts of the source code couldn’t be tokenised and so where there must have been invalid code. To be able to fill chunks of the array up in one command, the custom extension method Populate from Tools.cs takes a single value of the type of the array elements as value, the start index of where to fill the array as startIndex, and the length of part of the array to fill as length as arguments and uses them to fill an array with a single value. Simply using a for loop counting up to length, the element in the array indexed at startIndex plus the for loop indexer is set to have the value value. To clarify, this function can be applied on an array of any type by using generics i.e. the type of the array replaces T, and as such specifies what the type of the value to populate the array must be.

To actually find tokens from the code using Regular Expressions, every TokenType enum type as provided by the built-in enum extension method GetValues is looped through, and each of their associated Regex pattern is retrieved using GetPattern which is then used to find all matches in the source code using the static method Matches from the built-in class Regex. For every match, a new Token structure object is instantiated and passed the code that it matched with, the token type it matched for, and the index of the match such that the token can then be sorted. The new Token object gets pushed into a list of Token objects, and the codeMatched array is populated using the Populate method for the entirety of the Regex match, which indicates that this part of the code has been parsed.

Once every token type has been searched for and therefore the parts of code where tokens matched indicated by populating the codeMatched array, it is then necessary to populate codeMatched for where tokens would not have been found but where there exists only whitespace or a backslash followed by a newline, which are valid. The source code is searched for those using Regex once more and Populate is used to validate the code as indicated by codeMatched.

### Sorting tokens – merge sort

Since the tokens are instantiated in the order of which token types were searched for first, the tokens will not be in the right order. It is therefore necessary to sort them, as can be done by the custom extension method Sort from Tools.cs which passes the associated array to MergeSort, which implements a non-recursive merge sorting method that instead handles its own memory allocation with the use of a stack. A merge sort seemed appropriate for such an application, as long files with many tokens could produce very long unsorted arrays which would be most effectively sorted with this particular algorithm.

By implementing a stack which holds integer tuple arrays to indicate divisions, a working merge sort algorithm can be written. The integer tuple array represents all the levels of divisions made, with the first item in a tuple indicating the start of a divided part and the second item in a tuple indicating the length of it. To start, the stack holds a single tuple which describes the entire array as one whole undivided part.

To make all the divisions in the array, the process of dividing is repeated until, for all of the divided parts at a given level, no more divisions occur. For a new level of divisions, a list of integer tuples is initialised, used rather than an array such that a dynamic number of division parts can be specified. Each currently divided part is considered by looping through the array given by peeking the top of the stack, and so long as that part is divisible (to its length is greater than 1), the part is divided: a division part with the same start index and half the length (rounded down) is added to the divisions list, and another division part with index starting at that half-way point (rounded down) and half the length (rounded up) is added to the divisions list. The array is considered still dividing if for a given level of divisions, at least one new division is made. Once through every current division, the new divisions list is made into an array and pushed to the stack, and the process repeats.

The importance of using a stack to represent all the divisions is that there may exist cases where, at the bottom of the splitting process, only parts of the array were split. Therefore these must always be addressed first, attempting to merge nothing else. For every stack element only the necessary divisions are indicated, and so only the necessary merges will be made.

To merge, the set of division parts to merge for is popped from the stack. Every division comes in division part pairs, so the set is looped through by two, and two division parts are processed at a time. A temporary array is instantiated so that all the elements in a divided part of the original array can be copied to it first before it replaces the array. To keep track of the progress made in both division parts when merging, two indexers are instantiated, and the sum of both is used to index the merging array. Accessing an element from the original array of either division part is done by taking the sum of its start index and its respective indexer.

Until either one of the division parts in merging has had its indexer reach its end, the currently indexed element of both is compared using the CompareTo method (needs to be used rather that < or > for types of IComparable) and whichever is lowest is added to the temporary array using the summated indexer (the comparison is flipped if the function parameter flag descendingOrder set to is true; default is false). The process is repeated until an indexer reaches the end of its division part, then the remaining uncopied elements from the other division part (which of the parts that is can be checked by comparing the two indexers) are copied to the temporary array. The temporary array is copied to the original array, replacing the elements that have just been sorted.

This merging process repeats for every pair of division parts at a division level, then for every division level in the stack. Once the stack only has a single division level left, indicating the whole complete array, the sort is complete and the sorted array is returned.

### Producing an output

Once the tokens are sorted, the token list is ready to be provided as output. However, there exist cases where the tokeniser fails, which is where certain parts of text could not be tokenised and did not count as valid whitespace. This particular type of information is encapsulated in the TokeniserReturnState class, which has a bool flag Success indicating if the Tokeniser completed successfully and a queue of TokeniserReturnError instances for if not successful. The TokeniserReturnError class is simply a wrapper for the index of an error, given by its property Index, which indicates where any invalid tokens were written.

The tokeniser produces these by first checking if any of the elements in the codeMatched array are equal to false. If not, then a TokeniserReturnState object is initialised and passed true to set its flag Success to true, indicating success in the tokeniser, and this object is returned. If there are any elements in codeMatched that are false, however, then a TokeniserReturnState object is initialised and passed false for failure, and the entire codeMatched array is looped through to check for an element set to false. Once found, a TokeniserReturnError object is initialised with the index of the element and enqueued to the errors queue in the TokeniserReturnState. Consecutive false elements are ignored, since they would all compose one token, and the indexer is incremented until an element set to true is found, and the next invalid token would be searched for. Once codeMatched has been totally considered, the TokeniserReturnState object is returned with its errors.

The more useful output of the token code, primarily for when the tokeniser succeeds, is actually provided through the parameter using an “out” parameter modifier. An argument of the type of a Token array is passed to be worked with; once the list of Token objects has been made, it is turned into an array using the ToArray extension method (it is in fact at this point that the array gets sorted), and the output is assigned to the TokenCode “out” parameter to be used again by the caller that passed it the empty array in the first place.

The tokeniser errors are primarily used by the IDE in its HighlightErrors method. The public method GetTokeniserErrors wraps the tokeniser error return by calling the Tokenise with the source code passed, any token code is disposed of, and the errors queue is returned.

## Compiling token code

When writing Paskell code, every line of code represents an expression definition – a “line” can be divided across multiple lines in the code editor by writing a backslash before a line break. The structure follows a type signature, an expression identifier, parameter identifiers (only as many as the type signature indicates, which may be none at all), and an expression definition as given by a sequence of subexpressions, potentially within nested brackets or condition blocks. The Paskell compiler is responsible for taking such a sequence of code as would be represented by token code, and converting it into PExpression instances which can be evaluated by at runtime. Any invalid code would be caught compile time and execution would be prevented, and useful errors would be provided to the user by returning PaskellRuntimeException instances. This compiling process is done using the Compile method.

### Defining base expressions

The compiler uses a general context where all of its expressions definitions are kept, using this to be able to define expressions in terms of others in the context; this is maintained by the PExpression list called Expressions.

The first expressions to be defined and added to this context are the base expressions that comprise the lowest-level function implementations in the runtime environment. Actual function definitions as part of the static Translator class are passed to PExpression constructors, with appropriate type signatures and matching identifier strings. These functions correspond to actual arithmetic, boolean logic, and comparison and inequality functions, and are added to Expressions as part of the context.

### Defining user expressions

Next, all of the user expressions as given by each line of code need to be defined and added to the context. Firstly, each line of code needs be isolated, so the entire token code array needs split into lines by searching the whole token code array for line breaks and copying the array between line breaks (or potentially the start or end of the array.) These copied arrays then get added to a list as part of a nested tuple. The tuple contains integers which indicate the dividing locations to make the particular sections of the expression; these include an index indicating the end of the type signature and the start of the expression signature i.e. the expression name and parameters, and an index indicating the end of the expression signature and the start of the expression definition. These division locations are found when first instantiating the expressions, and are kept in the tuple to be used in other parts of compilation so that they don’t need to be found again.

When finding these division points, the structure of tokens within an expression definition must be considered. For a type definition, there are no consecutive words, as every type reference is punctuated by a function map (ignoring brackets). Therefore, the location of the start of the expression signature can be found by finding the first consecutive words in the line; this is done by setting a flag indicating if the last word was true, and if it remains true and another word is encountered then the division is set at that point. The end of an expression signature and the beginning of a definition is simply indicated by the location of an equals sign.

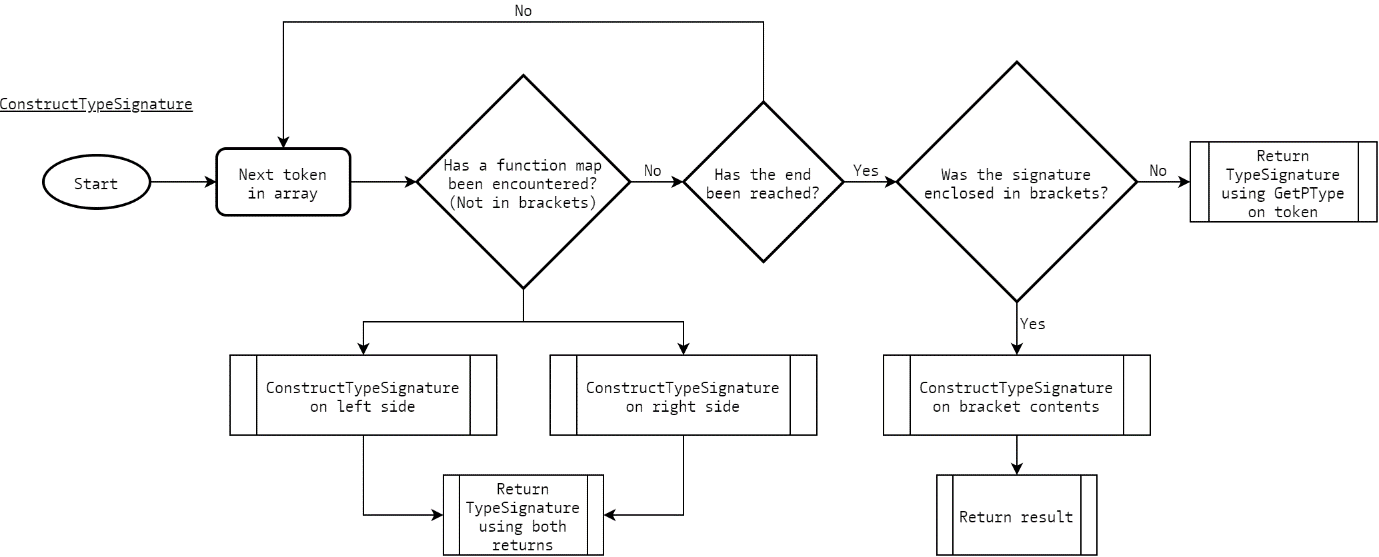
Once divided, every expression the user defines must be instantiated – this occurs directly after division of each line. All of the expressions must be instantiated first before being defined, because multiple expression definitions will inevitably contain references to other expressions; therefore they must be instantiated and known by the compiler to exist first before their subexpression stacks are populated. Retrieving an identifier string for the expression is done simply by taking the first word after the type signature. The type signature, however, must be parsed in a particular way.

#### Constructing a type signature

In order to construct a type signature, the compiler passes the type signature segment of the line, indicated by the expression signature start division, to the ConstructTypeSignature method. Much like the self-referencing nature of the structure of the TypeSignature class, the method uses recursion to produce a TypeSignature object.

Looping through the tokens, anything in brackets are ignored, and only tokens at the lowest bracket level i.e. are not nested within any brackets, are considered. At the first function map token encountered, the loop breaks and ConstructTypeSignature is called, passing it the subarray of the all the tokens on the left of the function map. The remaining tokens on the right, so the ones that haven’t been looped through, are then passed to another ConstructTypeSignature call, and the result of these two calls compose the parameter and return of a new TypeSignature object that is instantiated, and finally returned.

If no function map token is encountered, then either the only token in the subarray passed to this particular method call is a Paskell type, or the entire segment is nested within brackets. If this is the case, the ConstructTypeSignature is recursively called, passing it the original array excluding the brackets on either side, and the result of this is returned. In the case of the subarray just containing a Paskell type, a new TypeSignature object to represent this type is instantiated and returned.

  
ConstructTypeSignature implementation

At the base case of this algorithm, which is constructing a type signature from just a type, the type in C# associated with the type given by the token must be found in order to instantiate a TypeSignature object representing that type. All the possible types in Paskell are given in the OperandType enum type, with values for int, float, char, and bool (note the @ symbol before these definitions to distinguish them from C# keywords). The associated C# types (using long and double for int and float for better values) are attached to the enum values using a custom Attribute called PaskellType. These associated types are retrieved using the custom extension method GetPType, and it operates exactly like the GetPattern extension method for retrieving Regex patterns of tokens.

In the compiler, once an expression has been instantiated and its type signature constructed and assigned, definitions for each expression then need to be created. First, the right expression must be found from the list of expressions that have been declared (which if it cannot be found or there exist multiple copies, throws an error), and once found its parameters need to be represented as their own PExpression objects specialised as parameters, with the identifier as what the parameters are called. When compiling the subexpressions of an expression, both these parameters and the global expression already declared are used to compose the definition, but the parameters apply only to the context of the expression they are used in, so are discarded for the next expression and its own parameters are used, if any. This is effectively creating a local context for an expression, sensible considering for example defining multiple functions which all use parameters of the same name, such as x. With the context for an expression ready, it then must have its subexpressions parsed.

#### Compiling subexpressions

In order to properly understand the process of the compiler at this point, it becomes necessary to reconsider function implementation. In most cases, no subexpression stands alone. The subexpressions group that composes the function definition is not unique; it is a clause, and within in can be many other clauses. When trying to apply the functional programming mind-set of clauses in expression definitions, two very important considerations arise:

* Every clause starts with a base subexpression
* Every subexpression or clause of subexpressions must fulfil a certain type signature

Bear in mind that a clause can take the place of *any* subexpression within a definition, and this include the base subexpression of another clause itself. This result in particular is special and requires further consideration of base subexpressions themselves when trying to understand what the type of any clause should be.

The first thing to note about subexpression clauses is that the base subexpression of that clause doesn’t necessarily have to match any function or function clause type in particular i.e. there is no exact, perfectly defined type that the function should match. This is because any given function that is to be passed arguments, effectively by definition does not have any one particular type signature; it can end up being *any* of the return type signatures it eventually would give back when given arguments.

Subexpressions and subexpression clauses that comprise the arguments of a function, however, *do* have to match a particular type expression, regardless of the type i.e. even if the argument is another function, such as for higher order functions.

Considering these things when handling individual subexpressions rather than clauses isn’t too difficult to handle as it would all within the context of one method call. For when clauses comprise these definition components, however, so for when clauses are contained in brackets or condition blocks, then recursively calling a method to compile these clauses requires passing on the information required so that the next method call knows what type of a subexpression its clause is composing – effectively, whether or not it is a base subexpression, and also if not, what type signature it must match.

#### Compiling subexpressions – recursive clause compilation

Considering this, the PushSubExpressions method takes, naturally, the token code representing a clause and the expressions that make up the context for an expression definition; in addition to this, however, the method must take a target signature specifier to indicate what type the clause is meant to match as a TypeSignature parameter called targetTypeSignature, and whether or not the clause is meant to be a base subexpression or not, so whether or not it needs to match the type signature exactly, as a bool flag typeSignatureMustMatch.

In addition to these, the method needs to take: a stack of condition specifiers, such that when calling to compile the clauses within a condition block, the condition specifiers can be nested, so the condition clause currently being compiled remembers in what clause it was in; and a PExpression parameter which is used as the output of the expression, so whose subexpression stack the subexpressions are pushed to. The reason for this is because in the case of the clause being a base subexpression, the type signature doesn’t have to match the target passed (as indicated by the bool flag passed). What this means, however, is that what is compiled is a new base subexpression that may still take arguments in the outer clause is it part of, and it can’t be known what the type signatures of those arguments must be without knowing the type signature of that base subexpression clause given. Therefore, the PushSubExpressions method needs to specify this by returning the type signature of the clause it just compiled. The compile output therefore needs to be handled without using returns, and so simply by passing along the PExpression object to which the subexpressions are being give solves this.

To actually compile the subexpressions, all of these considerations of clauses having to be base subexpressions or arguments, or potentially even just literal values to pass to variables for example, have to be take into account. The first PushSubExpressions call is specified that the type signature *does* have to match the given type signature; this is because any expression the user defines must be able to give a real value. This applies even to functions, because once fully parameterised they must be able to be fully evaluated to the final type they provide (in fact, the type signature passed to the first PushSubExpressions call is always the *final type* of a type signature i.e. for a function, what it will eventually be once totally parameterised). The subexpressions clause passed is the whole expression definition, the expressions passed composing the context for that expression is the concatenation of the global expressions and the parameters of the function, if the expression is a function, and the output expression passed is the one that has already been instantiated. The stack of condition specifiers is initialised empty.

Clauses in subexpression sequences can arise as either bracket clauses or condition clauses. For bracket clauses, when an open bracket is encountered, the amount of bracket nesting is counted so that for any closing bracket it is known whether it is closing the whole bracket clause or just another clause within it. Once the final close bracket is found, the clause is considered as the contents of those brackets, not including the brackets themselves.

For condition clauses, the condition nesting is first considered when an “if” token is encountered i.e. a condition statement token with “if” as its code. Note that anywhere within the following selection block, other selection blocks are also ignore much like nested brackets by just considering an “if” as their start and an “endif” as their end. From the “if” to the “then” is the condition subexpression clause, which is treated unlike any other clause as it must always produce a bool result. Then between the “then” and “else” is the if-true clause and between the “else” and “endif” is the if-false clause. Both of these clauses are then considered, much like the bracket clauses, by recursive calling of PushSubExpressions. The type signatures of both of these clauses must match, and the condition clause must always return a bool, and an error is thrown if either are not the case. Any missing condition statement tokens also throw an error.

Given that evaluating clauses can be done appropriately by recursively calling PushSubExpressions with the right flags and arguments, these clauses can then be considered like any other subexpression within a clause. Either:

* If they represent a base subexpression, the call is passed the type signature of what they should eventually be able to match, and the typeSignatureMustMatch flag is set to false; what the call returns is the resulting type signature of the base subexpression just defined, and this is used to specify the type signature of the base subexpression the clause represents.
* If they represent an argument, the call is passed the type signature of what they must match to match the parameter, and the typeSignatureMustMatch flag is set to true; the call will simply return an error if the resulting type signature of that clause doesn’t match appropriately.

This is how clauses are considered, and from this they can be processed just like other subexpressions like expression references and literal values; next must be considered how this occurs.

#### Compiling subexpressions – the subexpression sequence

Having already considered the possibility of a subexpression being a clause – in which case they are considered by a recursive call – the remaining possibilities of what a subexpression may be represented by must be considered.

The simplest possibility to consider is if the subexpression is just a literal value. Here, the literal value must match the type, and using the Type object specified from a type signature, a TypeConverter object can be used to parse the source code from the token; if it fails, an error is thrown indicating the value was the wrong type. For a generic type i.e. the Type object is null, every Paskell type as part of the OperandType enum is cycled through and the associated type from GetPType is used to attempt to parse the source code in a similar fashion.

The only other possibility is if the subexpression is another expression, so represented by a word token matching the name of another expression in the context. This could be any of the base expressions as part of the runtime environment, any of the parameters, or any of the other user-defined expressions in the file.

With all these possibilities of subexpressions considered, they can then be considered mostly equal (with some caveats) and the overall process of evaluating the subexpression can be considered. Firstly, and most importantly, there must be a base subexpression in the definition. Interestingly, a base subexpression doesn’t have to be a function; its properties still remain consistent if it in fact represents a variable and therefore doesn’t take any arguments. Once a base subexpression has been encountered, which would be the first of any form of a subexpression, then its type signature is recorded in a TypeSignature variable called baseTypeSignature; this variable begins as null to indicate that the base subexpression hasn’t been compiled yet.

In order to properly apply the type signature comparisons to the subexpressions, it becomes necessary to identify key parts of a type signature. The type signature for arguments is found by indexing a TypeSignature object – part of the TypeSignature class definition is an indexer method that allows it to be indexed much like an array; for a given index value, the result is the index of the Return TypeSignature for the index value minus one, and the zeroth index of a TypeSignature object is simply itself. From this definition, a TypeSignature object can only be indexed as far as however many arguments the expression it represents takes, and the total argument count is given by an ArgumentCount integer property which operates on the same recursive process, stopping at where the TypeSignature being called for is not a function. Using the indexer, the type signature for the nth argument to a function can be found by indexing the TypeSignature object for n and taking the Parameter property (important because what the indexer does is return the function type signature for that index, not what that function takes as an argument).

After the base subexpression, its arguments passed must then be compiled and checked. The type signature of each argument is found as described above, and when considering a subexpression which is an argument to the base subexpression, this resulting argument type signature is recorded to argumentTypeSignature. Which argument is currently being checked, and therefore what the valid type signature is, is maintained by the argumentCount variable, which begins at 0 and increments for every argument. Once it reaches the ArgumentCount property of the base type signature, an error is thrown indicating that too many arguments have been passed.

Unlike the base subexpression, the type signature of an argument must be correct. For clauses this is naturally handled using the typeSignatureMustMatch flag, and for expressions and literal values, this is checked directly. If the type signature doesn’t match, an error is thrown indicating that the argument was of the wrong type signature. Interestingly, for the sake of literal values being compiled no distinction is actually made by the method as to whether or not the current subexpression is the base or not, and instead before a base subexpression has been defined argumentTypeSignature is actually equal to the value of the target type signature. This works because for literal values, there cannot be flexibility in type signature, and they either have the right type or don’t. So if they are the base subexpression, they are the *only* subexpression and they entirely define the clause type signature. This doesn’t affect any other aspect of the process, as argumentTypeSignature isn’t considered anywhere else until a base subexpression has been defined, after which argumentTypeSignature is properly assigned the type signature for the appropriate argument of the base subexpression.

With all of this considered to ensure that every subexpression is correct, every subexpression still needs to be correctly pushed to the stack. For arguments, they are simply pushed onto the subexpressions stack of the output expression with a 0 argument count. For base subexpressions however, how many arguments they end up taking must be correctly assigned and given as the argument count: this is done by taking the difference between the argument counts of the clause base type signature and the target type signature of the clause. In effect, provided the base type signature can eventually become the target type signature with the right amount of arguments, checked later as shown below, then that amount of arguments is simply the difference between the totals, so that once that many are passed the base subexpression will then have the same argument count as the target. If for any reason the result is negative, then it is handled elsewhere because the type signatures will not match where they need to. An important note here is that when consider a clause as a base subexpression, the recursive call actually passed *the same* target type signature as already being considered; this harks back to the though experiment of the difference between and – they are the same. By providing the bracket clause for the base subexpression with the same target type signature, it will provide the same result just with more nesting.

Once every subexpression has been considered, it comes the time to evaluate the process according to the typeSignatureMustMatch flag. If true, so the clause’s resulting type signature must match the target, then the base type signature indexed to the number of arguments, so what the type signature becomes when passed that many arguments, is compared to the target. If it doesn’t match, and error is thrown indicating as such. If the flag is false, no check is made. Regardless of the check however, the function returns back the base target signature indexed to the number of arguments. This return is how the type signature of base subexpressions which are clauses is found; though bracket nesting base subexpressions, as shown above, is not necessary, enclosing condition blocks is, and so this result is important in allowing the type signature of those clauses once considered to be used for compiling the rest of the subexpressions. The way that type signatures are actually matched in in the Equals extension method in the TypeSignature class, by comparing the parameter and return of two TypeSignature objects if they are functions, and the type of two TypeSignature objects if they are not. Functions and non-functions don’t compare so return false, and non-functions return true if their types match or either of those types is null, so indicating a generic type. The == operator overload combines this with checks for if either object is null first, preventing comparison errors where null objects are attempted to be accessed.

### Producing an output

After an expression has been parsed and then its subexpressions prepared, it is ready to be used by the runtime environment. In the cases where evaluating the subexpressions fail, or even any part of compilation of a line fails in fact, then it needs to be handled accordingly. Using the PaskellCompileException object, error messages can be generated appropriately for the error and the token index, which is which token on the line induced the error, is provided as well as the line on which the expression is written. The expression which associates with this line is removed and other expressions are attempted to be compiled, as there is no reason why the compiler should be able to check multiple lines and find errors in more than one place. A useful feature here is remembering how many lines were previously deleted so that the line number associated with an error is correct, and doesn’t decrement with each line removed; this is done using the deletedLines integer variable.

Eventually, when compilation is complete, a PContext object which simply acts as a wrapper for the expressions is instantiated and given all of the expressions together as an array. This is given back to the caller using an “out” parameter again like the tokeniser, and what is returned is a CompilerReturnState object which contains all of the errors encountered, and a flag indicating success if there were none.

The public method Compile is used by the IDE to process source code directly, and it implements both the tokeniser and the compiler as one component. When the tokeniser returns a failure, the compiler is never called but instead a compile error indicating failure parsing tokens is given.

The IDE applies the entire runtime environment and translator component by using the PContext output object to attempt to evaluate a variable identified as “main”; it in fact displays its own error directly if it cannot find it or there are multiple definitions. As such, the app can encapsulate the entirety of these two large components into the functionality of one single Start button, and an output window which pops the result of the magic. Or just the errors to tell you let you know you’re not smarter than it is.