A Brief Introduction to F#

This tutorial gets you started learning the F# programming language, which is a modern version of the highly influential ML programming language. We focus at first on the F# programming environment and basic syntax. In subsequent reading, we will dive deeper into F#.

Most students find F# to be foreign at first. It will require you to think in a new way about programming. However, even if you never program in F# again, it will very likely influence your programming in positive ways. After I discovered F#, I wondered why more conventional languages like Java and C++ had to be so complicated. The short answer is: they don't have to be!

Note that if you have F# installed on your computer, you should be able to follow along by starting the F# interpreter[16] and then by typing expressions into your console.

[16] Type `dotnet fsi` on the command line. To quit, type `#quit;;`. You will sometimes hear me refer to this interpreter as a "REPL," which stands for "read-eval-print loop."

Let's look at our favorite starter program, but written for F#.

```
printfn "Hello world!\n"
```

That is the entire program. Refreshing, isn't it?

What is F#?

F# is a *functional* programming language. A functional programming language differs in form than a conventional programming language like C[17] or Java[18]. Even if you decide that functional programming is not for you, exposure to functional programming ideas will change the way you think about coding.

[17] C is an *imperative* programming language.

[18] J is an *object-oriented* programming language. Object-oriented languages are also usually imperative, and this is true for Java.

Functional programming encourages *expressions* over *statements*, *immutable* instead of *mutable* variables, and *pure, first-class functions* instead of *side-effecting* procedures. F# is also *strongly typed* unlike C, which is *weakly typed*, and Python, which is *dynamically typed*. These differences contrast sharply with those encountered in languages like C, Java, or Python. The end result is that functional programs read more like mathematical statements than a sequence of steps. Let's briefly touch on each of these concepts.

Immutable variables

In a language like Python or C, a variable can be declared and written to many times. E.g.,

```
x = 2
x += 1  # the value of x is now 3
x += 1  # the value of x is now 4
x += 1  # the value of x is now 5
```

In a functional programming language, a variable can only be written to once, when it is declared.

```
let x = 2
x += 1   // can't do this in F#; will not compile
```

You might be wondering how on earth you "update" data. It's done like this:

```
let x = 2
let y = x + 1
```

where x and y are *not* the same variable.[19]

[19] In other words, data is *never* updated!

Variables in F# are *immutable*, meaning that once they are declared, their values will never change. If you're like me, this idea probably has left you scratching your head. Good. The value of this model of programming will become apparent to you in time.

Expressions

In a language like Python or C, a line of code can either return a value or not. For example, in Python:

```
x = 2   # returns nothing; this is a statement
x + 1   # returns the value 3; this is an expression
```

In a functional language, all language constructs are expressions.

```
let x = 2  // returns a binding of the value 2 to the variable 'x'
x + 1      // returns the value 3
```

When a line of code returns nothing, we call it a *statement*. Since it is pointless to have a line of code that does nothing, a statement does something by *changing the state of the computer*. Changing the state of the computer independently of a return value is called a *side effect*. Side effects are either banned in functional languages (e.g., pure Lisp, Haskell, Excel) or strongly discouraged (e.g., Standard ML, F#).

Pure, first-class functions

A *pure* function is a function that has no side effects. In F#, we usually write pure functions.

In C, one can write the following:

```
int i = 0;

void increment() {
    i++;
}
increment();  // i has the value 1
```

Observe that the `increment` function takes no arguments and returns no values and yet, it does something useful by altering[20] the variable `i`. One is not permitted to write code like this in a functional programming language because variables are immutable and functions are pure. Instead, one might write

[20] The technical term is *mutating*.

```
let increment n = n + 1
let i = 0
let i' = increment i  // i has the value 0; i' has the value 1
```

where `i` and `i'` are different variables, and where `increment` is a *function definition* for a function called `increment` that takes a single argument, `n`. Function calls look a little strange in F#, so recognize that it might be a little while before you are good at recognizing their form. It often helps to rewrite a program to use explicit parentheses and type annotations:

```
let increment(n: int) : int = n + 1
let i: int = 0
let i': int = increment(i)
```

This is also a valid F# program—in fact, it's exactly the same program—and if you find yourself struggling with syntax, I encourage you to write in this style instead.

Function definitions in F# are also *first class values*. What does that mean? Among other things, any first class value can always be assigned to a variable. So yes, you can assign a function definition to a variable.[21]

[21] Most students struggle with this concept, but it is very important. If you're struggling to understand this idea, this is a great topic of discussion for class or help hours.

For instance,

```
let increment(n: int) : int = n + 1
let addone = increment
addone(3)  // returns 4
```

The type of the variable `addone` is a function definition (specifically, a function that takes an `int` as input and returns an `int`, or as we say for short "a function from `int` to `int`"), and since it's a function we can *call* it just as we would call `increment`.

Since values and variables can be passed into functions, one can pass variables of "function type" into functions as well:

```
let increment(n: int) : int = n + 1
let doer_thinger(f: int -> int, n: int) : int = f(n)
doer_thinger(increment, 3)  // returns 4
```

And, just for fun, let's get rid of the unnecessary syntax so you can see how simple this program can look:

```
let increment n = n + 1
let doer_thinger f n = f n
doer_thinger increment 3  // returns 4
```

Strong types

F# is a *strongly-typed* programming language. A strongly-typed language is one that enforces data types strictly and consistently. That means that the following kinds of programs are not admissible in F#. For example, the Python program,

```
x = 1
x = "hi"
```

or the C program,

```
int x = -3;
unsigned y = x;
```

Even with all the warnings enabled, a C compiler (like `clang`), won't flinch: no errors or warnings are printed for the above program. Nevertheless, it doesn't make sense to disregard the fact that an `int` is not an `unsigned int`, because assigning `-3` to `y` dramatically changes the meaning of the value. `y` is very much not `-3` anymore[22].

[22] If you know some C, try running a little experiment to see what happens.

Both of the above programs would be considered *incorrect* in F#, since both contain `type errors`. Neither program will compile. To convert from an integer to an unsigned integer, we must explicitly convert them in F#:

```
let x: int = -3
let y: uint32 = uint32 x
```

Strong types help you avoid easy-to-make but costly mistakes.

Other features

F# has many other features, such as garbage collection (like Java), lambda expressions, pattern matching, type inference, concurrency primitives, a large, mature standard library, object-orientation, inheritance, and many other features. Don't worry if you don't know what these words mean now—we will discuss these features throughout the remainder of the semester.

Microsoft .NET

F# is a part of an ecosystem of languages and tools developed by Microsoft called .NET (pronounced "dot net"). Programs written in .NET are almost entirely interoperable, meaning that different parts of the same program can be written in different languages. For instance, I routinely write software that makes use of modules written C#, F#, and Visual Basic combined into a single program.

.NET is also *portable*, meaning that it can run on many computer platforms. Unless you specifically seek to write platform-specific code, .NET code can be run anywhere the .NET Common Language Runtime (CLR) is available. This language architecture is similar to, and heavily inspired by, the technology behind the Java Virtual Machine (JVM). The .NET Core CLR is available on Windows, the macOS, and Linux. Additional platforms (like Android, iOS, and FreeBSD) are supported by the open source Mono project.

We will be using the .NET Core framework on Linux for this class. If you would like to install .NET Core on your own machine, you may do so by downloading the installer[23].

[23] `https://www.microsoft.com/net/download`

Modularity

One feature that we will address right away is F#'s strong support for modularity. *Modules* are a way of organizing code so that similarly named functions and variables in different parts of code do not conflict. In C, libraries are imported by the C preprocessor by performing the moral equivalent of pasting code from an included library libraries into a single file and then compiling that. As a result, it is easy to accidentally give two different function definitions the same name, a so-called *name conflict*. Name conflicts are an annoying and commonplace occurence in C. In F# and other .NET languages, name conflicts are impossible, because names are *scoped*, meaning that they only have meaning within certain boundaries.

F# has a variety of constructs available to scope names: solutions, projects, namespaces, and modules. For now, we will focus on projects.

A *project* is a unit of organization defined by .NET. A project contains a collection of source code files, all in the *same* language. A project is either a *library*, meaning that it must be called by another project, or an *application*, meaning that it has an *entry point* and can run by itself.

Creating the HelloWorld project

In order to provide some structure for our hello world program, let's generate an *application project*. Having an application packaged in this way makes it self-contained and easy to manage during the development process.

We create new F# projects using the `dotnet` command on the UNIX command line. Because `dotnet` creates a project in the existing directory, you should first create a directory for your project.

```
$ mkdir helloworld
```

Now `cd` into the directory and create the project.

```
$ cd helloworld
$ dotnet new console -lang "F#"
```

By default, the above command will generate a Hello World program.

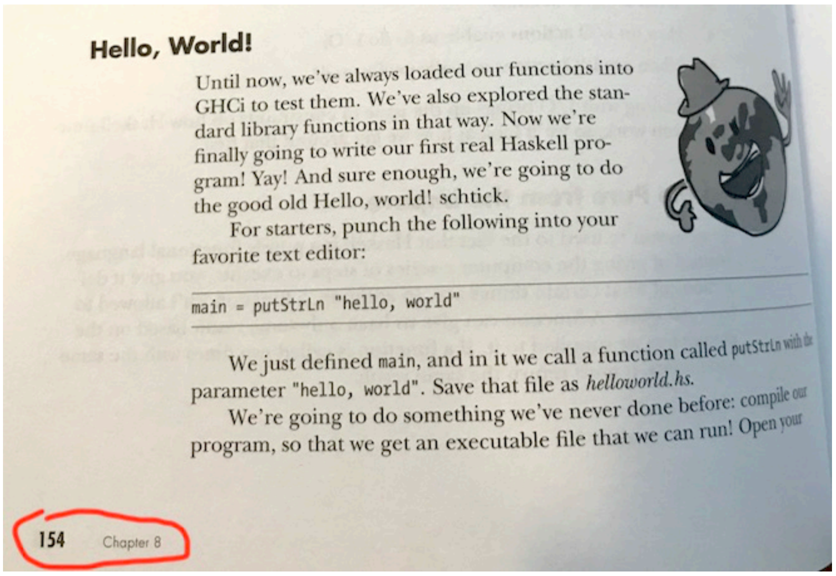
```
// Learn more about F# at http://fsharp.org

open System

[<EntryPoint>]
let main argv =
    printfn "Hello World from F#!"
    0 // return an integer exit code
```

There's a little more boilerplate here than we saw when using the `dotnet fsi` REPL, and it is mostly unnecessary. But we will keep it around because it makes working with arguments a little easier.

F# is a "whitespace sensitive" language, like Python. That means that the scope of a function definition is determined using rules about indentation instead of relying on curly braces. Thus the last line in the above `main` function is the expression `0`. The last line of a function definition denotes the function's return value, so this function returns `0`[24].

[24] When a program's `main` function returns `0` it informs the operating system that everything went A-OK. A non-zero return value indicates a failure. We'll talk more about this in the context of C.

One last thing. Since every language construct in F# is an expression, `printfn` is an expression. However, it falls into an special class of side-effecting expressions. Input and output are inherently side-effecting, and so any functional language that does not allow at least some side effects is seriously constrained in terms of expressiveness. Pure functional languages like Haskell have a clever but somewhat byzantine system for dealing with side effects, which is why the first "hello world" program in my "easy-to-read" Haskell programming book appears on page 154. Right after the section on "functors," of course ¯_(ツ)_/¯.



Hello, World!

Until now, we've always loaded our functions into GHCi to test them. We've also explored the standard library functions in that way. Now we're finally going to write our first real Haskell program! Yay! And sure enough, we're going to do the good old Hello, world! schtick.

For starters, punch the following into your favorite text editor:

```
main = putStrLn "hello, world"
```

We just defined `main`, and in it we call a function called `putStrLn` with the parameter `"hello, world"`. Save that file as *helloworld.hs*.

We're going to do something we've never done before: compile our program, so that we get an executable file that we can run! Open your

154 Chapter 8

Compiling and running your project

Compile your project with:

```
$ dotnet build
```

You may also just run the project, and if it needs to be built, `dotnet` will build it for you before running it.

```
$ dotnet run
```

I personally prefer to run the `build` command separately because the `run` command hides compiler output. I like to see compiler output since it will tell me if it finds problems with my program. Unlike other languages you may have used, F#'s compiler generally produces very good error messages.

Code editors

You are welcome to use whatever code editor you wish on this assignment. Two in particular stand out for F#, however: Visual Studio Code and `emacs`. Both are installed on our lab machines. Note, however, that we will strictly manage our projects using the `dotnet` command line tool.

Visual Studio Code

Visual Studio Code works out of the box with F#, but an extension called Ionide[25] adds additional features like syntax highlighting and tooltips to your editor. To install Ionide, follow this tutorial on installing extensions[26].

[25] `http://ionide.io/`

[26] `https://code.visualstudio.com/docs/editor/extension-gallery`

Note: Ionide comes with a variety of build tools such as FAKE, Forge, Paket, and project scaffolds. Please do not use these tools for this class as they do not interoperate well with our class environment. Instead, please use the `dotnet` command line tool to compile and run your tool as discussed earler.

`emacs`

If you prefer `emacs`, you can add the `fsharp-mode` which adds syntax highlighting, tooltips, and a variety of other nice features. I personally prefer this environment, but I understand that `emacs` is not everybody's cup of tea.

If using `emacs` on a lab machine, try pasting the following into `~/.local_emacs`:[27]

[27] On your personal machine, use `~/.emacs` instead.

```
(require 'package)
(add-to-list 'package-archives '("melpa-stable" . "https://stable.melpa.org/packages/") t)
(package-initialize)
(package-refresh-contents)

;;; Install fsharp-mode
(unless (package-installed-p 'fsharp-mode)
(package-install 'fsharp-mode))
;;; Run fsharp-mode
(require 'fsharp-mode)
```

The above will install both MELPA, which is an online package repository for `emacs`, and the `fsharp-mode` package. Note that MELPA has many other modes you can install if you like what you see. One downside to MELPA is that it adds a few seconds of startup time to `emacs`, but in my opinion, the delay is well worth the wait.

The next time you start `emacs` with F# code, you will see the new mode in action.