Analysis

# Premise

I’ve been interested in the idea of functional programming and have been intending to learn it. The functional programming paradigm is a very different from imperative or object-oriented programming, and I believe requires a unique framework to be developed on. In the past I’ve built various levels of imperative programming simulations, including a simple imperative virtual machine and a custom RISC CPU simulation, though neither of them had useful interfaces to interact with them. For my NEA I would like to build a more useful tool, geared towards allowing the user to visually see the processes of a program and learn about them without being overloaded with information. The application I would like to build is a functional programming IDE, kitted with a code editor featuring syntactic highlighting, and debug functionality for syntax errors and runtime issues. By implementing a more user-oriented type of solution, it is possible to bypass the intricacies of an accurate simulation of functional programming processes, and instead focus on delivering a higher-level interpretation of functional programming paradigm, much closer to what a user trying to code in a functional programming language would need to understand and visualise.

The first problem is the user interface. As an IDE, the app requires a code editor with rich text formatting to allow syntactic highlighting; this is likely to be the most challenging UI hurdle in the project. The majority of the IDE’s functionality should be made available from dropdown menus, and these should include file creating, opening, saving, closing etc. which constitute the basic functionalities of text editors, in addition to an option to run code. Finally, there needs to be an output box to print compiler errors and runtime output.

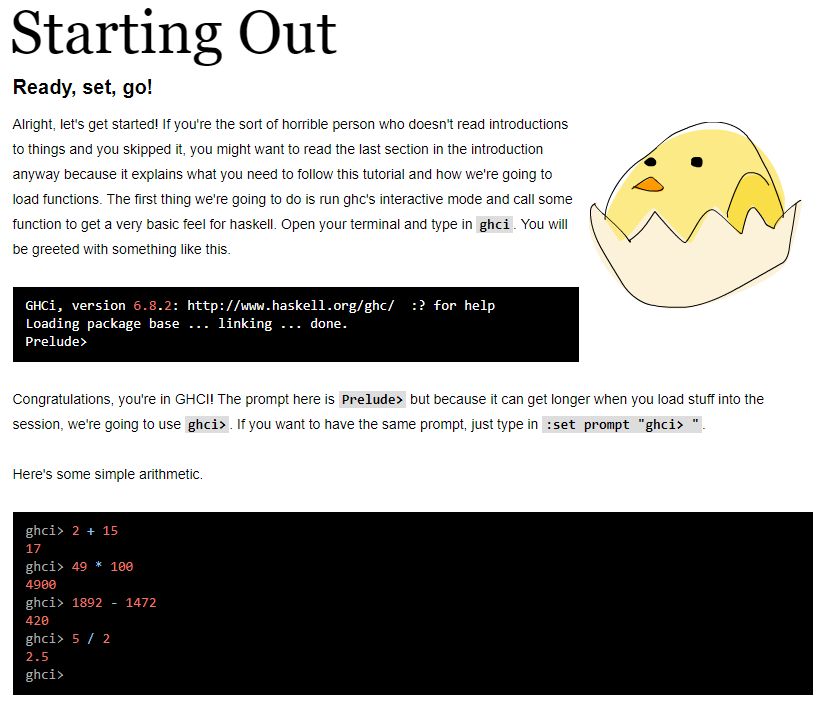
The next challenge is the runtime environment and the execution of functional code. In order to simulate the processes of a piece of functional programming code, the source code written by the user will need to be translated to a more useful format for the program to interpret. Likely the most useful format would be a symbolic representation of the code, stored in Reverse Polish Notation to simplify the run-through of code and reduce the need to hop about in it. For the translation, the compiler needs to be able to rigorously check functions and their definitions so that their types are correct, they have the right number of parameters, etc. and then be able to predictably translate source code into the symbolic representation, following a consistent language ruleset.

To complement this, it will be necessary to design a language which allows the user to build functional programs while making easy use of the complexities of the paradigm including higher order functions, but keep it as pure and simple as possible to make it easy to understand and see the process of execution the runtime environment follows. These language features will have to match user requirement, where the user is trying to learn functional programming, much like I had to as part of our class.

# Research

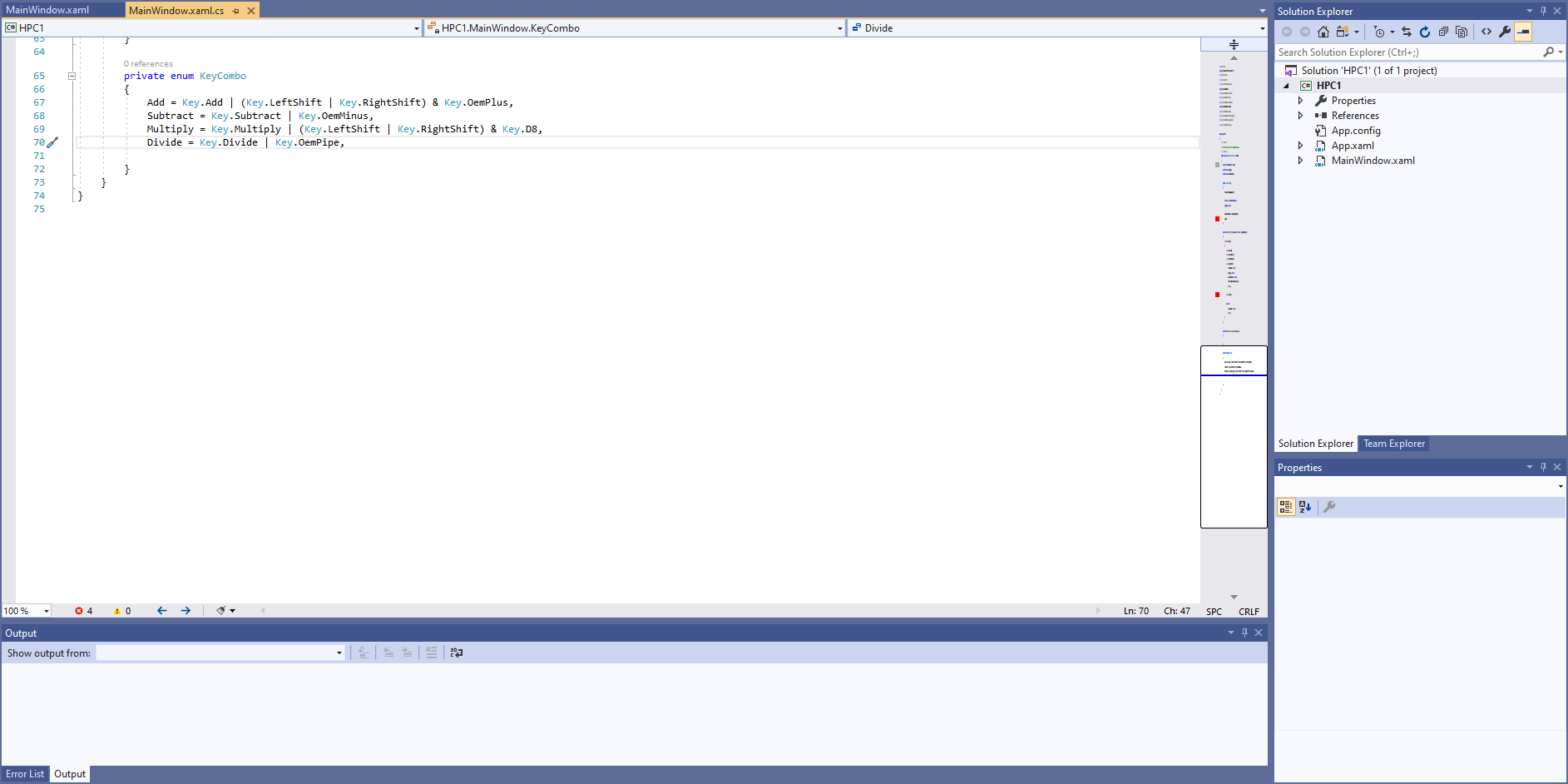
### Functional programming paradigm

Since functional programming is a novel concept to me, it was worth attempting to better understand the paradigm by learning a functional programming language, particularly Haskell. As part of our syllabus, our class was taught the fundamental concepts surrounding functional programming, particularly statelessness, function composition, type, and higher order functions.

In class we practiced writing basic functions, including using recursion to achieve algorithms such as the factorial function. A primary source of research was the website <http://learnyouahaskell.com/> from which I have been learning the basics of Haskell programming.

This has been a valuable resource for my research into functional programming, as from it I have been able to learn the basics of the paradigm and understand how I may model my own functional programming language; but it also served as a strong standard for a functional programming teaching resource that I’d like to be able to match in the context of a useful IDE. What is practical for students when trying to learn something new is simplicity and ease of use, which this website achieves by exposing them bit by bit to Haskell, and which I would like to achieve with my IDE by providing a simple language which is easy to learn and understand.

### IDE UI

A template for my app that I could use would be the widely popular Visual Studio, which is the staple IDE for developing apps and libraries for Windows. The first striking features of a new project opened on Visual Studio is the editor, which has syntactic highlighting, and notably line numbers in the left margin which make it easy to track your position in a file. Every open file is kept available by a tab above the editor which gives the name of the file and shows whether or not it is saved by showing a star \* by the name. A subtle feature, but one I particularly really like, is the ability to scroll far down enough that you effectively can have a clean page to work on, no matter how large your file already is. Finally an output window in the bottom of the app shows all syntax and compiler errors, and provides information while executing code.

More complex features include a solution explorer to provide control over projects and solutions, and a properties tab for modifying properties of controls in windows forms. However, neither of these features seem appropriate for my project, as I’m not intending to include more complex project/solution implementation, and there are no visual components to work properties for. As such, my primary UI goals from Visual Studio would be: tab control for the editor, which itself would feature line numbering and syntactic highlighting; more involved file saving visualisation with the star to indicate unsaved files; custom scrolling to enable scrolling beyond and up to a page below the final line of code; and an output window where compiler errors and program output would be displayed.

### End-user requirements

In regards to the features of my app and the requirements of the language, I spoke to my computer science teacher Ralph who taught us these fundamentals of functional programming, so that I could better understand what a user trying to learn functional programming may be looking for or what they may benefit from having as language features. He wrote:

*This is something that I would like to use in the classroom to introduce students to some of the basic, but key, features of functional programming.*

* *It would need to have some similarity to Haskell (as this is what AQA appear to use in exam questions).*
* *Function types*
* *Arguments to follow function name (so more like prefix than infix)*
* *The ability to save and open files.*
* *Higher order functions (function that takes a function as an argument or returns a function as a result)*
* *As this would be an introductory program, then the use of lists and map, reduce and fold would be an optional bonus, but not essential.*

From this, my primary goals are: to provide clarity in code in this language, i.e. clear function types given in definition, prefix syntax for all functions to help visualise the function-argument relationship of terms in an expression, etc.; to allow more complex function programming features, such as higher-order functions, without compromising simplicity; and to provide all this using an intuitive and easy-to-use IDE interface.

In regards to the higher-order list functions i.e. map, reduce, and fold in Haskell, implementing these functions would require a support for lists first. Having read on <http://learnyouahaskell.com/> how lists are implemented in Haskell, it would appear that they are actually a form of data type, which are more complex structures based on existing fundamental types, and to support them properly would mean properly supporting data types in my language. Since Ralph doesn’t consider these list functions essential, I’m compelled to leave data types, and therefore lists, as a challenge for another time, and instead focus on delivering this language with a simpler feature set that only involves the fundamental types including integers, floats, etc.

# Objectives

* Accessible IDE UI, easy-to-use features
  + Code editor provides primary functionality of the program, and operates similar to a code editor like in Visual Studio
    - Text is colour-coded to indicate syntax
    - Red underlining for errors, like this in Word
    - Custom scrolling allows for scrolling to a page below the final line of code
    - Line numbers in left margin, only displayed up to last line of code
  + Tab control allows multiple instances of a code editor to exist
    - Every code editor instance has its own code, its own file path if saved etc.
    - Star \* displayed next to name if not saved
    - If new file, call it Untitled
  + General file functionality available from top menu, toolbar icons, and keyboard shortcuts
    - Opening, saving, closing files are all easily accessible functionalities
    - Unsaved files provide save prompt when closing, and if they haven’t been saved before, they allow for a file location to be chosen to save to
    - Option to run code available
  + Output window
    - Any compiler errors thrown are caught and displayed here, and should indicate where in the code the error was caused
    - When code is run, the output should be displayed here. Any runtime error caught also displays and indicates in what expression it failed
* Runtime pseudo-interpreter, which runs code from a file
  + Execution includes 3 stages
    - Tokenise
    - Compile
    - Execute
  + Tokeniser behaves as a precursor to the translator by using Regular Expressions to convert all valid code into tokens which can be more easily processed. Any major syntactic errors such as invalid words or characters get caught here e.g. 123hello isn’t a valid identifier
    - Tokens are split into types to allow the compiler to sort through them effectively, and the source code that the Regex matched is stored as part of the token object to be compiled if necessary
    - The tokeniser function returns an object with a list of errors for all untokenised code
    - The output token code is passed back using a C# out parameter
  + Compiler processes tokens and instantiates objects for any variables and functions that have been defined, generalised as expressions. More nuanced syntax errors are caught here e.g. not providing an expression with some value when defined, or not declaring variables with a valid type
    - Firstly expression definition objects for each function and variable defined need to be instantiated with the appropriate type given by the signature and with the right identifier
    - Then if the expression is a function, the parameters need to be identified and added to the local context of expressions to be used in a function definition
    - The compiler then needs to process what a given expression is set to be equal to, and ensure that the type of what the definition returns matches the type of the expression
    - The definition of an expression should be either a literal value or follow the structure of calling another function and passing it arguments, such that the return value of that function call matches that of the expression. The arguments passed can also be literal values or follow the same structure, effectively allowing nesting as part of the definition. For any element of the definition, condition blocks can be used to control what is called or passed
    - The compiler function returns an object with a list of errors for each line of code with a description of why compilation failed
    - The output expression objects are passed back using a C# out parameter
  + At execution, the main function/expression is placed on the call stack and run. This is done by “evaluating” the main expression, which recursively evaluated all expressions that it is composed of, down to the base functions of the language i.e. arithmetic, Boolean logic etc.
    - The components of the definition of an expression are kept in a stack and evaluated in Reverse Polish Notation
    - Firstly if the expression is a function, it must be passed arguments which then replace the parameter slots in the definition, which continues until the expression is able to be properly evaluated
    - Once parameterised, or if the expression wasn’t a function, the parts of the definition get popped from the stack one by one and placed in a working stack. If indicated that one of those components is a function that is being called, it takes arguments back from the working stack, operates on them, and then places the results back on the working stack. This continues until the final result is on the working stack

# Modelling

**IDE UI**

Tab control

New file

Open file

Close file

UI functionality

All features, buttons, shortcuts

Code editor

Saving

Scrolling

Line numbers

Highlighting

... Multiple instances

**Runtime and Compiler**

Tokeniser

Divide into tokens

Sort tokens

Return syntax errors

Compiler

Instantiate expressions

Define expressions

Return errors compile errors

Runtime environment

Evaluate expression in RPN

Consider condition statements, only evaluate necessary branch

Recurse through sub expressions until base cases

Throw runtime exceptions at failure

Source code

Output window

Compiler errors

Runtime output

Compiler errors and runtime output

# Potential for high-level programming technique

By itself, any kind of runtime environment simulator is complex and requires high-level problem solving strategy and algorithm building to create. In the case of this app, there are many places where high-level technique may be employed to achieve a good result.

As part of the tokeniser, the simplest way to create tokens for the code would be run through the source code and search for one type of token, then repeat will the remaining types, which would likely be a simple pattern-matching algorithm based on Regular Expressions. This leaves all the tokens unordered afterwards, and so they need to be ordered. Since the number of tokens could be large for big files with many lines of code, a merge sort would be effective and appropriate to implement for this. Coding a proper merge sort requires either recursion with proper base case checking, or advanced data structure usage and memory management i.e. stacks for tracking divisions/merges, and list operation.

The runtime environment almost certainly will implement recursion, as function definitions will often reference other functions or expressions which aren’t yet evaluated, sometimes even themselves (recursion within the functional programming language). As such, any method that is responsible for evaluating an expression will have to call the same evaluation method for another expression to be able to successfully run. In itself, a runtime environment such as this is a complex model which would delve into the intricacies of the functional programming paradigm in order to faithfully represent it and allow functional programming code to be executed.

The compiler is likely to be the most complex component of the entire runtime/compiler library, as is has to take source code and turn it into the symbolic form which the runtime environment would actually follow through and execute. Primarily, it will dynamically generate objects which represent expressions in code. Furthermore, it composes function definitions and expressions by aggregating these expression objects with each other i.e. a function that is composed of other functions would mean that its expression object is an aggregate of the expression objects representing the functions it is composed from. In order to achieve this however, there must be a complex pattern matching algorithm with many layers of depth in order to identify a function type, properly parse it, identify parameters, define the expression using reference to other expressions, all while ensuring the type of the definition matches the type given by the user, etc. which would demonstrate a level of understanding of the paradigm and how/why everything is written as it is in functional programming languages.

I am confident that the level of depth this project will help me learn functional programming as I have intended to do, by attempting to emulate it at its lowest level and test it with my own program and runtime environment.