The #12in23 Challenge - F# - Trying F# in Functional February 2023

Written by Seth Corker on Benevolent Bytes

Banner photo by Michael Dziedzic on Unsplash

Why F#?

There were quite a few languages to choose from for Functional February, many of which I've used in some capacity. Elixir is my hobby language of choice, I've been tinkering with it over the past few years. I've also done small exercises in most of the other languages on the list. F# looked interesting because I discovered it years ago but never really looked much deeper. This is the perfect time to try it out.

Interesting language features

I discovered F# around the same time I learned about functional programming and everything was different, a little bit confusing and difficult to contextualise. Coming back to it now, it feels familiar. The concepts I've come to understand and the patterns I now use everyday has meant coming to F# again feels somewhat familiar. So what stands out?

Type system

F# is strongly typed and can infer type information. This perfect combo gives you the benefit of being able to identify problems with type mismatches without the boilerplate for constantly writing out type information. I've seem this before with Reason and as F# takes inspiration from ML languages, this feels right at home. Modelling your domain with types can be really powerful and when combined with type inference, it seems to work really well.

Option type

`null` or `nil` don't exist in F# this makes way for `None` and `Some(value)` instead. I really like this pattern in languages, it's explicit and when combined with pattern matching, it makes things much easier to reason about. It's much more coherent than JavaScript's `null`, `undefined` or `-1` roulette. ### Higher order functions Functions in F# can be composed and curried without ceremony.

Let's say we have the following:

```
let inc x = x + 1
let timesByTwo x = x * 2
let incThenDouble = inc >> timesByTwo

printfn "%i" (inc 1)
printfn "%i" (timesByTwo 1)
printfn "%i" (incThenDouble 2)
```

- `inc` increments a value by 1
- `timesByTwo` multiplies a value by 2
- `incThenDouble` increments a value then multiplies it by 2. Notice, we didn't give the function any arguments because both functions expect a single argument we get a a function that also accepts a single argument.

Here's a another example:

```
let wrap s = $"[{s}]"
let prefix pre s = $"{pre}: {s}"
let errorMsg = prefix (wrap "ERROR")

printfn "%s" (errorMsg "something went wrong")
```

We can create a new function, `errorMsg` which returns a partially complete function - `prefix`, where the first argument will be `[ERROR]`. F# makes it simple to create these partial functions that can be reused.

Pattern matching

Pattern matching with types allows you to make invalid states unrepresentable which is fun and it forces exhaustive matches etc. It's much easier to parse when reading than if statements and you can add guard clauses for more specific matches.

Here's a simple match, it's pretty similar to match statements you'd find in other languages.

```
let x = "espresso"

match x with
| "espresso" -> printfn "don't add milk"
| "flat white" -> printfn "add milk"
| _ -> printfn "not sure, leave it out just to be safe"
```

Here's an example with some more pattern matching.

```
let letters = [ Some("bill"); None; None; Some("advertisement"); Some("unknown") ]

let sortLetter input =
    match input with
    | Some("bill") -> "Put in tray A"
    | Some("advertisment") -> "Put in tray B"
    | Some(other) -> $"{other} should be read first before sorting"
    | None -> "Forward this on, it's not ours"

letters |> List.map sortLetter |> List.iter (printfn "%s")
```

We can match on `Some` or `None` and also pull out the value if it's a `Some`. The compiler also gives a helpful hint as you write these matches because it's a ware of what paths are covered and which aren't. Leaving off the final `None` clause shows this warning: > Incomplete pattern matches on this expression. For example, the value 'None' may indicate a case not covered by the pattern(s).

Developer experience

Getting started

I used to work with .NET and C# professionally and the experience in those days was very focused around Visual Studio and Windows. Getting anything working on macOS or Linux was much more challenging. Nowadays, dotnet is much easier to work with across multiple platforms. The command line tool follows standard *NIX conventions and you can simply download the .NET SDK and get started. It includes project creation from the CLI where you can specify the language as `F#` and you're good to go.

Editor integration

The experience of working with F# in VSCode is great. Ionide for F# works really well, it has autocomplete and shows type information as well as parameter hints. It's one of the easiest setups for a programming language I've seen in VSCode. Everything just works.

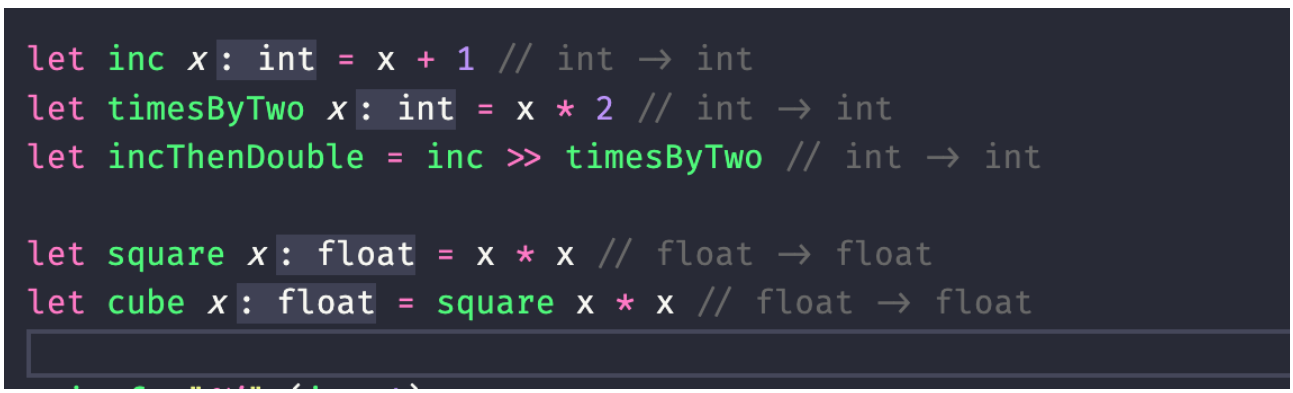

Figure 1: image

You can see inferred type annotations in Visual Studio Code as seen above.

Ecosystem

The biggest win about F# is that it's part of a bigger ecosystem. Much like Elixir, leverages the ecosystem of the BEAM, F# leverages the .NET ecosystem. Having access to C# libraries is a cool trick, it allows you to tap into a large ecosystem of existing packages or integrate it inso an existing .NET project. I tried this out by using Avalonia to try out a UI and ran it on macOS.

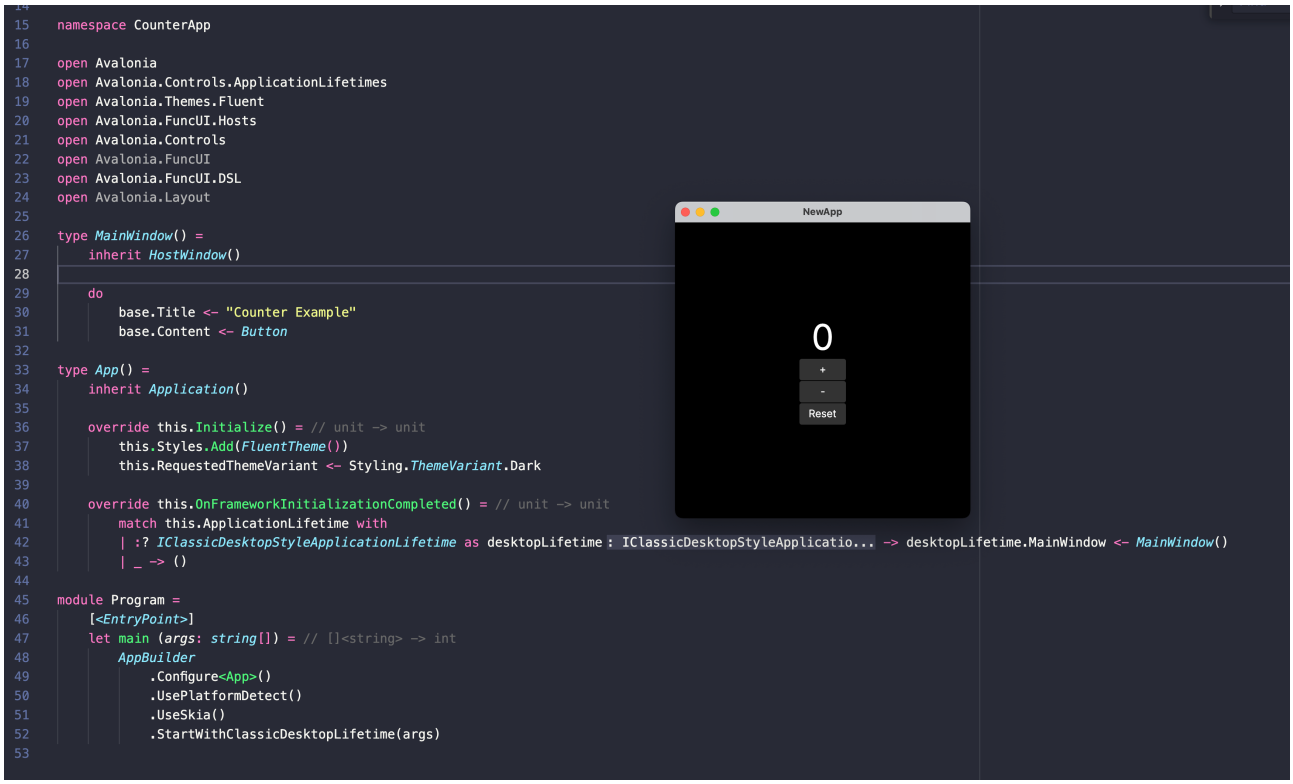

Figure 2: image

Takeaways

F# is an interesting language that gives you the functional familiarity you're used to without leaving you stranded in a small ecosystem. It's got some great features like pattern matching and has removed the concept of null. It makes it a fun language to try and by leveraging the existing .NET ecosystem, it can be introduced more easily into existing projects and leverage existing packages. F# was a fun excursion in functional February, I'm glad I spent some time tinkering with it.

Resources

- F# Software Foundation
- Learn F# Microsoft
- F# on Exercism