A Slightly Longer Introduction to F#

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

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

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

Type this program into an editor and save it with the name `helloworld.fs`. I recommend typing the program instead of copying-and-pasting it because retyping it will force you to notice important details about the program.

Hopefully it's not too much of a stretch to figure out what this program does! We will look at this program line-by-line to understand what its parts are, but first, let's understand how to run this program.

The F# Compiler

As with any other programming language, your computer cannot understand an F# program in source code form. It must be translated into another form. Unlike a language like C, however, we do not translate F# directly into an executable binary. Instead, the F# compiler, called `fsharpc`, converts F# source code into an *architecture independent* form; architecture independence means that the resulting compiled program can be *run on any computer*: a personal computer, a cellphone, a supercomputer, a watch, or even an embedded computer (like the kind in "smart lightbulbs").

How is this independence achieved? By using a *virtual machine*[28]. A virtual machine provides a simple abstraction that hides many of the quirks present in specific hardware. A virtual machine looks like a new kind of simple hardware, with its own instruction set and simplified semantics. Virtual machines even allow languages to be *type safe* at the virtual hardware layer, which means that many of the security vulnerabilities commonly exploited in languages like C simply are not possible.

[28] Specifically, a *process virtual machine*, which is a virtual machine specifically designed to host a language. Such VMs are simpler than *fully virtual* machines, which are intended to mimic an actual processor, along with all its quirks, in order to host an entire operating system.

The task of building a virtual machine for your platform is up to the language implementors, who have typically have a much better understanding of how to address code portability concerns than your typical programmer. This design is essentially the same taken by the Java programming language: `javac` produces Java byte code, which is then interpreted by the Java Virtual Machine (JVM). This allows you to write your code once and run it anywhere. "Write once, run anywhere" was even the slogan used by Sun Microsystems when they originally marketed the Java programming language in the mid 1990's[29]. *f* Here is a (snippet of the) translation of the above program into virtual machine *byte code*, which is the virtual machine equivalent of machine code. This byte code, called the Common Intermediate Language (CIL), is specifically for Microsoft's VM, the Common Language Runtime (CLR).

[29] It should be noted that neither Microsoft nor Sun Microsystems invented the idea of portable bytecode. That honor appears to go to Martin Richards, who came up with *O-code* to make it easier to port the BCPL language in the mid-1960's.

```
0000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00
0000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000030 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00
0000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68
0000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f
0000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20
0000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00
0000080 50 45 00 00 4c 01 03 00 da 06 ca 5b 00 00 00 00
0000090 00 00 00 00 e0 00 0e 01 0b 01 08 00 00 08 00 00
...
```

Although it looks similar from our perspective to x86 machine instructions, the format is quite different. Here's the same information using CIL's instruction mnemonics (i.e., "assembly"):

```
.class public abstract sealed auto ansi
  Program
    extends [mscorlib]System.Object
{
  .custom instance void [FSharp.Core]Microsoft.FSharp.Core.
      CompilationMappingAttribute::.ctor(valuetype [FSharp.Core]
      Microsoft.FSharp.Core.SourceConstructFlags)
    = (01 00 07 00 00 00 00 00 ) // ........
    // int32(7) // 0x00000007

  .method public static int32
    main(
      string[] argv
    ) cil managed
  {
    .entrypoint
    .custom instance void [FSharp.Core]Microsoft.FSharp.Core.
        EntryPointAttribute::.ctor()
      = (01 00 00 00 )
    .maxstack 4
    .locals init (
      [0] class [FSharp.Core]Microsoft.FSharp.Core.FSharpFunc`2<
          string, class [FSharp.Core]Microsoft.FSharp.Core.Unit>
           V_0,
      [1] string V_1
    )

    // [3 5 - 3 25]
    IL_0000: ldstr        "Hello, %s!"
    IL_0005: newobj       instance void class [FSharp.Core]
        Microsoft.FSharp.Core.PrintfFormat`5<class [FSharp.Core]
        Microsoft.FSharp.Core.FSharpFunc`2<string, class [FSharp
        .Core]Microsoft.FSharp.Core.Unit>, class [mscorlib]
        System.IO.TextWriter, class [FSharp.Core]Microsoft.
        FSharp.Core.Unit, class [FSharp.Core]Microsoft.FSharp.
        Core.Unit, string>::.ctor(string)
    IL_000a: call         !!0/*class [FSharp.Core]Microsoft.
        FSharp.Core.FSharpFunc`2<string, class [FSharp.Core]
        Microsoft.FSharp.Core.Unit>*/ [FSharp.Core]Microsoft.
        FSharp.Core.ExtraTopLevelOperators::PrintFormatLine<
        class [FSharp.Core]Microsoft.FSharp.Core.FSharpFunc`2<
        string, class [FSharp.Core]Microsoft.FSharp.Core.Unit>>(
        class [FSharp.Core]Microsoft.FSharp.Core.PrintfFormat
        `4<!!0/*class [FSharp.Core]Microsoft.FSharp.Core.
        FSharpFunc`2<string, class [FSharp.Core]Microsoft.FSharp
        .Core.Unit>*/, class [mscorlib]System.IO.TextWriter,
        class [FSharp.Core]Microsoft.FSharp.Core.Unit, class [
        FSharp.Core]Microsoft.FSharp.Core.Unit>)
    IL_000f: stloc.0      // V_0
    IL_0010: ldarg.0      // argv
    IL_0011: ldc.i4.0
    IL_0012: ldelem       [mscorlib]System.String
    IL_0017: stloc.1      // V_1
    IL_0018: ldloc.0      // V_0
    IL_0019: ldloc.1      // V_1
    IL_001a: callvirt     instance !1/*class [FSharp.Core]
        Microsoft.FSharp.Core.Unit*/ class [FSharp.Core]
        Microsoft.FSharp.Core.FSharpFunc`2<string, class [FSharp
        .Core]Microsoft.FSharp.Core.Unit>::Invoke(!0/*string*/)
    IL_001f: pop

    // [4 5 - 4 6]
    IL_0020: ldc.i4.0
    IL_0021: ret

  } // end of method Program::main
} // end of class Program
```

There is quite a lot of stuff baked into this bytecode. Ordinary assembly code is very primitive; it has no notion of data types, classes, methods, and so on. The CLR, on the other hand, *does* know about these things. For example see if you can find the text `.method public static int32 main` in the CIL program above. That section looks a lot like a Java method, doesn't it? Note that you do not need to know CIL in this class. I'm just showing this to you to give you some perspective.

History

F# was developed by a research group at Microsoft Research, led by Don Syme (Figure 17). Although F# has some novel features, particularly the ways in which it interoperates with other .NET code, its syntax and semantics are largely inspired by the ML family of programming languages. Syntactically, F# added whitespace-sensitivity (like Python) and "lightweight" refinements of older ML syntax that, in my opinion, makes it very pleasant to use. If you like Python, you'll probably like F#.

Figure 17: Don Syme.

ML was designed by researchers at the University of Edinburgh, most notably Robin Milner (Figure 18) and Luca Cardelli (Figure 19), in the early 1970's. ML stands for "meta language," because it was originally designed to be a meta language for writing "proof tactics" (you can think of these as search procedures) for the LCF automated theorem prover. Although ML was heavily inspired by mathematical logic and early functional programming languages like LISP[30], its authors made a concerted effort early on to create something "elegant." But what makes ML especially interesting is that the language design was not static. Milner was inspired by other programming language research happening concurrently at Edinburgh, notably the HOPE programming language. ML borrowed many ideas from these other languages whenever a feature made the language feel simpler or more elegant to its authors. For example, pattern matching, which is a feature widely enjoyed by ML users originally came from HOPE.

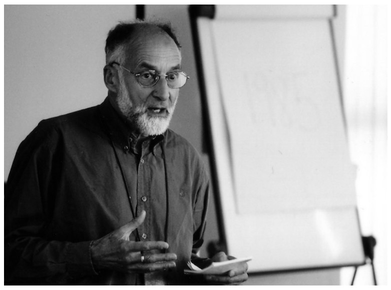
Figure 18: Robin Milner.

Figure 19: Luca Cardelli.

[30] The first version of ML was written in LISP!

I enjoy reading ML's early design documents, because it is clear that the most important thing was to build a "simple and well-understood" language. ML was also one of the first languages to have a complete formal specification. Nonetheless, ML has a strong pragmatic streak that makes it—in my opinion—a lot more fun to program in than other programming languages.

ML quickly outgrew its origins in the LCF project and was used widely among academics starting in the late 1970's. "Standard ML" (SML) arose in the 1980's out of a desire to allow for many implementations of ML. One of the most popular versions of SML is the "Standard ML of New Jersey" (SML/NJ) implementation that was jointly developed by Bell Labs and Princeton University, both of which are in New Jersey. Our lab machines have `smlnj` interpreters installed on them, so if you're curious about the differences between F# and SML, have a look.

Compiling using `dotnet`

If you've programmed using Microsoft Windows before, you may have used the Visual Studio IDE[31]. Visual Studio is the *de facto* code editor for commercial F# programmers. You are welcome to use Visual Studio if you have it, but as it is quite expensive, I do not require it for this class. Instead, we will be using the cross-platform `dotnet` tool, which runs the F# compiler for us.

[31] Confusingly, "Visual Studio" is a completely different product than "Visual Studio Code."

MSBuild

Manually managing the F# compiler can get a little annoying in the same way that managing `javac` or `gcc` or `gcc` can be annoying. Therefore, this class asks you to produce code as a part of an *MSBuild project*. MSBuild is Microsoft's (much more sophisticated) equivalent to C's Makefiles. You should have first encountered `dotnet` in A Brief Introduction to F#.

Because MSBuild projects are written in XML, we will mostly create and manage our projects using a command line tool called `dotnet`. This tool automatically generates and edits MSBuild files for you. However, since the `dotnet` tool is still in its infancy (it was first released in late 2016), we will occasionally need to modify MSBuild files by hand.

New projects

We create new projects using the `dotnet new` command. Typing this command without arguments will show you a set of project templates that you can use to create a new project. For this class, we will mostly use the command `dotnet new console -lang F#` which creates a new project for a command-line program using F# as the language.

Note that `dotnet new` creates a new project in the current directory. Be careful if you are putting your code in a location that already has code as the effect may not be what you intend.

The default console project is a helloworld program, which is quite convenient.

Building

To build a project, `cd` to the directory containing your project files and type

```
$ dotnet build
```

Running

To run a project, `cd` to the directory containing your project files and type

```
$ dotnet run
```

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 where you put your code. It could all go into a single source code file.

However, as your projects grow in size, you will find it beneficial to organize your code across multiple files. 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.

However, use whatever system of organization makes sense to you.

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>netcoreapp2.1</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>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

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

</Project>
```

Running `dotnet build` should now allow `Program.fs` to refer to code stored in `io.fs`. Note that if you're following along at home, you will likely see the following error when you try the above.

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

What's the deal? In short, in F#, you must place all library code inside either a *module* or a *namespace*. Both of these two things are a form of code organization. We'll stick to modules for this course, as namespaces are only slightly different and are not really necessary unless you are mixing F# and C# code (C# has no notion of modules, only namespaces).

Put a module declaration at the top of your module to make this error message go away. For example, in `io.fs`, put:

```
module IO
```

and in your `Program.fs`, write

```
open IO
```

Figure 20: "This means something. This is important." If you don't get the reference, your homework is to gather your friends together and watch *Close Encounters of the Third Kind*.

"This means something. This is important." Understanding code you wrote.

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 `helloworld.fs` program we typed in at the beginning of this reading, line by line.

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, 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*. Unlike C and Java, F# is whitespace-sensitive, like Python. In F# code must be indented using spaces (not tabs– F# is an *opinionated* language, meaning that code style is enforced in the language). The body of the function definition begins at the = character and extends until the end of the indented region below.

Note that, unlike most languages you've studied so far, F# functions are *true functions*, meaning that they must *always return a value*. While side-effecting functions are possible in F#, they are strongly discouraged, and you have do some extra things to use them (like using the `mutable` annotation). Thus you are encouraged to write *pure functions*. In this class, you should assume that we will be writing pure functions unless otherwise specified.

This function definition looks fairly sparse, doesn't it? In fact, despite the fact that F# is a *statically typed* programming language, there are no type annotations in the above. That's because F# can usually *infer* your type annotations without your help.

In F#, declarations of all kinds start with the keyword `let`. In general, you should assume that `let` simply means that we should bind the expression to the right of the = to the name on the left of the =.

If you've played with F# a bit you might be thinking, "wait, variables and functions are declared the same way in F#?" Indeed they are. So

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

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

```
let x = 1
```

declares a variable called `x` bound to the value `1`. The way that F# knows the first example is a function and not just a variable is because the part of the expression to the left of the = sign has an argument (i.e., `argv`).

So in this case, we are declaring a function `main` that has a single argument, `argv`. Since you're new to F#, you may be thinking "how am I supposed to know what type `argv` is?" This is admittedly one of the downsides of type inference– that information is not obvious in the program text. That said, unlike a dynamically typed language like Python, if you get the type of `argv` wrong, the F# compiler will tell you. Suppose for a moment that my `main` function was:

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

Then F# will report,

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

and I will not be able to run the program. Since the type check fails—`argv` is a `string[]`, not an `int`—compilation also fails. This is a *feature* of a statically-typed language, because it enables you to find bugs in your program *before you run it*.

You can also add type annotations yourself, and if you are at all unsure what the types of various things are, I encourage you to write them. Let's rewrite our `main` function with types.

```
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>
```

It's up to you how you want to write your programs. F# doesn't care, and neither do I. I encourage you to try out the parens-less syntax, however, as once you are accustomed to it, you will find F# programs very easy to read.

Function body

The meat of our `main` function is the following:

```
    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.

The first line *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. However, the important thing to know for now is that an expression of the form

```
a b
```

means that we should *call* the function `a` with the argument `b`. The above is actually valid F#. Here's an example that might make more sense to you:

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

which returns 2. Try it in `dotnet fsi` if you don't believe me.

Function types

Let's talk a little about function types. When you type an expression into `dotnet fsi`, it will print that expression's type. The `->` type notation tells us that something is a function. So, for instance,

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

has type

```
int -> int
```

because it is a *function* that *takes* an `int` and *returns* an `int`.

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 this. F# is trying to be helpful by including names along with the types. So the entire expression is called `a`. This makes sense, because we asked F# to name the entire expression `a` by using the `let` keyword. 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. The stuff on the right side of `->` is the type of the function's return value. So the type of this function's argument, `x`, is `int`. Finally, the return value has type `int`.

Polymorphic functions

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

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

for an integer, or

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

for a string, but we only have to. create one `List` implementation.

In F# polymorphic types are shown as *tick variables.* A function with a tick variable can take *any* kind of data. For example, let's look at the *identity function*. The identity function just returns whatever it is given. This should work equally well for any type, right?

```
let ident x = x
```

If we type this into `dotnet fsi`, F# will tell us that the type is:
`'a -> 'a`

Let's try using `ident` for values with different types.

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

It works for numbers. We got `5` back.

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

And it works for strings. We got `"hi"` back.

Curried functions definitions, function types, and function application

OK, I'm about to introduce something very weird. Try not to get upset. Have a look at the following program again.

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

We can rewrite `main` like:

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

and this is exactly the same program. "BUT WAIT," you say, "WHY DOES `printfn` HAVE TWO PARENS????"

That's because, in F# function calls are *curried*.

F# is strongly based on a model of computation called the *lambda cal-*

culus. We will discuss the lambda calculus in detail this class. For now, it's worth noting that the lambda calculus has no notation for functions that take multiple arguments. It doesn't have them because they are not necessary.

Here's a function in F# that we tend to think of as "taking two arguments."

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

However, the type for `let f x y = x y` is:

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

which may make your mind melt a little the first time you see something like it. These things are actually easy to read with a little practice. Let's break it into pieces.

According to the above type, `x`, our first variable, must be a function from any type `'a` to type `'b`. Since the types of `'a` and `'b` need not be the same type, F# uses two different letters (`a` and `b`).

The thing is, if we squint at the type above a little, it has the form:

```
stuff -> other stuff
```

So, in principle, we should be able to give it some stuff and get some other stuff back. We know that the type of that input stuff, `x`, is `('a -> 'b)`. `x` must be a function, because that's what it's type says it is. So let's do that. Let's call `f` with a function we'll call the output g:

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

What is the type of g? It's also a function. Try it in `dotnet fsi` so you can see what type it prints out.

We can keep going. If g is a function, we can call it with an argument, right?

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

So what we've learned is that functions of multiple arguments in F# really are functions that *return another function*. Composing a multi-argument function from single-argument functions is called *currying*, and calling them with arguments one at a time is called *partial application*.

You may find it hard to imagine why we would ever want partially applied functions. I did too, when I first learned F#! And, in fact, I went years without explicitly constructing any curried functions, which seemed to suggest that they were not necessary. Nevertheless, when I discovered their first "killer application", parsing, it changed the way that I thought about them. I now write much more concise, readable code than the code I wrote before. Part of the reason is that I curry my functions when it makes sense.

Let's look at the type of the `printfn` function. It is:

```
TextWriterFormat<'a> -> 'a
```

which is, perhaps, a little puzzling until you recall that F# is designed to interoperate with other .NET code. People using .NET mostly write C# code, and C# was strongly inspired by Java, especially its use of generics. This means that F# programs can have *both* polymorphic types like `'a` and generic types! For the most part, F# will handle the gory details of converting between these kinds of types for you, but you can see that the above type declaration uses both: `TextWriterFormat` is a C# generic class, but we can give it a polymorphic type `'a`.

Anyway, with the above type declaration, we can see that `printfn` takes a `TextWriterFormat<'a>` and returns an `'a`. Hang on... we called `printfn` with more than one argument, remember?

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

So should't `printfn` be a curried function? Actually, no— and the reason is that `TextWriterFormat<'a>` is already secretly a function. For example, when used in `printfn`, the string `%s` causes F# to infer that you need a function `string -> unit`, and so the type of `printfn` becomes:

```
(string -> unit) -> string -> unit
```

and you'd call it like:

```
printfn "%s" "heya"
```

If we use the format string `%s %d`, then `printfn` becomes:

```
(string -> int -> unit) -> string -> int -> unit
```

and you'd call it like:

```
printfn "%s %d" "hello" 1
```

This is how `printfn` is able to "magically" determine how many parameters to take depending on the given format string. Cool, huh? *Java cannot do this*, and in fact, Java's `String.format` method *cannot be stati-*

cally type checked. Instead, it checks dynamically and throws an exception when you mess up, which is an ugly hack in my opinion.

Return value

The last line in our `main` program 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.

A few more features

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. I strongly discourage you from using these features in this class. In fact, for this class, use of mutability, side effects, imperative code, and casts will be penalized. Why? Because I want you to learn *functional* programming. When you're all grown up and you leave Williams College, feel free to use those other features. I myself do this in some circumstances, particularly when it is important that my code be fast. By the end of this semester, you will have an appreciation for the tradeoffs that these little "escape hatches" entail, which are substantial.

Expressions

First, 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 everything we type in *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`? Well, the purpose of F# is essentially to produce a side effect. It does not return anything. But in F# everything is an expression, so something must be returned. In this case, since we don't expect anything back, 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 = ()
```

Lambda expressions

Lambda expressions are a feature of F# that allow us to create functions definitions "anonymously." In other words, a lambda expression is a function definition with no name.

The following is a lambda expression that computes the identity function:

```
fun x -> x
```

Try it in `dotnet fsi`. Here's another example.

```
let y = 1
(fun x -> x) y
```

which evaluates to `1`.

Lambda expressions are very useful in F#, and we use them widely, particularly in `map` and `fold` functions, which we will get to later.

Types

F# is a *statically typed* programming language, which means that every variable and datum in the language must have an associated type label, and that all operations on data must *type check*, meaning that those operations are logically consistent.

F# has a small set of *primitive data types*. These types represent fundamental categories of data representation in the underlying CLR virtual machine. The complete list may be found online[32]. Primitive types are written in lowercase in F#.

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

F# also allows you to create user-defined types. Note that the convention in F# is to write variables in all lowercase and user-defined types in upper camel case[33].

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