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, plus a visible (and hopefully useful) call stack widget and I/O queues. 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.

Firstly a user interface could be implemented through Windows Forms, for example the RichTextBox form would be appropriate for the code editor, with the use of rich text formatting for syntactic highlighting and showing errors with underlines etc. (this would likely be the most challenging UI hurdle in the project.) Table forms can be used to show the function call stack and I/O queues, and menu strips can make the majority of the IDE’s functionality available from dropdowns.

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, reordered into Reverse Polish Notation to simply the run-through of code and reducing the need to hop about in it. Everything can be represented using the unsigned long data type, and the code would be an array of this data type. Unless the symbolic representation is required to be exported by itself without machine code (unlikely to be a feature as it reduces debug functionality), then different functions can be stored separately, not needing to cluster everything into a single string in a file.

Debugging would involve maintaining a link between the symbolic code and the source code, such that the IDE can track where in the source code the execution is occurring and highlight it to the user. The IDE then also needs to keep track of call stacks, something which would actually be fundamental to the execution of the symbolic code itself, but then also be displayed to the user. Basic run, pause, breakpoint, step-in/-over capabilities should be easily accessible from a debug toolbar. To aid in bug-hunting, all necessary information about execution should be made available – like the call stack – but also return values of functions stepped over and the I/O queues (the input queue should be available to the user when debugging to control input directly mid-execution). Syntax errors should be caught before execution, perhaps using an IntelliSense-style code check. More subtle bugs such as infinite recursion and unreachable states would need to be probed for by an algorithm.

# Research

Look at GHCi online, F# etc.

Look at AQA spec

Clear function type and purpose

Clear error messages for syntax

Prefix emphasises function and parameter relationships

# Objectives (old)

* Build an IDE
  + IDE user interface
    - Code editor (RichTextBox)
      * Syntactic highlighting, error underlining, other language-based code visualisations
        + An algorithm linked to the symbolic interpreter needs to be fed the source code every time the user stops typing (or finishes a keyword) in order to update the text box. 1

Identify keywords or variable names from a backlog of valid words

Words should be identified differently/separately in different contexts, so perhaps different word logs for different contexts, including for variables (i.e. local variables in different functions)

Attempt part-compilation in order to find syntax errors, and then highlight the word(s) causing the error, displaying it to the user

Some amount of inference should be made during this process, so that one error doesn’t prevent the rest of the code from being checked i.e. when an error is found, the compiler would need to pick up from wherever is next possible, and keep checking for errors

* + - * + During debug, code should firstly be unchangeable, and secondly the line (or perhaps more specifically function) being executed next should be highlighted

Association between symbolic code and source code needs to be remembered, such that the location of every function call can be tracked back to their reference in the source code, and be highlighted

* + - Debug menu(s) (collection of forms)
      * Access to all debug functions
        + Debug toolbar at the top of the window, should have Run, Pause, Step over, Step into, Step out of buttons
        + Next to code editor (perhaps implemented directly into textbox) should be column to insert breakpoints
        + Call stack should be displayed as a table with function names
        + I/O queues should be written horizontally
        + Table at the bottom of the IDE in debug mode should display important return values (similar to auto watches in Visual Studio, but doesn’t watch any variables). Whenever a the execution steps out of or over a function, its return value should be displayed with its function name and position in the call stack at return, and remain until its value is processed
  + Functional code execution
    - JIT compiler to symbolic code representation with links to source code for debug purposes
      * Order of symbolic code is RPN: arguments pushed to stack first, then function called to process them
      * Separate functions would be stored as arrays of unsigned long data types which represent symbolic code. All functions in a program would be also stored in an array, producing a list of functions all stored together. The compiler would need to produce an index as a lookup table of the functions as a separate object, so that the process knows which function it is calling, and what arguments and argument types are valid for it
      * Lowest level functions including impure functions involving I/O or base operations should be called similarly to user-defined functions but made distinct such that the process knows whether to call one of the functions in the list or search for a function built into the IDE. This fundamentally risks the purity of the functional programming paradigm as there is no real guarantee that the built-in functions (which would be coded in C#) don’t provide different inputs depending on state, so making them impure. However, this is necessary to some degree for I/O functions, but in general the built-in functions should be made as pure as possible
      * Symbolic code syntax:
        + Structured as unsigned longs
        + Long stores data type and data for argument to be pushed to argument stack

If data type is an array or string, single long to specify data type and array length, followed by longs to make array

* + - * + To call function, single long to call indicating if function is user-defined (therefore elsewhere in the list of functions) or built-in. For either, reference the right function by index, which would call the correct lookup table. This would also call a verification of the argument types in the call stack
      * Linking symbolic code to source code
        + Separate to symbolic code should be a parallel table where each function definition has its own list of calls it makes. In the symbolic code, each function call within a definition is labelled, and the table will associate those labelled called with locations in the source code which represent those calls made
    - Code execution using symbolic code
      * As the execution runs through the symbolic code, it needs to maintain the call stack and argument stack
        + Argument stack needs to be able to store various valid types as arguments
        + The call stack will need to replace the latest call instead of add on-top of it if that call simply returns the return value of another function (tail-recursion). This is to prevent clogging up the stack with unnecessary calls and allows for recursion

# Objectives (new, working on it)

* 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
    - Every file has a tab with its own editor instance
    - Text is colour-coded to indicate syntax
    - Red underlining for errors, like this in Word
    - Opening, saving, closing files are all easily accessible functionalities accessible from various locations i.e. the toolbar, the menu dropdowns, and keyboard shortcuts
* Runtime pseudo-interpreter, including debug functionality
  + 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
  + 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
  + 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 e.g. arithmetic, Boolean logic, if-else statements etc.

# Language syntax and semantics

* Primarily based on the syntax of Haskell
* No type inference, everything is explicitly typed
  + Every expression will fall under the type:
    - Integer
    - Float
    - Char
    - Bool
    - Array (technically *data* in Haskell, data structures composed of fundamental types)
    - Function composed of all the above
* Expressions are declared with a type signature followed by an identifier, then equalling some valid expression
  + **Type variable = value** e.g.  
    **Int myInt = 42**
  + **TypeSignature function = expression** e.g. **(Int -> Int) increment x = x + 1**
* \*all for now