Evaluation

# The outcome

The primary goal of this project was to produce an IDE that runs functional code: this objective was met, almost entirely. Considering the IDE UI, each aspect analysed has been addressed in the Design section. File functionality was a simple implementation using the basic features of Visual Studio, including the accessibility for all the features; the code editor was managed to be created as a custom control and achieved all the objectives assigned for it.

One design goal as prompted by the intention to produce an IDE similar to Visual Studio was to implement syntactic highlighting, which unfortunately, as was addressed in the Design section, was impractical. The first solution added was error underlining which worked very well, but formatting text on the fly proved too costly to do, so the feature was never brought to fruition. How it could be though would need to involve using an engine much more powerful than WinForms and which allows deeper manipulation of the controls the user interfaces with.

Though not originally intended as feature, reconsidering the design after having implemented it leads to consideration of the potential to scroll horizontally – this wasn’t initially considered, but eventually realising that code lines could be written long enough to go off the side of the screen prompted attempting to enable sideways scrolling. Such attempts were fruitless, pushing the limitations of custom controls as part of the Windows Forms API beyond its already strained limit for implementing vertical scrolling. A compromise had to be made by enabling multi-line expression definitions using a backslash before a newline in the compiler; that rounds off the limitation presented by the problem, and as such the IDE is very complete, even if not as rich as it could possibly be.

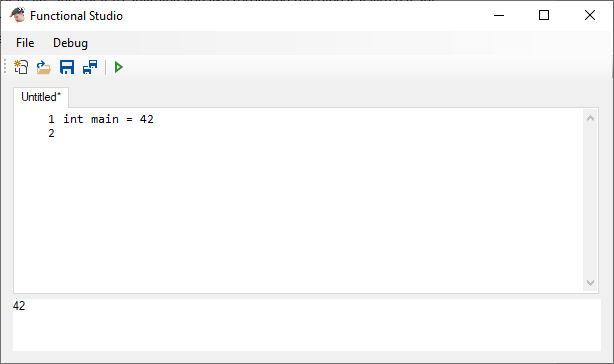
For the runtime environment and compiler, nearly every intended feature made it through to the final product, and that product still feels final despite this. The tokeniser and compiler serve no other purpose than to enable the user’s interaction with the runtime environment, which they do completely. Then the runtime environment itself does what it was intended to do, and though limited to a certain degree, was planned to be that way. Writing code to run code is hard, which I was aware of, and was I to attempt to implement the more complex data structures such as arrays, I was sceptical that I would be able to succeed. There does lack a feature, however (not necessarily being a part of the runtime environment itself, but of the IDE as a whole), which I only briefly addressed. Debug functionality is a key component of many IDEs, the definition of which can often be vague. In effect, debug functionality was successfully implemented in the form of compile-time and runtime-exceptions thrown back to the user in the output window. But debugging is more than just errors; often times debugging includes live tracking of execution, using breakpoints and code stepping. Eventually I realised that this was too tall an order to accomplish, and never attempted to add it, leaving debug functionality to the complete but limited feature-set of simple exceptions. Considering the application, however, such a solution could well be deemed efficient, potentially even all that is needed. The functional programming paradigm is built on the bases of purity and immutability, which is what allows complex implementation of many components at many levels without issues. Once a function is known to work for everything it can be tried it, *it works*, and there is no difference to it being used alone versus as part of a huge structure. Therefore the ability to debug at this humble level may be enough to debug these lowest components, and when these are known to work then there simply is not need to delve so deep again such as might be possible with real-time debugging when considering higher-level applications.

There is one aspect of the runtime environment which in fact has not been addressed at any point yet. It is a rigorous system, where almost every potential point of failure is addressed and the user is made aware of failure. This is the case for every potential point of failure but one, which itself cannot be considered by the CLI which the entire IDE runs on. When a user expression is defined and run which creates a case of infinite recursion, the IDE crashes – hard. No exception can be caught, as what is thrown is a stack overflow exception from the runtime environment in which the IDE resides; these indicate disaster, and that the application has broken completely. Rare, and only caused by poor user implementation, this kind of event is still unfriendly. There is, however, a potential way to implement a fix; given that this exception is called by attempting to evaluate an expression recursively, where the recursion is indefinite, then the amount of recursions can be counted. Up to an arbitrary number, one considered plenty for natural evaluation but safe enough from causing a crash, the amount of recursions will count; once the number is reached, the runtime environment will have had enough, and it throws its own handleable exception that can be presented back to the user. Such a counter would be implemented likely by passing it as an argument to the evaluation methods that comprise the expressions in the environment, and the arbitrary number could be considered by seeing how far the counter would go before a crash and adding a reasonable margin. This is how the one only possibility of the IDE breaking could be patched.

# End-user feedback

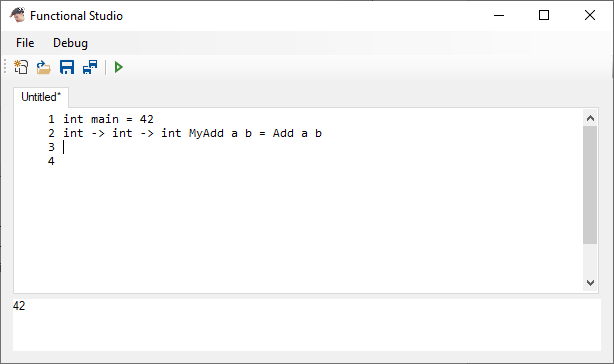
An appropriate end-user to test and provide feedback for my project was my computer science teacher Ralph. After giving him some guidance on using the IDE to create something tangible and that could be used to make calculations, he gave feedback on the app, and on how it met the requirements of the user as was specified in the Analysis, which in fact was aided by himself. His testing went as follows (comments in italics were my instructions):

*Every program in Functional Studio provides an output through a definition of an expression “main”. First write a definition for main, and set it to anything you like (provided the type is valid) e.g. int main = 42, or float main = 3.14, or bool main = true. The value of main should then be displayed in the output window when you run; this should give you an idea of how the app mostly interacts with the user.*

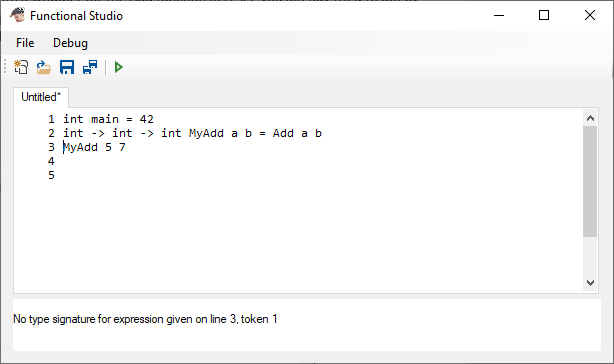


Yes, this works.

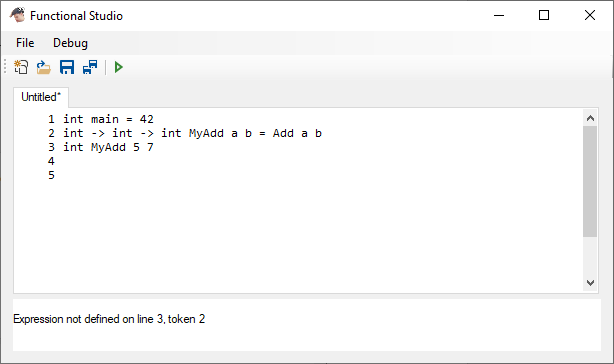
*Then you can define your own functions. Start with a wrapper for the Add function (as I see you already tried to do) by defining int -> int -> int MyAdd a b = Add a b (important that the type signature is valid, you initially defined it so it could only take one argument, hence probably the error.)*



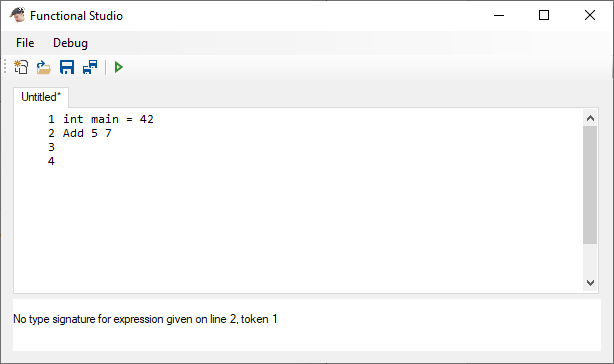
This doesn’t work:



Nor does this:



Nor does this:

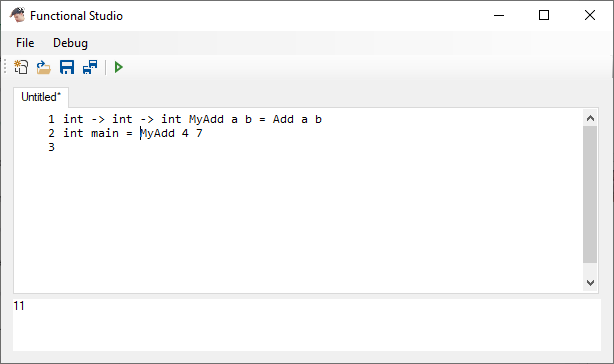


Following some further advice from developer

*return a result using main. Once you have defined your own MyAdd function, test it by changing the definition of main to equal MyAdd and some arguments. You cannot just “call” any function on some given line, since this isn’t like a console or shell where functions can just be procedurally called.*

*Try again with the MyAdd function, but instead of trying to call it randomly, assign it to main.*

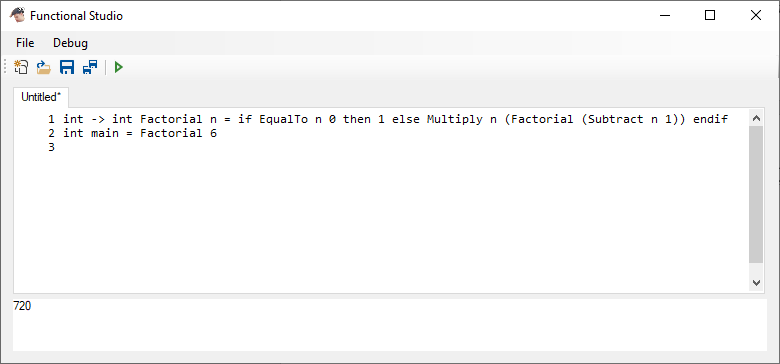
this does work:



*Cool, next would be a Factorial function. Define it as a single parameter int function (so int -> int) and using the if condition block to properly allow for a base case. It goes as follows:*

*int -> int Factorial n = if EqualTo n 0 then 1 else Multiply n (Factorial (Subtract n 1)) endif*

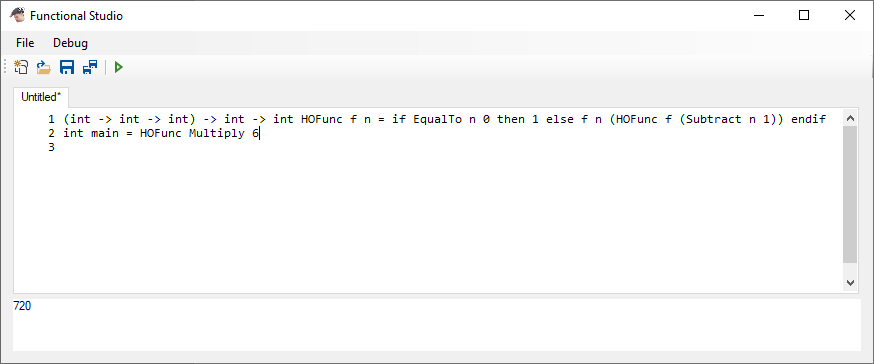
*Important to note the use of bracket nesting towards the end. You want to pass the result of the factorial function to multiply, not the function itself; same for how you want to pass the result of the subtract to the factorial, not the function itself.*



*Next to implement higher order functions, rename factorial to some other more appropriate name, maybe just HOFunc (higher order function). Redefine it as a function that also takes another function as a parameter, one of type int -> int -> int like add, multiply, subtract, etc. This makes the final type signature (int -> int -> int) -> int -> int, so the function takes another function and an integer as an argument, name them f and n for clarity. Replace the Multiply call with f, and add f as an argument to the recursion call (you won’t be calling Factorial this time, but HOFunc). Right before passing the Subtract result, pass f again so that it continues to use it.*

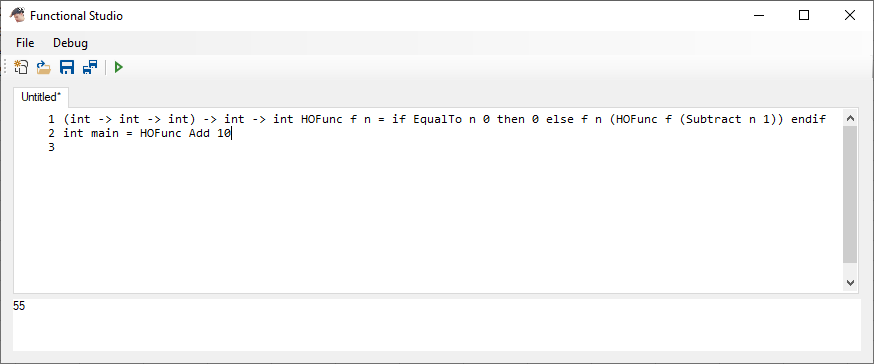
*This should give you a function that can take another function like Multiply and effectively fold it over the integer you pass it. Try then redefining Factorial using this, so it would look something like HOFunc Multiply n. You can then try defining other functions that use HOFunc but instead use Add or Subtract.*

HOFunc Multiply 6 = 6x5x4x3x2x1 i.e. factorial 6



HOFunc Add 10 = 10+9+8+7+6+5+4+3+2+1 i.e. 10th triangular number

(result of base case changed to 0)



His feedback on meeting the requirements of the user:

*Primary goals are:*

*(1) 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.*

I think this goal has been met. The syntax required is consistent throughout the language, reflecting some of the key principles of functional programming, including prefix expressions, use of function types and representation of the function argument relationship.

*(2) to allow more complex function programming features, such as higher-order functions, without compromising simplicity*

The facility to be able to use recursion with (or without) a function as an argument is a clear indication that this goal has been met.

*(3) to provide all this using an intuitive and easy-to-use IDE interface*

The IDE interface is easy to use and intuitive, although some sample code and instructions would have made this easier to use.

And from that, there is a confidence that this IDE is useful and potentially applicable to real-life scenarios such as the classroom when teaching functional programming. It has achieved what was intended, and what it lacks is known, and the potential solutions have been considered; I believe this project was a success.