Christian Wendlandt

Dr. Thomas Naps

CS 331

10 May 2018

Assessment Essay

Part 1:

For this assessment, I was assigned to produce the Hofstadter H-sequence in F#. F# is a functional programming language that prides itself on a concise syntax and a relatively simple grammar. To aid in its brevity, spaces, newlines, and indentation all carry syntactic significance. Spaces are used to delimit parameters and arguments, expressions are separated by newlines, and indentation is used for nesting blocks of code.

Notice that I used the word “expressions”, and not word “statements”. Since F# is not an imperative language, and is instead a functional language, lines are allotted to expressions which will evaluate as they’re read. Reading a sequence of captured expressions isn’t like assigning tasks to objects; it’s more like defining a problem piece by piece and then the last expression “solves” it. In fact, the last expression in a function is more significant than the rest because it helps define the function’s return type.

In that regard, F# is a type inferencing language. This, again, helps keep your definitions brief and reduces the number of compilation errors. It’s amazing with how complex you can make a function expression in as few as five tokens, provided that your tasks are properly factored.

Currying, composing, and applying functions are all one-liners in F# thanks to the fact that functions are first class entities in functional languages. This allows interfaces to be, far and above, more flexible than those that you would see in an imperative language. Passing functions into functions is the cornerstone of pattern matching. The primary use of a function is to repeat tasks with varied states, but with pattern matching, we can complete tasks with varied states and varied objectives. This allows us to abstract further out than we could have with just imperative languages.

All of this abstraction, type inferencing, and pattern usage will serve to create correct and easily readable code. Even in an imperative language, the most maintainable code tries to abstract away as much as possible; this is the goal of F#. If done correctly, functions should also be made to have no side effects. While “side-effectless” code is not always possible, the fact that F# works this way by default is going to significantly reduce the number of hair-loss inducing bugs that are able to show up your code. Without side effects, validating components becomes just a matter of validating the domain of one function at a time.

If each of your program’s components can be separated and validated individually, then you’ve reached the ultimate goal of modular design. In this ideal coding environment, a group of programmers can divide tasks by individual speciality, optimize components with autonomy, and reorganize existing modules into new products with ease.

Part 2:

I found that writing in F# was actually quite enjoyable. While taking the programming languages course and the compilers course at the same time, it occured to me that a highly abstractable language could be built from a very concise functional programming syntax. It’s funny to see that F# exemplifies these qualities in exactly the way that I imagined such a language would. With that said, its brevity lead to some confusion.

The lack of required return types sometimes put me in situations where incorrect types were being passed between functions. Using functions from the default library without fully understanding (or knowing about) their documentation was the most common culprit. In a statically typed language, this type of mistake wouldn’t even make it to compile time. In effect, statically typed languages are self documenting. Perhaps the concise nature of F# is outweighed by the fact that you will be required to comment almost twice as often when part of a dedicated project.

F#’s implementation of recursive functions was also a bit off putting. I didn’t think that “tying the knot” would be a strict requirement of the language, since recursion is so natural for lower level code (as in assembly). It’s still nice that the actual “knot” implementation is abstracted away by a special “let rec” keyword pair, but the added complexity still creates issues, particularly if you’re dealing with mutually recursive functions.

Honestly, I think that F# is easy to learn if you already know how functional programming works. It’s very easy to write a sequence definition with only a half-dozen lines of of code.