Preface

THIS COURSE takes you on a deep dive through programming language theory and implementation. To an outsider, the phrase "programming language theory" rouses thoughts of dry, technical debates carried out in quasi-mathematical notation. Indeed, most students who take this course take it because they have to. Maybe that's why you're here.

Regardless, I hope that for just one semester you can suspend judgement about this discipline. Why? For starters, I think that language designers are privileged to explore one of the most important questions in computer science: how do you best ask a computer to do what you want? Our aim is nothing less than to elevate humanity by improving access to computing. There are so many unexplored ways we *might* communicate with a computer that I personally find little else more exciting. To get the most out of this class, briefly permit yourself to imagine life as a professional language designer.

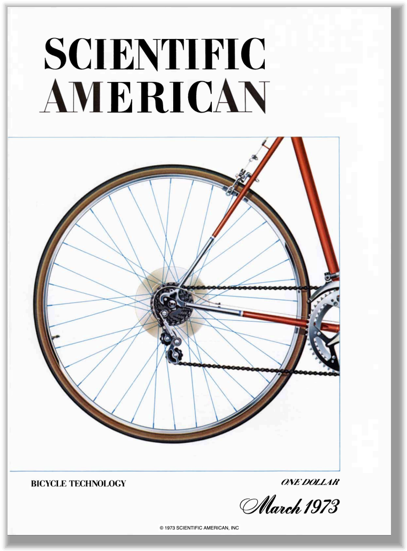

Figure 1: Jobs analogy was inspired by a Scientific American article showing that a bicycle is the most efficient known mode of transportation, at 0.15 calories per gram per kilometer. A cyclist expends roughly 5× less energy than a person who walks the same distance. It is a large force multiplier.

If you've taken CSCI 237, you have firsthand experience writing programs in a computer's "natural language." It is fiendishly difficult. Even when armed with powerful AI tools, writing good machine language programs remains frustrating.[1] Consequently, few programmers use or even know any machine language. Instead, most of us communicate with a computer using a programming language.

[1] I have struggled to goad ChatGPT into producing functioning machine language programs of any real-world complexity. That a computer program has trouble writing programs in its own native language is deliciously ironic.

Unlike a natural language, which comes about via processes that we only vaguely understand, a programming language is intentionally designed to serve a purpose. Although every widely-used language contains at least a few bad design choices, by and large, that purpose is to act as a force multiplier. Steve Jobs liked to call the computer a "bicycle for the mind" (Fig. 1). Computers do for mental effort what bicycles do for physical effort. [2] The analogy is a good one, but it's not entirely correct. Without a programming language, programming a computer is exhausting and slow. A good programming language makes many kinds of computation easy, even delightful. A good one can make you excited to program. Some can even change the way you think. A better analogy is that a *programming language* is a bicycle for the mind.

[2] S. S. Wilson. Bicycle technology. *Scientific American*, 228(3):81–91, March 1973

An important motivation for this course—and why it is a core requirement at Williams—is the recognition that the landscape of programming is constantly changing. I am frequently asked why we don't teach some programming technology x. Most recently, students have asked "Why not Torch?" In the recent past, it was "Why not React" or "Why not Rust?" Further back it was "Why not TypeScript" or "Why not Scala?" At the risk of sounding like an old person, when I was in school I was upset that we were spending time learning C++ when I just really wanted to know Ruby on Rails. If you want to learn a technology, you should certainly take the time to learn it. Nevertheless, our time in class is better spent on the most impactful ideas common to all technologies, because any given x is likely to change again by the time you graduate.

One of the most important foundations in this course is the "language of languages" spoken by the small group of computer scientists who design and implement programming languages. Fluency in this language enables you to look at any given programming language and understand what it really does beyond the marketing fluff. Moreover, in this course, you will experience firsthand what it takes to design and implement a programming language from scratch. Finally, you will gain experience programming F#, a language that I consider not just well-designed, but beautiful. With these skills you will have no trouble quickly learning new technologies on your own.

For now, I ask that you keep an open mind. This class will expose you to new ways of thinking about computation. You may feel as if you are unlearning lessons that you worked hard to learn in the first place. The experience will be uncomfortable at times. If, at the end of the semester, the new ways don't appeal to you, you are welcome to go back to your comfort zone. It's your life and the computer doesn't care how you program it! But if you stick with it, I promise you that you will become a better programmer.

Lastly, a note: in CSCI 134 and CSCI 136 you were taught how to write programs using data structures. This course expects that you know how to do those things. If you do not feel prepared, that's ok, but please make a point in coming to speak with me early in the semester. This course also expects that you know what files are and how to manage them on the command line. Many students just feel a little rusty on these ideas, and if that's the case, there's no need to see me. Just start with the short tutorial in Appendix A: Refresher on Files and the Shell.

What is a Programming Language?

Key ideas:
- Mathematical view of languages.
- Primitives and combining forms.

A *programming language* is a scheme for communicating with a computer. It is a *language* because it typically requires you to use words in a specific order. Because we care about achieving specific effects on a computer (Fig. 2), the *meaning* of those words in their particular order tends be to strongly related to changes in the state of the underlying computer. We call the set of words and their order the *syntax* of a language. The meanings of the words in a programming language are called its *semantics*. A "sentence" in a programming language is a *program*. The study of programming languages generally aims to understand the relationship between the kinds of programs one can write in a given language and whether the manner in which we ask programmers to write them helps or hinders from achieving one's programming goals. We will study both the syntax and the semantics of programming languages in this class.

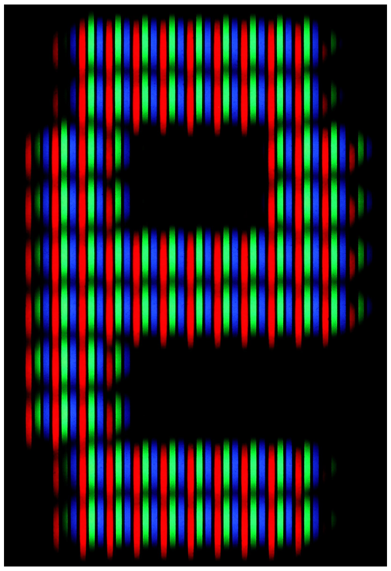
Figure 2: For example, printing a character to the screen.

Depending on your programming experience, you may have only used one or two programming languages, like Python or Java. In fact, there are thousands of programming languages. Although Python and Java *look* different, the meaning of Python and Java programs are similar. In other words, although Python and Java have different syntax, their semantics are closely related. This is why the CS department is confident that you can learn to program a computer in Python and then switch to learning data structures in Java. It really is just a matter of learning how to write what you already know in a different style. In this class, we will examine languages with different syntax *and* different semantics than you have seen before.

Importantly, a programming language defines what ideas you are capable of "discussing" with a computer. One of the most surprising facts I remember learning when I was new to computer science is that, natively, computers can only "discuss" numbers. They have no concept of characters, no concept of strings, or of images, or audio, or video, or any of the other kinds of data we might want to use in a program. Only through abstraction are we able to use those ideas in programs. For example, we can print a letter to the screen by telling the computer to interpret certain numbers as certain characters. [3] The fact that a computer, which can only technically operate over the domain of numbers, can universally compute using these other kinds of data is profound. Still, programming a computer to handle images or audio or video without being able to "talk" about images or audio or video is a real burden on the programmer. Programming languages are where the magic of abstraction happens.

[3] One simple scheme for encoding characters as numbers is called `ASCII`; you probably encountered `ASCII` in CSCI 136. The modern, de-facto encoding is called `utf-8`.

Primitives and Combining Forms

Although there are many programming languages, and they can differ wildly in syntax and semantics, that does not mean that they do not still have some similarities. In fact, most programming languages are organized and constructed using standard techniques. We will discuss these standard techniques at length. If this all sounds very vague to you, a good analogy is that of architecture. For example, two buildings may have very different appearances and functions. A typical suburban house and an aircraft hangar don't look at all alike and they serve different purposes (Figure 3). Nevertheless, an architect will point out that they both feature foundations, walls, and roofs. In the same way, differing programming language designs can be constructed from the same components.

Figure 3: A house and an aircraft hangar. Different, but also sort of the same.

The simplest kind of component is called a *primitive*. What, precisely, constitutes a primitive depends on the needs of the language designer. But a common way to decide whether some idea in a programming language should be primitive is to ask whether the idea is itself made of parts. For example, in Java, an `int` does not have parts. It is, in some sense, indivisible. Although you may know that an `int` is stored on a computer in binary, and a binary number really does have parts, Java hides those facts from you. In Java an `int` is primitive. Primitives are also usually used as data, although this is not always the case.

By contrast, a `class` in Java has parts. Classes have fields (variables) and methods (functions). A class is not a primitive. Instead, a `class` is an example of a *combining form*, another kind of language component. Combining forms combine ideas in a language into new ideas. One interesting thing about Java's classes is that you can use them to combine not just primitives, but also other classes. For example,

```
class Point {
  int x;
  int y;
}
```

is a simple Java `class` that defines a `Point` using primitive values. But we can also use classes to combine classes, as in the following definition.

```
class Line {
  Point start;
  Point end;
}
```

The above examples combine data for organizational purposes, but we can also combine for the purpose of calculation. For example, the `+` operator in Java combines two pieces of data and calculates their sum.

```
1 + 2;
```

Primitives and combining forms are the two most basic forms of languages components. Most other language features are usually a kind of primitive or a kind of combining form. Go back to a program you've written before and have a look at it. Which parts are likely primitives and which are likely combining forms? While you're at it, here's something to ponder: is a function definition in a programming language a primitive, a combining form, or something else? Pick your favorite programming language and see if you can figure it out.

An Introduction to F#

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

F# will force you think about programming in new ways. However, even if you never program in F# again, the experience will likely influence your programming for the better. After I discovered F#, I wondered why conventional languages like Java and C++ had to be so complicated. The short answer is: they don't have to be!

The readings in this course packet are intended to be read with your programming environment running so that you can try things yourself. You can either use one of the lab machines, which have F# preinstalled, or you can install F# on your own computer.[4] Many of the course's weekly quizzes assume that you are actively typing in and running code from the book, so if you want to best prepare for them, do that!

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

Key ideas:

- Foundational concepts in functional programming.
- Using `dotnet` to create a project.
- Compiling and running a project.

Let's look at our favorite starter program written in F#.

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

That is the entire program.[5]

[5] When typing programs into the REPL, you must terminate them with `;;`. For example, type `printfn "Hello world!\n";;` When coding outside of the REPL, you do not need to use `;;`. This may seem confusing at the moment, but you'll get the hang of it.

What is F#?

F# is a *functional* programming language. A functional programming language differs in form from programming languages like C[6] or Java[7]. Even if you decide that functional programming is not for you, exposure to functional programming ideas will change the way you think about

[6] C is an *imperative* programming language, meaning that programmers are expected to spell out every step of a program, including how values are stored in memory.

[7] Java is an *object-oriented* programming language. Object-oriented languages frame memory management and common software designs in terms of "objects," and are also usually imperative. Python and Java are imperative and object-oriented.

coding for the better.

Functional programming encourages *expressions* over *statements*, *immutable* instead of *mutable* variables, and *pure functions* instead of *side-effecting* procedures. Functions are *first-class*. F# is also *strongly typed* unlike C, which is *weakly typed*, and Python, which is *dynamically typed*. A functional program reads more like mathematical statements than a sequence of steps. You may not have heard some of these terms before, so let's take the time to understand each one.

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 assigned once, when it is defined.

```
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. In other words, you *cannot* update a variable!

Variables in F# are *immutable*, meaning that once they are defined, their values will never change. This may seem like a strange "feature," since mutable variables are undoubtedly useful. Of the many features of functional programming, this is considered by many to be one of the most important. Immutable variables force programmers to think of the result of every computation as a new piece of information, with a new name, and this constraint often has a clarifying effect on program logic. If this does not seem like a good idea yet, give it some time; its value will become apparent with practice.

Expressions

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

```
print("hi")   # 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[8] 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

[8] 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 you should expect that will take some time before you are adept 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.

F# has different syntax for function definitions, but the general idea is the same as in a language like Java. This function has a name, `increment`. It has a parameter called `n` of type `int`. It returns a value of type `int`. It computes and returns `n + 1`. Observe that it has no `return` statement. Because everything must be an expression, a function *must* return a value, so F# returns whatever the expression in the last line of the function returns.

Function definitions in F# are also *first class values*. Among other things, any first class value can be assigned to a variable. That lets us do something that might surprise you: assign a function definition to a variable.[9] For instance,

[9] 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.

```
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[10].

[10] 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 their meanings 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 code written in 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[11].

[11] `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 which performs the moral equivalent of pasting code from included libraries into a single file. 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. When a duplicate name appears, .NET signals that something is wrong by issuing a compilation error.[12]

[12] Compilation errors are a good thing. When they occur, the compiler is telling you that you definitely made a logic error. Learn to be friends with compiler errors.

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 a New F# Project

Let's conclude this chapter by writing a "helloworld" program. It is conventional in F# programming (and in the wider world of .NET programming) to package your code as an *application project*, and that's what we're going to do for all assignments in this class.

An application project is a development convention that collects all of the files for one program into a single folder with some metadata.

Whenever you want to write a new program to solve a new problem, you should generate an application project. An application project makes a program self-contained and simplifies the development process. Remember this procedure (or bookmark this page), because you are going to have to repeat these steps many times this semester.

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 called `Program.fs`.

```
// For more information see https://aka.ms/fsharp-console-apps
printfn "Hello from F#"
```

There is very little boilerplate text in the above program. Unfortunately, newcomers are often confused by F#'s conciseness. Therefore, throughout the course, we are going to always ensure that our F# programs contain at least a little boilerplate: a `main` function, marked as the `[<EntryPoint>]`. Replace the text in `Program.fs` with the following:

```
open System

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

F# is a *whitespace sensitive* language, like Python. Whitespace sensitivity means that the scope of a function definition is determined by indentation rules instead of explicit delimeters like curly braces. When using the REPL, we always have to use the delimiter `;;` to let the program know that we are done typing. Outside of the REPL, we do not need to use them because F# can infer the end of a line or program using indentation. Therefore, as in Python, you must be careful to always appropriately indent your code or the language will misunderstand you.

The last line in the above `main` function is the expression `0`. Because the last line of a function definition denotes the function's return value, When a program's `main` function returns `0` it informs the operating system that everything went OK. A non-zero return value indicates a failure.

this function returns zero.[13]

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 inform me when 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 in this class. 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[14] adds additional features like syntax highlighting and tooltips to your editor. To install Ionide, follow this tutorial on installing extensions[15].

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

[15] `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:[16]

16 On your personal machine, use ~/.emacs instead.

```
(require 'package)
(add-to-list
  'package-archives
  '("melpa" . "http://melpa.org/packages/"))
(unless package-archive-contents (package-refresh-contents))
(package-initialize)

(unless (package-installed-p 'fsharp-mode)
  (package-install '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.

More F#

This reading goes deeper into the F# language. As before, you are strongly encouraged to follow along on your computer.

Key ideas:

- Entry points.
- Defining and using variables and functions with or without type annotations.
- Currying and partial application.
- Modules.
- Conditionals.
- Lambda expressions.

Let's look at a simple F# program in *source code* form.[17]

```
[<EntryPoint>]
let main argv =
  printfn "Hello, %s!" argv[0]
  0
```

[17] The term *source code* refers to the code written by you, the programmer. In a compiled language like F#, source code is translated into *machine code* by the compiler. A computer can only run a program in machine code form.

Hopefully it's not a stretch to figure out what this program does. Nevertheless, it contains some things that are likely unfamiliar, so let's look at it line-by-line to understand what its parts are.

Understanding the Code You Write

As a new CS student, you've probably used code that you don't understand. Doing so is a bad habit. Whenever you borrow code from somebody, you really should make the effort to understand it. Let's *understand* the program above.

Entry Points

The first line,

```
[<EntryPoint>]
```

marks the function as the entry point to the program. The entry point is the location in the program where computation begins. In Java, the `main` function is always the entry point. F# gives you a bit more flexibility: it can be any single function that takes a `string[]` and returns an `int`, as long as that one function is labeled with the `[<EntryPoint>]` annotation.

Function Definitions

The next line,

```
let main argv =
```

denotes the start of a *function definition*. F# is whitespace-sensitive, like Python, and unlike Java or C. In F# code must be indented using spaces.[18] The body of the function definition begins at the = character and extends until the end of the indented region below.

[18] Not tabs. F# will reject programs that use tabs for indentation.

In F#, definitions of all kinds start with the keyword `let`. In general, a `let` definition *binds*[19] the result of the expression on the right of a = to the name on the left of the =.

[19] The term "bind" is intended to convey the idea that a definition "ties" a name and a value together.

Observe that variables and functions are defined in much the same way. The expression

```
let main argv = ...
```

defines a function called `main` with a single argument called `argv`, bound to the expression on the right, and

```
let x = 1
```

defines a variable called `x` bound to the value `1`. F# knows that the first example is a function, not a variable, because the name on the left of the = sign includes an argument (i.e., `argv`).

Note that, unlike most languages you've likely studied, F# functions are *pure*, meaning that they must *always return a value*. While side-effecting functions are possible in F# they are strongly discouraged, and you must take extra steps to use them.[20] In this class, we will write pure functions unless otherwise specified.

[20] Variables changed via side effect must be marked `mutable`. A side-effecting function is more properly referred to as a *procedure*.

Static Types, Type Inference, and Type Errors

F# is a *statically typed* programming language. In such languages, every datum in a program must belong to an unambiguous data type[21], and any operation involving that datum must be valid for its data type. Ensuring that a program adheres to these rules is known as *typechecking*. [22] Statically typed languages will refuse to compile programs that fail a typecheck. In other words, type errors are detected at "compile time." Without static typechecking, such errors would remain undetected until "runtime."

[21] `int`, `char`, and `string` are common examples of data types.

[22] For instance, it typically does not make sense to use the multiplication operation on two `string`s.

More generally, programming languages can be placed on a spectrum based on the strength of their typechecking. At one end are languages with static types like F# and Java. In the middle are *dynamically typed* languages like Python, JavaScript, and Ruby, that defer typechecking to runtime. Unlike static languages, dynamic languages can contain type errors that cause program failures when they are executed. At the opposite end of the spectrum are *untyped* languages like machine language. In untyped languages, type checks are not performed at all. Mixing data and operations inappropriately can lead to crashes or silent data corruption.

In many programming languages, we supply the information needed to carry out a typecheck using a *type annotation*. A type annotation is a label that states a data type. For example, the following Java code labels the variable `x` with the type annotation `String`.

```
String x = "hello";
```

Java requires that all data and functions have type annotations. Although F# is statically typed like Java, it does not always require type annotations. Have another look at our F# program.

```
[<EntryPoint>]
let main argv =
  printfn "Hello, %s!" argv[0]
  0
```

There are no type annotations in this program. In F#, type annotations are optional. F# can usually deduce the type of an expression without your help, a feature called *type inference*.

Since you're new to F#, you may look at the above program and think "how am I supposed to know what the right types should be?" Admittedly, it's not obvious that the type for `argv` is `string[]`. A downside of type inference is that although type information can be deduced from program text, the type is not always evident. The upside is that, when you get the type of an expression wrong, the F# compiler will tell you.

You may recall from programming in Java that type errors are a little scary the first time you see them. A type error is evidence that your program is *provably wrong*. The right attitude to take is that your compiler is doing you an important service. Such compiler messages are only painful when you write a lot of code between typechecks. Develop the habit of checking your program for type errors early and often. You'll find that your typechecker is your friend.

Let's work through a variation of our `main` program to practice reading and understanding a type error. Suppose for a moment that my `main` function was:

```
let main argv =
    argv + 1
```

Then F# would report,

```
error FS0001: The type 'int' does not match the type 'string []'
```

and I would not be able to run the program. Do you see why I got this type error?

Recall that an `EntryPoint` *must* be a function that takes a `string[]` and returns an `int`. That means that `argv` must be a `string[]`. Because our program adds something to `argv`, the `+` operation must take a `string[]` for its left operand. The literal value `1` has type `int`, so the `+` operation must take an `int` for its right operand. The result of the expression must also be an `int` because return type of `main` is an `int`. Therefore, to satisfy the typecheck, F# must be able to find an operation called `+` that takes a `string[]` and a `int` and returns an `int`. Unfortunately, there is no such operation, so F# reports a type error.

You can always add type annotations yourself. If you are at all unsure what the types of various things are, I encourage you to write them. I don't always add type annotations to my code, but when I want others to read my code, I make a point of adding them.[23] Let's add types to the `main` function in our running example.

[23] Our department's `ColloquiumBot` program is written in F# and I have gone to great lengths to provide type annotations everywhere.

```
let main(argv: string[]) : int =
    printfn "Hello, %s!" argv[0]
    0
```

The syntax of a typed function in F# is the following:

```
let <function name> (<arg_1>: <type_1>) ... (<arg_n>: <type_n>) : <return type> =
        <expression>
```

I leave the decision to include or leave out type annotations up to you.

As long as your program compiles and runs, I do not care. I encourage you to try out the type-free and parens-free syntax, because shorter programs are often easier to read. However, to a large extent, this is a matter of personal taste.

Function Body

The meat of our `main` function consists of the following two lines:

```
    printfn "Hello, %s!" argv[0]
    0
```

Notice that this code is indented from `main`. The indentation is how we know that the code is a part of the `main` function definition. My personal convention is to use 4 spaces. Others use 2. Again, choose what you like, but note that the F# compiler *will not* let you use tabs.

Function Calls

The first line of `main` *calls* the `printfn` function. Function calls in F# work *exactly* the same way as "application" in the lambda calculus, which we will discuss in detail in this class. The important thing to know for now is that an expression of the form

```
a b
```

means that we are *calling* the function `a` with the argument `b`. As long as `a` and `b` are defined somewhere, it is valid F# code. Here's another example with `a` and `b` defined.

```
let b = 1
let a x = x + 1
a b
```

This program returns the value `2`. Try typing it into the `dotnet fsi` REPL to see for yourself.[24]

[24] Don't forget to add `;;` delimiters after each line.

Function Types

A function in a functional programming language is data. In a statically typed programming language, all data must be typed, so that begs the question: what is the type of a function? For example, what is the type of the following function?

```
let a x = x + 1
```

Try typing the above expression into `dotnet fsi`. You should see something like:

```
int -> int
```

The `->` notation tells us that a value is a function. That value is bound to the variable called `a`. Remember when I said that all definitions are defined using `let`? Because the above `let` expression has additional terms on the left hand side of the = sign, F# knows that you are defining a function. In fact, our `let` expression is a shorthand for the following,

```
let a = fun x -> x + 1
```

which may be a little clearer. We are defining a variable called `a` that is bound to a function that *takes* an `int` and *returns* an `int`. Just keep in mind: you can tell that a value is a function whenever you see a `->` in its type.

By the way, when we put the above function `a` into `dotnet fsi`, it actually prints out the following type:

```
> let a x = x + 1;;
val a : x:int -> int
```

Try not to be confused by the extra output. F# is trying to be helpful by including names along with the types. The value (`val`) of the entire expression is a name binding called `a`. Since the entire expression has a `->` in it, we know it's a function. The stuff on the left side of `->` is the type of the function's argument. Therefore, the type of the function's argument, `x`, is `int`. The stuff on the right side of `->` is the type of the function's return value.

Polymorphic Functions

Polymorphic code is code that works for different types of data. You've seen polymorphism before. Java generics are a kind of polymorphism.[25] For example, we know that linked lists work equally well for integers and strings, so Java lets us write:

[25] Strictly speaking, Java generics are a Java-specific implementation of *parametric polymorphism*.

```
List<Integer> x = new List<>();
```

for an integer, or

```
List<String> y = new List<>();
```

for a string. Nevertheless, we only need to implement a single `List` implementation because our `List` can be made generic.

In F#, polymorphic types are prefaced by the single quote character, `'`. A function having polymorphic type can take *any* kind of data. For example, let's look at the *identity function*. The identity function just returns whatever it is given. Since a procedure that "returns whatever it is given" does not need to do anything different for values of different types, we ought to be able to write a single implementation that works for any type.

```
let id x = x
```

If we type this into `dotnet fsi`, F# tells us that the type is:

```
'a -> 'a
```

Let's try using `id` for values with different types. It works for numbers:

```
> id 5;;
val it : int = 5
```

It also works for strings:

```
> id "hi";;
val it : string = "hi"
```

Curried Definitions and Partial Application

I'm about to introduce something weird. Have a look at our hello world program again.

```
let main argv =
    printfn "Hello, %s!" argv[0]
    0
```

If we rewrite `main` with types and parens around function calls, we get

```
let main(argv: string[]) : int =
    printfn("Hello, %s!")(argv[0])
    0
```

and this is exactly the same program. But have a look at our call to the `printfn` function. Doesn't that look a little strange? Why does the call to `printfn` have *two* parenthesized arguments?

The short answer is that, in F#, function calls are *curried*. Let's spend a little time understanding what that means and why we have it.

F# is strongly patterned on a model of computation called the *lambda calculus*. We will discuss the lambda calculus in detail later in this class. For now, the important thing to know is that the lambda calculus has no notion of functions that take multiple arguments. It doesn't have them because they are not necessary: any function that takes multiple arguments can be constructed out of multiple, single-argument functions.

Here is a function in F# that we informally think of as being defined over two arguments.

```
let f x y = x y
```

The type for `let f x y = x y` says something a little more subtle though.

```
('a -> 'b) -> 'a -> 'b
```

Function types are a little hard to read at first, but once you understand the rules, they're informative. Let's look at this type one part at a time.

We know that the function `f` takes two arguments and returns one value, so we expect our type to say something like that. Indeed it does. `('a -> 'b) -> 'a -> 'b` has three parts:

- `'a -> 'b`, which is "one part" because of the parens,
- `'a`, and
- `'b`.

According to the above type, our first variable `x` must be `'a -> 'b`, a function from any type `'a` to any type `'b`. The type `'a -> 'b` is very general and admits any single-argument function. For example, the function `let foo (i: int) = str i`, which converts an integer `i` into a string, will satisfy the type for `x` because `'a` could be `int` and `'b` could be `string`.

Notice that F# *did not* say that the type of `x` was `'a -> 'a`. That type would be more restrictive than `'a -> 'b`. The function `let foo (i: int) = str i` would not satisfy the type `'a -> 'a`, because the type says that the input and the output of the function have to have the same type. When we do not explicitly tell F# the type of an expression, it will automatically deduce the type from the context, and the language always tries to compute the least restrictive type. Since nothing about the *use* of `x` implies that the types of the input and the output of `x` *must* be the same type, F# uses introduces two different polymorphic types, `'a` and `'b`.

In an important sense, we can say that `f` really only has *two* parts:

```
stuff -> other stuff
```

where `stuff = 'a -> 'b` and `other stuff = 'a -> 'b`.[26]

[26] If you are wondering why I split the expression up this way, the short answer is: trust me for now. The longer answer is that function application in the lambda calculus is left-associative. We will get there in time.

Since `f` is a function, we should be able to give it some `stuff` and get `other stuff` back. We know that the type of the input stuff, `x`, is `'a -> 'b`. We also know that `x` must itself be a function, because of its type.

Let's try using `f`. Since its first argument is a function, let's define a function we'll name `g`. We'll call `f` with `g` and name the output `h`.

```
let f x y = x y
let g x = x + 1
let h = f g
```

Is the function `g` acceptable to F#? Let's work through this "on paper." The type of `f`'s argument `x` is `'a -> 'b`. What is the type of `let g x = x + 1`? Try it in the REPL before continuing.

It's `int -> int`. Does this satisfy `('a -> 'b)`? Yes, because when we call `f` with `g`, the language determines that `'a = int` and `'b = int`.

Now what is the type of `h`? Again, we should be able to figure it out "on paper." `h` is what we get when we call the function `f` with the function `g`. According to our type for `f`, `('a -> 'b) -> 'a -> 'b`, we should get an `'a -> 'b` back, right? But we also know more! When we gave the function `f` the argument `g`, we constrained the types so that `'a = int` and `'b = int`. Therefore, `h` must have the type `int -> int`.

So `h` is a function. Try typing the definition for `h` into `dotnet fsi` to verify our reasoning. This already feels like a different way of programming than in a language like C or Python, doesn't it? As a matter of fact, you can also reason through C and Python on paper, but with varying degrees of difficulty. You can often figure out what a functional program should do with a little pencil and paper work. Predictability is an important aspect of programming, and making program outputs easy to predict is an important design goal for functional programming languages (see Figure 4).

Figure 4: To put this in perspective, suppose you hire an engineer to build you a bridge. Understandably, you want to know that your money is well-spent, so you ask them "is this bridge safe?" Imagine how you would feel if the engineer responded "I don't know. Let's try it!" *Just trying it* is the norm in programming, but it should not be.

Let's keep going! If `h` is a function, then we can call it with an argument, right?

```
> h 3;;
val it : int = 4
```

Does this result make sense to you? If you have trouble working through the program's logic on paper, this is a good topic to discuss in class or in office hours. Please ask!

What we've learned is that, in F#, when a function over multiple arguments is given fewer arguments than it takes, it returns a function that expects the remaining arguments. To be a little more precise, *arity* is the count of the number of arguments a function takes. Any function with an arity greater than 1 can be defined by composing multiple single-argument functions, a technique called *currying*.[27] Calling a multi-argument function with fewer arguments than its arity is called *partial application*.

You may find it hard to imagine why currying and partial application is useful. In the near future, we will discuss a style of functional programming called *combinator-style programming*, and you will see that these features can help you write more concise, more readable code than the alternatives.

[27] The term *currying* is not named for a dish with a spicy sauce, but rather for the 20th century mathematician, Haskell Curry. Curry founded the field of combinatory logic, which serves as a theoretical basis for the design of some functional programming languages.

Putting It All Together

Returning to our hello world program, let's examine the type of the `printfn` function. You can get F# to tell you its type by entering the name of the function into `dotnet fsi`:

```
> printfn;;
val it: TextWriterFormat<'a> -> 'a
```

Recall that we called `printfn` with *two* arguments, a strange fact that motivated this entire discussion. So what is a `TextWriterFormat<'a> -> 'a`? And why doesn't the REPL show a type for a curried function that takes two arguments? The short answer is that it does. A `TextWriterFormat<'a> -> 'a` is hiding a lot in that `'a` parameter. A longer answer is the number of arguments `printfn` expects depends on what you give it as a format string. We can see this if we play around with `printfn` a little. For example, if we give it the format string `"Hello, %s!"`,

```
> printfn "Hello, \%s!";;
val it: (string -> unit) = <fun:it>
```

it expects one more string `string` before it returns something [28]. If we give `printfn` the format string `"There are %d %ss"`,

[28] We'll discuss `unit` in a moment.

```
> printfn "There are \%d \%ss";;
val it: (int -> string -> unit) = <fun:it>
```

Then we need to give it an `int` and a `string`, like so.

```
>  printfn "There are \%d \%ss" 2 "Dan";;
There are 2 Dans
val it: unit = ()
```

Return Value

Have one last look at our program.

```
let main(argv: string[]) : int =
    printfn("Hello, %s!")(argv[0])
    0
```

The last line in our `main` function is `0`. In F#, the last expression in a function definition is the return value. If you recall, returning `0` tells the operating system that "everything ran OK." Any other value signals an error.

Adding a New File to a Project

By default, your project contains only a single file called `Program.fs`. Unlike Java, F# does not care what you name your code files as long as you tell it where to find the code in the MSBuild file.

I like to organize my code according to "responsibilities." For example, maybe I have a program that reads input, does some processing, builds a data structure, computes some values, and then prints out the result. In this case, I might have a file called `io.fs` for input and output processing, `utils.fs` to handle data manipulation (like converting data from arrays into hash tables), and `algorithms.fs` for the core computation. I personally like to keep very little in the `Program.fs` file, which mostly just contains the `main` function. Unless I ask you to organize your code in a specific manner,[29] use whatever system of organization makes sense to you.

[29] Be on the lookout for specific instructions in lab handouts.

To add a new file to your project, you need to do two things. Suppose we create a new file called `io.fs` and we want to call its code from the `Program.fs` file. Look for a `.fsproj` file in your project directory. This is your project specification. Open it up with your favorite code editor. You should see something like

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="Program.fs" />
  </ItemGroup>

</Project>
```

We need to add a `Compile` tag just above the `Compile` tag for `Program.fs` so that MSBuild will compile `io.fs` first. Here's what my `.fsproj` file looks like after I make the change:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="io.fs" />
    <Compile Include="Program.fs" />
  </ItemGroup>

</Project>
```

With this change, `Program.fs` is now permitted to refer to code stored in `io.fs`. Note that if you see the following error when you put your code in a separate file,

```
error FS0222: Files in libraries or multiple-file applications must
              begin with a namespace or module declaration
```

, you must place your code within a *module*. A module is simply a unit of code organization. To do so, put a module declaration at the top of your file to make this error message go away. For example, in `io.fs`, I might put:

```
module IO
```

and then in my `Program.fs`, I write

```
open IO
```

A Quick Tour of Some Other Important Ideas

Here, we explore a small number of additional foundational concepts you will need in order to program in F#.

Primitive Types

F# has a rich set of *primitive data types*. These types represent fundamental categories of data. As in Java, primitive types are always written in lowercase in F#. The primitive types are `bool`, `byte`, `sbyte`, `int16`, `uint16`, `int`, `uint`, `int64`, `uint64`, `nativeint`, `unativeint`, `decimal`, `double`, `single`, `char`, `string`, and `unit`.

Documentation for the above types may be found online[30].

F# also allows one to create user-defined types. Note: by convention, user-defined types are written in `UpperCamelCase`[31] in F#.

[30] `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/basic-types`

[31] `https://en.wikipedia.org/wiki/Camel_case`

Expressions

Everything in F# is an expression. Using the `dotnet fsi` read-eval-print-loop (REPL) program,

```
> 1;;
val it : int = 1
```

we can immediately see that anything we type into the REPL *returns a value*.

There are *no statements* in F#, although there are functions that look similar. Remember `printfn` from above? You may recall that when we called it like

```
printfn "Hello, %s!" "Dan"
```

it returned `unit`. What is `unit`? `unit` signals that a function only produces a side effect. In other words, it does not return anything. Nevertheless, everything in F# is an expression, so something must be returned. To fit into this scheme, the special value `()` is returned, which means "nothing" and has type `unit`. Let's see for ourselves in `dotnet fsi`.

```
> printfn "Hello, %s!" "Dan";;
Hello, Dan!
val it : unit = ()

> ();;
val it: unit = ()
```

Conditionals

`if/else` expressions look much like their counterparts in Python:

```
if x > 0 then
  1
else
  2
```

Since conditionals are *expressions* in F#, you can also use them to conditionally assign values.[32]

[32] If you know some C, this is like the ternary operator, `a ? b : c`.

```
let y = if x > 0 then
          1
        else
          2
```

Indentation is important for conditionals. Note that the body of the true and false clauses must be indented past the start of the `if` expression.

Lambda Expressions

A *lambda expression* is a function definition without a name. Recall our earlier discussion, where I claimed that the `let` expression,

```
let a x = x + 1
```

was a shorthand for

```
let a = fun x -> x + 1
```

I glossed over the meaning of `fun x -> x + 1`, but hopefully you can see that all we're doing is defining a function that takes an `int` and returns an `int`. An foundational idea in functional programming is that *function definition* and *naming* are different things, and that mixing them together adds unnecessary complexity (and confusion). The keyword `let` is for naming, and the keyword `fun` is for making functions. Because we make functions frequently while programming, the shorter version exists to save on typing. But we can see that they have the same type in the REPL.

```
> let a x = x + 1;;
val a: x: int -> int

> let a = fun x -> x + 1;;
val a: x: int -> int
```

The `fun` keyword makes functions definitions "anonymously." In other words, a lambda expression is a function definition with no name.

We are not required to name functions if we don't want to. Remember our friend, the identity function?

```
let id = fun x -> x
```

We can call the anonymous form of this function with a literal, it computes something, but nothing was ever named using `let` at any point in the program.

```
> (fun x -> x) 1;;
val it: int = 1
```

Lambda expressions are very useful in F#, and we use them widely in functional programming. We'll explore these uses in a future chapter, *Higher-Order Functions*.

Pragmatism Over Purity

The ML family of languages favors pragmatism over mathematical purity. Therefore, it allows a programmer great flexibility to wiggle out of tough situations using mutable variables, side effects, imperative code, and casts. In this class, use of mutability, side effects, imperative code, and casts will be penalized, because it's hard to learn *functional* programming if you can lean on those features. After this class is over, feel free to use those other features. I myself use them in some circumstances, particularly when it is important that my code be fast. By the end of this semester, you will have a better appreciation for the downsides of these "escape hatches," and then you can use them with a fuller awareness of their tradeoffs.