1 An Introduction to F#

F# is a functional-first programming language. That is, F# is a programming language which emphasizes the functional aspect of programming to a greater extent than other languages you may be familiar with such as C#, Visual Basic .NET, Python and JavaScript. You will already be quite familiar with functions: you met them in high school when you drew a plot of the function,

$$y = x^2$$

using your trusty Sharp EL 9300 graphing calculator or your favorite plotting program on the WWW (e.g. Desmos at https://desmos.com/calculator). Functions as opposed to objects are the first-class citizens of functional programming languages like F#. They allow one to think about and solve problems using features prominent in functional programming such as immutability, pipelining, partial application, type-inference, and strong typing. (Note that these features are not unique to F#, they exist in other functional programming languages, and indeed in ordinary imperative programming languages such as C#, Visual Basic .NET, Python and JavaScript).

F# was created by Don Syme who is an Australian Computer Scientist and Principal Researcher at Microsoft Research, Cambridge, UK. He is still very active in the F# community and makes contributions to the F# language to this day (March 24th 2021). You could call him the Grandfather of F# or the Supreme Leader of the F# community.

F# allows one to think about a particular programming problem or task in a clean and simple way. It has the clean simplicity of Python while retaining the performance and type safety found in languages such as C++ and C#. F# can be used to write small programs with specific goals set at the start to large systems with open-ended goals.

F# is fully open source and fully cross-platform. F# is integrated within Visual Studio and so can make full use of all the integrated development environment's (IDE's) features such as Intellisense, packages management, documentation browser, etc. Alternatively, The Visual F# Tools can be downloaded from https://github.com/Microsoft/visualfsharp to developer F# programs without an IDE. On top of that F# code can be run in Jupyter Notebooks hosted on Microsoft Azure Services.

I also use Visual Studio Code sometimes because it is cross-platform and has about a million different extensions from Markdown edting support to copy

code/text with line numbers and everything else you can think of in-between.

1.1 Hello, World!

As is traditional when learning most programming languages we will begin by writing a “Hello, World!” program. To do so first download and install the .NET SDK. You search the Internet for instructions how to download and install the .NET SDK. You are also free to choose whichever OS you desire.

Open a command terminal and type in

```
dotnet new console -lang "F#" -o hello
```

`cd` into the directory hello which was created when you typed in the above command. Then, open the file `Program.fs` which was created when the directory was created. It should look something like:

```
 01: // Learn more about F# at
http://docs.microsoft.com/dotnet/fsharp
02:
03: open System
04:
05: // Define a function to construct a message to
print
06: let from whom =
07:     sprintf "from %s" whom
08:
09: [<EntryPoint>]
10: let main argv =
11:     let message = from "F#" // Call the function
12:     printfn "Hello world %s" message
13:     0 // return an integer exit code
```

Replace the contents of the file with:

```
1: printfn "Hello, World!"
```

Now go back to the terminal and type in:

```
dotnet run
```

Pretty cool! Right? One line of code is all that's needed to print out a message to the console and a one line command is all you need to build and run the executable that prints out the message. That's a lot better than C or C++ and just as good as Python or any other scripting language.

1.1.1 But Wait!

I know what you're thinking!: There is something wrong here! Where is the function? There is a `printfn` function being called here which is taking a single string parameter as input and then outputting that string parameter to the console. But where is our function? Isn't F# supposed to be a functional-first programming language where nearly everything is a function? That is true. But in this case to make things easier to do like simple scripting, the F# compiler is creating a "main" function for us which is then called by the .NET runtime. To make this more explicit we can wrap the "Hello, World" code inside of a function ourselves. So let's do that.

Change the code to the following:

```
1: open System
2:
3: [<EntryPoint>]
4: let main(argv) =
5:   printfn "Hello, World!"
6:   0
```

Type in `dotnet run` to check that everything still works fine.

The code should be self-explanatory. Line 1 "open"s and makes the classes and objects in the .NET System library available for use by our little program. Line 3 sets the entry point for our program. Line 4 defines our function which takes in a string argument list as its one and only parameter. This parameter is not used in

our program. Line 5 prints our "Hello, World!" message and finally line 6 returns the exit value. Notice that on line 6 we don't need to write "return 0" as in some other languages that you may be familiar with. You just write down the actual return value and that's it!

1.1.2 Take a Tour of F#

Now that you've written and run "Hello, World!" in F# you should take a tour of F#. Start at the following URL http://docs.microsoft.com/en-us/dotnet/fsharp/tour. Once you feel ready to continue reading this book come back here and do the exercises in the following section.

1.1.3 Exercises

1. Modify the "Hello, World!" program to ask the user for his/her name and print out "Hello <name of user>!" instead of "Hello, World!".
2. Write a function which will take two numbers and add them together and return answer.
3. Write a function which will compute the factorial of a number. You will need to research how to write recursive functions in F#.
4. Write a function which will compute the n^{th} term in the Fibonacci sequence.

1.2 The dotnet Command

You should also familiarize yourself with the `dotnet` command as you will be using it a lot. You can get help by typing in `dotnet -h` and also by searching and reading the Microsoft documentation online. Here are a few exercises that you can try after you've familiarized yourself with the `dotnet` command.

1.2.1 Exercises

1. What is the default build configuration?
2. In which directory is the output stored?
3. What (SDK) command do you need to use to clean the output inside the build directory? Which files/directories still remain after you've cleaned the build directory?
4. What command do you need to type in to build a Release version of the "Hello, World!" application?
5. How do you run the Release version of the application.
6. How do you clean the Release build?

1.3 What Next?

By now you should have a fairly good grasp of functional programming and F# in particular. You could stop reading this book now and continue your exploration of F# as needed for the particular set of projects that you are working on.

So what value does this book on F# have? Well, read on and you can decide for yourself whether the book fulfills any purpose and hence has any value. The remainder of this book contains mini-projects on various interesting topics from UIs to Neural Networks to 3D Computer Graphics. Each of the projects was chosen to illustrate how to solve a particular type of problem with F#. In the next chapter we will tackle the problem of writing UIs with F#. By the end of the chapter we wil have a fairly functional fractal image generator for Julia and Mandlebrot sets. So without further delay let's get started!

1.4 Summary

In this chapter your wrote your first F# program!: A simple "Hello, World!" program. To do this you setup your system to write F# programs. You downloaded and installed the .NET SDK (if you didn't already have it installed) and then created a new console project. You then modified the code to print "Hello, World!" to the console.

You then took a tour of F# from around the web and returned to the book to complete some simple exercises. You also read some documentation on the `dotnet` command and played around with the command a little. You could say you are a beginner F# programmer, maybe even an intermediate F# programmer who is fairly competent at thinking about problems in a functional way. You are ready now to tackle what every programmer dreads and loves at the same time: UIs!

2 Writing UIs with F#

There are a number of paths available to programmers who decide to write UIs with F#:

1. Avalonia.FuncUI (Cross-Platform)
2. Elmish.WPF (Windows Only)
3. Use Windows Forms or WPF directly (Windows Only)

We will take the option 1, the Avalonia.FuncUI route since it is cross-platform. Avalonia.FuncUI is fairly intuitive to use and will suit our purposes admirably. If you want, you can explore the alternatives starting at http://fsharp.org/use/desktop-apps/.

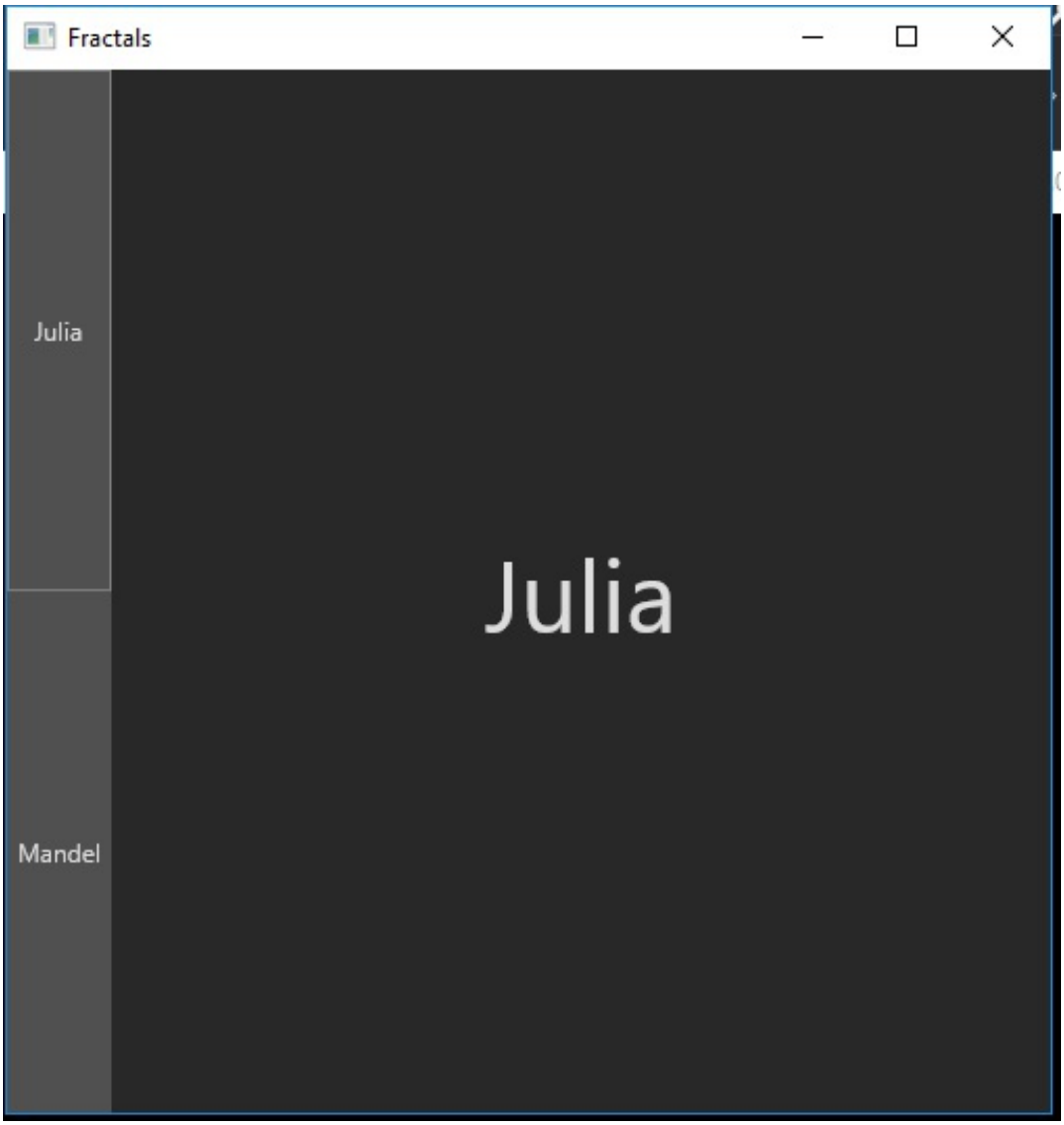
First read a little on how to get started with Avalonia.FuncUI at the following URL:

https://avaloniacommunity.github.io/Avalonia.FuncUI.Docs/

Install Avalonia.FuncUI and then afterwards create a new "Basic Template" project. Check to make sure everything compiles and runs smoothly. A description of the files created in the "Basic Template" project is at the URL:

https://avaloniacommunity.github.io/Avalonia.FuncUI.Docs/guides/Basic-Template.html

Modify the title of the MainWindow class to "Fractals" and then change the width and height of the window to 500 and 500 respectively. Next modify `Counter.fs` to display a window like the one shown in the figure below:



If you get stuck, or want to check your code against mine here is what I wrote:

```
01: namespace Fractals
02:
03: module Counter =
04:     open Avalonia.Controls
05:     open Avalonia.FuncUI.DSL
06:     open Avalonia.Layout
07:
08:     /// The fractal type.
09:     type FractalType = Julia | Mandelbrot
10:
```

```
11:    /// The state.
12:    type State = { ft : FractalType }
13:
14:    /// Initializes the fractal type.
15:    let Init = { ft = Julia }
16:
17:    /// Updates the fractal type.
18:    let Update (msg: FractalType) (state: State) : State =
19:      match msg with
20:      | Julia -> { state with ft = Julia }
21:      | Mandelbrot -> { state with ft = Mandelbrot }
22:
23:
24:    let View (state: State) (dispatch) =
25:        DockPanel.create [
26:            DockPanel.children [
27:                DockPanel.create [
28:                    DockPanel.children [
29:                        Button.create [
30:                            Button.dock Dock.Top
31:                            Button.height 250.0
32:                            Button.onClick (fun _ -> dispatch Julia)
33:                            Button.content "Julia"
34:                        ]
35:                        Button.create [
36:                            Button.dock Dock.Bottom
37:                            Button.height 250.0
38:                            Button.onClick (fun _ -> dispatch Mandelbrot)
39:                            Button.content "Mandel"
40:                        ]
41:                    ]
42:                ]
43:                TextBlock.create [
44:                    TextBlock.dock Dock.Right
45:                    TextBlock.fontSize 48.0
46:                    TextBlock.verticalAlignment VerticalAlignment.Center
47:                    TextBlock.horizontalAlignment HorizontalAlignment.Center
48:                    TextBlock.text (string state.ft)
49:                ]
50:            ]
```

```
51:         ]
```

If you've written your code correctly here is what should happen when you start Fractals.

1. A window should pop up like the one in figure 1.
2. Pressing the "Mandel" button should change the text in the TextBlock to "Mandelbrot".
3. Pressing the "Julia" button should change the text in the TextBlock to "Julia".

If something doesn't work as expected then check your code carefully for any mistakes and compare your code with mine if necessary.

Everything working correctly then? Great! Let's continue then.