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

Later

# Objectives

* 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