F Sharp Programming/Print version

F Sharp Programming

Preface

F# : Preface

How This Book Came To Be

> Note to contributors: normally a preface is written by the author of a book. Since this book might have several authors, feel free to write your own preface.

Written by Awesome Princess: Normally, authors choose to write a preface to a book about the book itself. But, just because I'm an egomaniac, I want to write about myself instead. Not because I'm an especially interesting person, but because my experiences with functional programming are relevant to the creation of this book.

So, in 2006, I was becoming bored with my job. The only kind of software I've ever written has been software that puts a GUI interface on top of a database, and I just became tired with it. I wanted to find an interesting programming job.

Just for fun, I started looking at job openings at different high tech companies (Google, eBay, Microsoft, Amazon, etc.). I noticed that all of the boring jobs—CRUD apps, simple web development—wanted programmers with Java, C#, or C++ experience. The interesting jobs—compiler programming, digital circuit verification, massively parallel computing, biometrics—sought programmers with experience in weird and unfamiliar languages. In particular:

- I read in Paul Graham's article Beating the Averages (http://www.paulgraham.com/avg.html) that the first version of Yahoo! Store was written largely in Lisp
- I came across job postings for Google looking for programmers with Haskell or Python experience in addition to C++.
- I read in an Erlang FAQ (http://www.erlang.org/faq/faq.html) that the Erlang programming language is the tool of choice for telecommunications providers like T-Mobile.
- I've heard for years that Lisp was a niche language during the Golden Age of AI research.
- I ran across numerous Microsoft job postings in the area of driver verification looking for OCaml programmers.

The most remarkable applications in the world aren't written in Java; they're written in these weird, obscure languages. More interestingly, the languages in the highest demand—Erlang, Haskell, Lisp, OCaml—were all functional programming languages, a wholly alien programming paradigm from my vantage point deep in C#-Land. I decided to supplement my programming wisdom by learning one of these obscure functional programming languages.

The choice between one language or another wasn't too hard to make. If I'm going to learn a new language, it needs to satisfy a few conditions: it should be practical enough for personal use, relatively speedy, useful to employers, and impress my friends when I tell them I learned a weird new language. Haskell was quite scary to me at the time, and I can't really exploit Erlang's concurrency with the tiny scope of the apps I write for myself. The choice came down to Lisp and OCaml; based on these comparisons of different languages (http://www.cs.ubc.ca/~murphyk/Software/which_language.html), I decided that OCaml's static-typing, speedy native code, tiny compiled binaries, and established niche in the financial market made it a good choice for me.

I learned OCaml and it completely changed my way of thinking. After using the language and keeping up with OCaml newsgroups, I heard about a .NET port of OCaml called F#. I figured I already knew the .NET BCL inside and out, and I was already familiar with OCaml, I could probably learn this language pretty quickly.

In August 2007, I took the time to get familiar with the F# language. While I picked up most of it fairly well, one thing I noticed about the language was how completely inaccessible it was to people trying to learn the language. The complete dearth of F# material out there just makes it impossible for beginners to learn F# as their first language. Even today, November 2008, there are only a handful of publications, but even as a person with many years of programming experience, I

struggled to follow along and comprehend the language.

For a long time, I wanted to write something that would actually be useful to F# neophytes, something that would contain everything anyone needs to know about the language into a single comprehensive resource. This book was originally started by a fellow Wikibookian in 2006, but no one had written any substantial content for it for nearly 2 years. I found this book and decided that, for the sake of people wanting to learn F#, I'd compile everything I knew about the language into a format that would be acceptable for first-time programmers.

I am happy with the way the book has been progressing. Ultimately, I'd like people to link to this book as the preferred, definitive F# tutorial on the Internet.

Introduction

F# : Introduction

Introducing F#

The **F#** programming language is part of Microsoft's family of .NET languages, which includes C#, Visual Basic.NET, JScript.NET, and others. As a .NET language, F# code compiles down to Common Language Infrastructure (CLI) byte code or Microsoft Intermediate Language (MSIL) which runs on top of the Common Language Runtime (CLR). All .NET languages share this common intermediate state which allows them to easily interoperate with one another and use the .NET Framework's Base Class Library (BCL), that is part of Standard Libraries.

In many ways, it's easy to think of F# as a .NET implementation of OCaml, a well-known functional programming language from the ML family of functional programming languages. Some of F#'s notable features include type inference, pattern matching, interactive scripting and debugging, higher order functions, and a well-developed object model which allows programmers to mix object-oriented and functional programming styles seamlessly.

A Brief History of F#

There are three dominant programming paradigms used today: functional, imperative, and object-oriented programming. Functional programming is the oldest of the three, beginning with Information Processing Language in 1956 and made popular with the appearance of Lisp in 1958. Of course, in the highly competitive world of programming languages in the early decades of computing, imperative programming established itself as the industry norm and preferred choice of scientific researchers and businesses with the arrival of Fortran in 1957 and COBOL in 1959.

While imperative languages became popular with businesses, functional programming languages continued to be developed primarily as highly specialized niche languages. For example, the APL programming language, developed in 1962, was developed to provide a consistent, mathematical notation for processing arrays. In 1973, Robin Milner at the University of Edinburgh developed the ML programming language to develop proof tactics for the LCF Theorem prover. Lisp continued to be used for years as the favored language of AI researchers.

ML stands out among other functional programming languages; its polymorphic functions made it a very expressive language, while its strong typing and immutable data structures made it possible to compile ML into very efficient machine code. ML's relative success spawned an entire family of ML-derived languages, including Standard ML, Caml, its most famous dialect called OCaml which unifies functional programming with object-oriented and imperative styles, and Haskell.

F# was developed in 2005 at Microsoft Research.[1] In many ways, F# is essentially a .Net implementation of OCaml, combining the power and expressive syntax of functional programming with the tens of thousands of classes which make up the .NET class library.

Why Learn F#?

Functional programming is often regarded as the best-kept secret of scientific modelers, mathematicians, artificial intelligence researchers, financial institutions, graphic designers, CPU designers, compiler programmers, and telecommunications engineers. Understandably, functional programming languages tend to be used in settings that perform heavy number crunching, abstract symbolic processing, or theorem proving. Of course, while F# is abstract enough to satisfy the needs of some highly technical niches, its simple and expressive syntax makes it suitable for CRUD apps, web pages, GUIs, games, and general-purpose programming.

Programming languages are becoming more functional every year. Features such as generic programming, type inference, list comprehensions, functions as values, and anonymous types, which have traditionally existed as staples of functional programming, have quickly become mainstream features of Java, C#, Delphi and even Fortran. We can expect next-generation programming languages to continue this trend in the future, providing a hybrid of both functional and imperative approaches that meet the evolving needs of modern programming.

F# is valuable to programmers at any skill level; it combines many of the best features of functional and object-oriented programming styles into a uniquely productive language.

References

1. http://research.microsoft.com

Getting Set Up

F# : Getting Set Up

Windows

At the time of this writing, its possible to run F# code through Visual Studio, through its interactive top-level F# Interactive (fsi), and compiling from the command line. This book will assume that users will compile code through Visual Studio or F# Interactive by default, unless specifically directed to compile from the command line.

Setup Procedure

F# can integrate with existing installations of Visual Studio 2008 and is included with Visual Studio 2010. Alternatively, users can download Visual Studio Express or Community (https://www.visualstudio.com/en-us/products/visual-studio-express-vs.aspx) for free, which will provide an F# pioneer with everything one needs to get started, including interactive debugging, breakpoints, watches, Intellisense, and support for F# projects. Make sure all instances of Visual Studio and Visual Studio Shell are closed before continuing.

To get started, users should download and install the latest version of the .NET Framework from Microsoft. Afterwards, download the latest version of F# (http://fsharp.org/) from the F# homepage on Microsoft Research, then execute the installation wizard. Users should also consider downloading and installing the F# PowerPack (http://fsharppowerpack.codeplex.com/), which contains handy extensions to the F# core library.

After successful installation, users will notice an additional folder in their start menu, "Microsoft F# 2.0.X.X." Additionally, users will notice that an entry for "F# Projects" has been added to the project types menu in Visual Studio. From here, users can create and run new F# projects.

It is a good idea to add the executable location (*e.g.* `c:\fsharp\bin\`) to the `%PATH%` environment variable, so you can access the compiler and the F# interactive environment (FSI) from any location.

As of Visual Studio 2012 the easiest way to get going is to install Visual Studio 2012 for Web at [1] (http://www.microsoft.com/visualstudio/eng/downloads) (even if you want to do desktop solution). You can then install "F# Tools for Visual Studio Express 2012 for Web" from [2] (http://www.microsoft.com/en-us/download/details.aspx?id=34675). Once this is done you can create F# projects. Search Nuget for additional F# project types.

Testing the Install

Hello World executable

Lets create the Hello World standalone application.

Create a text file called `hello.fs` containing the following code:

```
(* filename: hello.fs *)
let _ = printf "Hello world"
```

The underscore is used as a variable name when you are not interested in the value. All functions in F# return a value even if the main reason for calling the function is a side effect.

Save and close the file and then compile this file:

```
fsc -o hello.exe hello.fs
```

Now you can run `hello.exe` to produce the expected output.

F# Interactive Environment

Open a command-line console (hit the "Start" button, click on the "Run" icon and type `cmd` and hit ENTER).

Type `fsi` and hit ENTER. You will see the interactive console:

```
Microsoft F# Interactive, (c) Microsoft Corporation, All Rights Reserved
F# Version 1.9.6.2, compiling for .NET Framework Version v2.0.50727

Please send bug reports to fsbugs@microsoft.com
For help type #help;;

>
```

We can try some basic F# variable assignment (and some basic maths).

```
> let x = 5;;
val x : int

> let y = 20;;
val y : int

> y + x;;
val it : int = 25
```

Finally we quit out of the interactive environment

```
> #quit;;
```

Misc.

Adding to the PATH Environment Variable

1. Go to the Control Panel and choose System.
2. The System Properties dialog will appear. Select the Advanced tab and click the "Environment Variables...".
3. In the System Variables section, select the Path variable from the list and click the "Edit..." button.
4. In the Edit System Variable text box append a semicolon (;) followed by the executable path (e.g. `;C:\fsharp\bin\`)
5. Click on the "OK" button
6. Click on the "OK" button
7. Click on the "Apply" button

Now any command-line console will check in this location when you type `fsc` or `fsi`.

Mac OSX, Linux and UNIX

F# runs on Mac OSX, Linux and other Unix versions with the latest Mono (http://www.mono-project.com/). This is supported by the F# community group called the F# Software Foundation (http://fsharp.org/).

Installing interpreter and compiler

The F# Software Foundation (http://fsharp.org/) give latest instructions on getting started with F# on Linux and Mac. Once built and/or installed, you can use the "fsharpi" command to use the command-line interpreter, and "fsharpc" for the command-line compiler.

MonoDevelop add-in

The F# Software Foundation (http://fsharp.org/) also give instructions for installing the Monodevelop support for F#. This comes with project build system, code completion, and syntax highlighting support.

Emacs mode and other editors

The F# Software Foundation (http://fsharp.org/) also give instructions for using F# with other editors. An emacs mode for F# (https://fsharp.github.com/fsharpbinding) is also available on Github (https://github.com/fsharp/fsharpbinding#emacs-support).

Xamarin Studio for Mac OSX and Windows

F# runs on Mac OSX and Windows with the latest Xamarin Studio (https://www.xamarin.com/download). This is supported by Microsoft. Xamarin Studio is an IDE for developing cross-platform phone apps, but it runs on Mac OSX and implements F# with an interactive shell.

Basic Concepts

F# : Basic Concepts

Now that we have a working installation of F# we can explore the syntax of F# and the basics of functional programming. We'll start off in the Interactive F Sharp environment as this gives us some very valuable type information, which helps get to grips with what is actually going on in F#. Open F# interactive from the start menu, or open a command-line prompt and type `fsi`.

Major Features

Fundamental Data Types and Type System

In computer programming, every piece of data has a *type*, which, predictably, describes the type of data a programmer is working with. In F#, the fundamental data types are:

F# Type	.NET Type	Size	Range	Example	Represents
Integral Types					
`sbyte`	`System.SByte`	1 byte	-128 to 127	`42y` `-11y`	8-bit signed integer
`byte`	`System.Byte`	1 byte	0 to 255	`42uy` `200uy`	8-bit unsigned integer
`int16`	`System.Int16`	2 bytes	-32768 to 32767	`42s` `-11s`	16-bit signed integer
`uint16`	`System.UInt16`	2 bytes	0 to 65,535	`42us` `200us`	16-bit unsigned integer
`int/int32`	`System.Int32`	4 bytes	-2,147,483,648 to 2,147,483,647	`42` `-11`	32-bit signed integer
`uint32`	`System.UInt32`	4 bytes	0 to 4,294,967,295	`42u` `200u`	32-bit unsigned integer
`int64`	`System.Int64`	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`42L` `-11L`	64-bit signed integer
`uint64`	`System.UInt64`	8 bytes	0 to 18,446,744,073,709,551,615	`42UL` `200UL`	64-bit unsigned integer
`bigint`	`System.Numerics.BigInteger`	At least 4 bytes	any integer	`42I` `1499999999999999999999999999999I`	arbitrary precision integer
Floating Point Types					
`float32`	`System.Single`	4 bytes	±1.5e-45 to ±3.4e38	`42.0F` `-11.0F`	32-bit signed floating point number (7 significant digits)
`float`	`System.Double`	8 bytes	±5.0e-324 to ±1.7e308	`42.0` `-11.0`	64-bit signed floating point number (15-16 significant digits)
`decimal`	`System.Decimal`	16 bytes	±1.0e-28 to ±7.9e28	`42.0M` `-11.0M`	128-bit signed floating point number (28-29 significant digits)
`BigRational`	`Microsoft.FSharp.Math.BigRational`	At least 4 bytes	Any rational number.	`42N` `-11N`	Arbitrary precision rational number. Using this type requires a reference to FSharp.PowerPack.dll
Text Types					
`char`	`System.Char`	2 bytes	U+0000 to U+ffff	`'x'` `'\t'`	Single unicode characters
`string`	`System.String`	20 + (2 * string's length) bytes	0 to about 2 billion characters	`"Hello"` `"World"`	Unicode text
Other Types					
`bool`	`System.Boolean`	1 byte	Only two possible values, `true` or `false`	`true` `false`	Stores boolean values

F# is a fully object-oriented language, using an object model based on the .NET Common Language Infrastructure (CLI). As such, it has a single-inheritance, multiple interface object model, and allows programmers to declare classes, interfaces, and abstract classes. Notably, it has full support for generic class, interface, and function definitions; however, it lacks some OO features found in other languages, such as mixins and multiple inheritance.

F# also provides a unique array of data structures built directly into the syntax of the language, which include:

- Unit, the datatype with only one value, equivalent to `void` in C-style languages.
- Tuple types, which are ad hoc data structures that programmers can use to group related values into a single object.
- Record types, which are similar to tuples, but provide named fields to access data held by the record object.
- Discriminated unions, which are used to create very well-defined type hierarchies and hierarchical data structures.
- Lists, Maps, and Sets, which represent immutable versions of a stack, hashtable, and set data structures, respectively.
- Sequences, which represent a lazy list of items computed on-demand.
- Computation expressions, which serve the same purpose as monads in Haskell, allowing programmers to write continuation-style code in an imperative style.

All of these features will be further enumerated and explained in later chapters of this book.

F# is a statically typed language, meaning that the compiler knows the datatype of variables and functions at compile time. F# is also strongly typed, meaning that a variable bound to `int`s cannot be rebound to `string`s at some later point; an `int` variable is forever tied to `int` data.

Unlike C# and VB.Net, F# does not perform implicit casts, not even safe conversions (such as converting an `int` to a `int64`). F# requires explicit casts to convert between datatypes, for example:

```
> let x = 5;;

val x : int = 5

> let y = 6L;;
```

```
val y : int64 = 6L

> let z = x + y;;

  let z = x + y;;
  ------------^

stdin(5,13): error FS0001: The type 'int64' does not match the type 'int'
> let z = (int64 x) + y;;

val z : int64 = 11L
```

The mathematical operators `+, -, /, *,` and `%` are overloaded to work with different datatypes, but they require arguments on each side of the operator to have the same datatype. We get an error trying to add an `int` to an `int64`, so we have to cast one of our variables above to the other's datatype before the program will successfully compile.

Type Inference

Unlike many other strongly typed languages, F# often does not require programmers to use type annotations when declaring functions and variables. Instead, F# attempts to work out the types for you, based on the way that variables are used in code.

For example, let's take this function:

```
let average a b = (a + b) / 2.0
```

We have not used any type annotations: that is, we have not explicitly told the compiler the data type of `a` and `b`, nor have we indicated the type of the function's return value. If F# is a strongly, statically typed language, how does the compiler know the datatype of anything beforehand? That's easy, it uses simple deduction:

- The `+` and `/` operators are overloaded to work on different datatypes, but it defaults to integer addition and integer division without any extra information.
- `(a + b) / 2.0`, the value in bold has the type `float`. Since F# doesn't perform implicit casts, and it requires arguments on both sides of a math operator to have the same datatype, the value `(a + b)` must return a `float` as well.
- The `+` operator only returns `float` when both arguments on each side of the operator are `floats`, so `a` and `b` must be `floats` as well.
- Finally, since the return value of `float / float` is `float`, the `average` function must return a float.

This process is called type-inference. On most occasions, F# will be able to work out the types of data on its own without requiring the programmer to explicitly write out type annotations. This works just as well for small programs as large programs, and it can be a tremendous time-saver.

On those occasions where F# cannot work out the types correctly, the programmer can provide explicit annotations to guide F# in the right direction. For example, as mentioned above, math operators default to operations on integers:

```
> let add x y = x + y;;
val add : int -> int -> int
```

In absence of other information, F# determines that `add` takes two integers and returns another integer. If we wanted to use `floats` instead, we'd write:

```
> let add (x : float) (y : float) = x + y;;
val add : float -> float -> float
```

Pattern Matching

F#'s pattern matching is *similar* to an `if... then` or `switch` construct in other languages, but is much more powerful. Pattern matching allows a programmer to decompose data structures into their component parts. It matches values based on the *shape* of the data structure, for example:

```
type Proposition =  // type with possible expressions ... note recursion for all expressions except True
    | True          // essentially this is defining boolean logic
    | Not of Proposition
    | And of Proposition * Proposition
    | Or of Proposition * Proposition

let rec eval x =
    match x with
    | True -> true // syntax: Pattern-to-match -> Result
    | Not(prop) -> not (eval prop)
    | And(prop1, prop2) -> (eval prop1) && (eval prop2)
    | Or(prop1, prop2) -> (eval prop1) || (eval prop2)

let shouldBeFalse = And(Not True, Not True)

let shouldBeTrue = Or(True, Not True)

let complexLogic =
    And(And(True,Or(Not(True),True)),
        Or(And(True, Not(True)), Not(True)) )

printfn "shouldBeFalse: %b" (eval shouldBeFalse)    // prints False
printfn "shouldBeTrue: %b" (eval shouldBeTrue)      // prints True
printfn "complexLogic: %b" (eval complexLogic)      // prints False
```

The `eval` method uses pattern matching to recursively traverse and evaluate the abstract syntax tree. The `rec` keyword marks the function as recursive. Pattern matching will be explained in more detail in later chapters of this book.

Functional Programming Contrasted with Imperative Programming

F# is a mixed-paradigm language: it supports imperative, object-oriented, and functional styles of writing code, with heaviest emphasis on the latter.

Immutable Values vs Variables

The first mistake a newcomer to functional programming makes is thinking that the *let* construct is equivalent to assignment. Consider the following code:

```
let a = 1
(* a is now 1 *)
let a = a + 1
(* in F# this throws an error:  Duplicate definition of value 'a' *)
```

On the surface, this looks exactly like the familiar imperative pseudocode:

```
 a = 1
 // a is 1
 a = a + 1
 // a is 2
```

However, the nature of the F# code is very different. Every *let* construct introduces a new scope, and binds symbols to values in that scope. If execution escapes this introduced scope, the symbols are restored to their original meanings. This is clearly not identical to variable state mutation with assignment.

To clarify, let us desugar the F# code:

```
let a = 1 in
  ((* a stands for 1 here *);
   (let a = (* a still stands for 1 here *) a + 1 in (* a stands for 2 here *));
   (* a stands for 1 here, again *))
```

Indeed the code

```
let a = 1 in
  (printfn "%i" a;
   (let a = a + 1 in printfn "%i" a);
   printfn "%i" a)
```

prints out

```
 1
 2
 1
```

Once symbols are bound to values, they cannot be assigned a new value. The only way to change the meaning of a bound symbol is to *shadow* it by introducing a new binding for this symbol (for example, with a *let* construct, as in `let a = a + 1`), but this shadowing will only have a localized effect: it will only affect the newly introduced scope. F# uses so-called 'lexical scoping', which simply means that one can identify the scope of a binding by simply looking at the code. Thus the scope of the `let a = a + 1` binding in `(let a = a + 1 in ..)` is limited by the parentheses. With lexical scoping, there is no way for a piece of code to change the value of a bound symbol outside of itself, such as in the code that has called it.

Immutability is a great concept. Immutability allows programmers to pass values to functions without worrying that the function will change the value's state in unpredictable ways. Additionally, since value can't be mutated, programmers can process data shared across many threads without fear that the data will be mutated by another process; as a result, programmers can write multithreaded code without locks, and a whole class of errors related to race conditions and dead locking can be eliminated.

Functional programmers generally simulate state by passing extra parameters to functions; objects are "mutated" by creating an entirely new instance of an object with the desired changes and letting the garbage collector throw away the old instances if they are not needed. The resource overheads this style implies are dealt with by sharing structure. For example, changing the head of a singly-linked list of 1000 integers can be achieved by allocating a single new integer, reusing the tail of the original list (of length 999).

For the rare cases when mutation is really needed (for example, in number-crunching code which is a performance bottleneck), F# offers reference types and .NET mutable collections (such as arrays).

Recursion or Loops?

Imperative programming languages tend to iterate through collections with loops:

```
void ProcessItems(Item[] items)
{
    for(int i = 0; i < items.Length; i++)
    {
        Item myItem = items[i];
        proc(myItem); // process myItem
    }
}
```

This admits a direct translation to F# (type annotations for `i` and `item` are omitted because F# can infer them):

```
let processItems (items : Item []) =
  for i in 0 .. items.Length - 1 do
    let item = items.[i] in
      proc item
```

However, the above code is clearly not written in a functional style. One problem with it is that it traverses an array of items. For many purposes including enumeration, functional programmers would use a different data structure, a singly linked list. Here is an example of iterating over this data structure with pattern matching:

```
let rec processItems = function
  | []       -> ()  // empty list: end recursion
  | head :: tail -> // split list in first item (head) and rest (tail)
     proc head;
     processItems tail // recursively enumerate list
```

It is important to note that because the recursive call to `processItems` appears as the last expression in the function, this is an example of so-called *tail recursion*. The F# compiler recognizes this pattern and compiles `processItems` to a loop. The `processItems` function therefore runs in constant space and does not cause stack overflows.

F# programmers rely on tail recursion to structure their programs whenever this technique contributes to code clarity.

A careful reader has noticed that in the above example `proc` function was coming from the environment. The code can be improved and made more general by parameterizing it by this function (making `proc` a parameter):

```
let rec processItems proc = function
  | []       -> ()
  | hd :: tl ->
     proc hd;
     processItems proc tl // recursively enumerate list
```

This `processItems` function is indeed so useful that it has made it into the standard library under the name of `List.iter`.

For the sake of completeness it must be mentioned that F# includes generic versions of `List.iter` called `Seq.iter` (other List.* functions usually have Seq.* counterparts as well) that works on lists, arrays, and all other collections. F# also includes a looping construct that works for all collections implementing the `System.Collections.Generic.IEnumerable`:

```
for item in collection do
  process item
```

Function Composition Rather than Inheritance

Traditional OO uses *implementation inheritance* extensively; in other words, programmers create base classes with partial implementation, then build up object hierarchies from the base classes while overriding members as needed. This style has proven to be remarkably effective since the early 1990s, however this style is not contiguous with functional programming.

Functional programming aims to build simple, composable abstractions. Since traditional OO can only make an object's interface more complex, not simpler, inheritance is rarely used at all in F#. As a result, F# libraries tend to have fewer classes and very "flat" object hierarchies, as opposed to very deep and complex hierarchies found in equivalent Java or C# applications.

F# tends to rely more on object composition and delegation rather than inheritance to share snippets of implementation across modules.

Functions as First-Order Types

F# is a functional programming language, meaning that functions are first-order data types: they can be declared and used in exactly the same way that any other variable can be used.

In an imperative language like Visual Basic, there has traditionally been a fundamental difference between variables and functions.

```
Function MyFunc(param As Integer)
    MyFunc = (param * 2) + 7
End Function

' The program entry point; all statements must exist in a Sub or Function block.
Sub Main()
    Dim myVal As Integer
    ' Also statically typed as Integer, as the compiler (for newer versions of VB.NET) performs local type inference.
    Dim myParam = 2
    myVal = MyFunc(myParam)
End Sub
```

Notice the difference in syntax between defining and evaluating a function and defining and assigning a variable. In the preceding Visual Basic code we could perform a number of different actions with a variable we can:

- create a token (the variable name) and associate it with a type
- assign it a value
- interrogate its value
- pass it into a function or subroutine (a function that returns no value)
- return it from a function

Functional programming makes no distinction between values and functions, so we can consider functions to be equal to all other data types. That means that we can:

- create a token (the function variable name) and associate it with a type
- assign it a value (the actual calculation)
- interrogate its value (perform the calculation)
- pass a function as a parameter of another function or subroutine

- return a function as the result of another function

Structure of F# Programs

A simple, non-trivial F# program has the following parts:

```
open System

(* This is a
multi-line comment *)

// This is a single-line comment

let rec fib = function
    | 0 -> 0
    | 1 -> 1
    | n -> fib (n - 1) + fib (n - 2)

[<EntryPoint>]
let main argv =
    printfn "fib 5: %i" (fib 5)
    0
```

Most F# code files begin with a number of `open` statements used to import namespaces, allowing programmers to reference classes in namespaces without having to write fully qualified type declarations. This keyword is functionally equivalent to the `using` directive in C# and `Imports` directive in VB.Net. For example, the `Console` class is found under the `System` namespace; without importing the namespace, a programmer would need to access the `Console` class through its fully qualified name, `System.Console`.

The body of the F# file usually contains functions to implement the business logic in an application.

Finally, many F# application exhibit this pattern:

```
[<EntryPoint>]
let main argv =
    // Code to be executed
    0
```

The entry point of an F# program is marked by the [<EntryPoint>] attribute, and following it must be a function that accepts an array of strings as input and returns an integer (by default, 0)

Values and Functions

F# : Declaring Values and Functions

Compared to other .NET languages such as C# and VB.Net, F# has a somewhat terse and minimalistic syntax. To follow along in this tutorial, open F# Interactive (fsi) or Visual Studio and run the examples.

Declaring Variables

The most ubiquitous, familiar keyword in F# is the `let` keyword, which allows programmers to declare functions and variables in their applications.

For example:

```
let x = 5
```

This declares a variable called `x` and assigns it the value `5`. Naturally, we can write the following:

```
let x = 5
let y = 10
let z = x + y
```

`z` now holds the value 15.

A complete program looks like this:

```
let x = 5
let y = 10
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

The statement `printfn` prints text out to the console window. As you might have guessed, the code above prints out the values of `x`, `y`, and `z`. This program results in the following:

```
x: 5
y: 10
z: 15
```

> Note to F# Interactive users: all statements in F# Interactive are terminated by `;;` (two semicolons). To run the program above in fsi, copy and paste the text above into the fsi window, type `;;`, then hit enter.

Values, Not Variables

In F#, "variable" is a misnomer. In reality, all "variables" in F# are immutable; in other words, once you bind a "variable" to a value, it's stuck with that value forever. For that reason, most F# programmers prefer to use "value" rather than "variable" to describe `x`, `y`, and `z` above. Behind the scenes, F# actually compiles the "variables" above as static read-only properties.

Declaring Functions

There is little distinction between functions and values in F#. You use the same syntax to write a function as you use to declare a value:

```
let add x y = x + y
```

`add` is the name of the function, and it takes two parameters, `x` and `y`. Notice that each distinct argument in the functional declaration is separated by a space. Similarly, when you execute this function, successive arguments are separated by a space:

```
let z = add 5 10
```

This assigns `z` the return value of this function, which in this case happens to be `15`.

Naturally, we can pass the return value of functions directly into other functions, for example:

```
let add x y = x + y

let sub x y = x - y

let printThreeNumbers num1 num2 num3 =
    printfn "num1: %i" num1
    printfn "num2: %i" num2
    printfn "num3: %i" num3

printThreeNumbers 5 (add 10 7) (sub 20 8)
```

This program outputs:

```
num1: 5
num2: 17
num3: 12
```

Notice that I have to surround the calls to `add` and `sub` functions with parentheses; this tells F# to treat the value in parentheses as a single argument.

Otherwise, if we wrote `printThreeNumbers 5 add 10 7 sub 20 8`, its not only incredibly difficult to read, but it actually passes 7 parameters to the function, which is obviously incorrect.

Function Return Values

Unlike many other languages, F# functions do not have an explicit keyword to return a value. Instead, the return value of a function is simply the value of the last statement executed in the function. For example:

```
let sign num =
    if num > 0 then "positive"
    elif num < 0 then "negative"
    else "zero"
```

This function takes an integer parameter and returns a string. As you can imagine, the F# function above is equivalent to the following C# code:

```
string Sign(int num)
{
    if (num > 0) return "positive";
    else if (num < 0) return "negative";
    else return "zero";
}
```

Just like C#, F# is a strongly typed language. A function can only return one datatype; for example, the following F# code will not compile:

```
let sign num =
    if num > 0 then "positive"
    elif num < 0 then "negative"
    else 0
```

If you run this code in fsi, you get the following error message:

```
> let sign num =
    if num > 0 then "positive"
    elif num < 0 then "negative"
    else 0;;

      else 0;;
  ---------^
```

```
stdin(7,10): error FS0001: This expression was expected to have type    string    but here has type    int
```

The error message is quite explicit: F# has determined that this function returns a `string`, but the last line of the function returns an `int`, which is an error.

Interestingly, *every* function in F# has a return value; of course, programmers don't always write functions that return useful values. F# has a special datatype called `unit`, which has just one possible value: `()`. Functions return `unit` when they don't need to return any value to the programmer. For example, a function that prints a string to the console obviously doesn't have a return value:

```
let helloWorld () = printfn "hello world"
```

This function takes `unit` parameter and returns `()`. You can think of `unit` as the equivalent to `void` in C-style languages.

How To Read Arrow Notation

All functions and values in F# have a data type. Open F# Interactive and type the following:

```
> let addAndMakeString x y = (x + y).ToString();;
```

F# reports the data type using chained arrow notation as follows:

```
val addAndMakeString : x:int -> y:int -> string
```

Data types are read from left to right. Before muddying the waters with a more accurate description of how F# functions are built, consider the basic concept of Arrow Notation: starting from the left, our function takes two `int` inputs and returns a `string`. A function only has one return type, which is represented by the rightmost data type in chained arrow notation.

We can read the following data types as follows:

`int -> string`

> takes one `int` input, returns a `string`

`float -> float -> float`

> takes two `float` inputs, returns another `float`

`int -> string -> float`

> takes an `int` and a `string` input, returns a `float`

This description is a good introductory way to understand Arrow Notation for a beginner—and if you are new to F# feel free to stop here until you get your feet wet. For those who feel comfortable with this concept as described, the actual way in which F# is implementing these calls is via **currying** the function.

Partial Function Application

While the above description of Arrow Notation is intuitive, it is not entirely accurate due to the fact that **F# implicitly curries (https://en.wikipedia.org/wiki/Currying) functions**. This means that a function only ever has a single argument and a single return type, quite at odds with the previous description of Arrow Notation above where in the second and third example two arguments are passed to a function. In reality, a function in F# **only ever** has a single argument and a single return type. How can this be? Consider this type:

`float -> float -> float`

since a function of this type is implicitly curried by F#, there is a two step process to resolve the function when called with two arguments

1. a function is called with the first argument that **returns a function** that takes a float and returns a float. To help clarify currying, lets call this function **funX** (note that this naming is just for illustration purposes—the function that gets created by the runtime is anonymous).
2. the second function (**'funX'** from step 1 above) is called with the second argument, returning a float

So, if you provide two floats, the result appears as if the function takes two arguments, though this is not actually how the runtime behaves. The concept of currying will probably strike a developer not steeped in functional concepts as very strange and non-intuitive—even needlessly redundant and inefficient, so before attempting a further explanation, consider *the benefits of curried functions* via an example:

```
let addTwoNumbers x y = x + y
```

this type has the signature of

`int -> int -> int`

then this function:

```
let add5ToNumber = addTwoNumbers 5
```

with the type signature of **`(int -> int)`**. Note that the body of `add5ToNumber` calls `addTwoNumbers` with only one argument—not two. It **returns a function** that takes an int and returns an int. In other words, `add5toNumber` **partially applies** the `addTwoNumbers` function.

```
> let z = add5ToNumber 6;;

val z : int = 11
```

This partial application of a function with multiple argument exemplifies the power of curried functions. It allows **deferred application** of the function, allowing for more modular development and code re-use—we can re-use the `addTwoNumbers` function to create a new function via partial application. From this, you can glean the power of function currying: it is always breaking down function application to the smallest possible elements, facilitating greater chances for code-reuse and modularity.

Take another example, illustrating the use of partially applied functions as a bookkeeping technique. Note the type signature of `holdOn` is a function (int -> int) since it is the partial application of `addTwoNumbers`

```
> let holdOn = addTwoNumbers 7;;

val holdOn : (int -> int)

> let okDone = holdOn 8;;

val okDone : int = 15
```

Here we define a new function `holdOn` on the fly just to keep track of the first value to add. Then later we apply this new 'temp' function `holdOn` with another value which returns an int. Partially applied functions—enabled by currying—is a very powerful means of controlling complexity in F#. In short, the reason for the indirection resulting from currying function calls affords partial function application and all the benefits it supplies. In other words, the goal of partial function application is enabled by implicit currying.

So while the Arrow Notation is a good shorthand for understanding the type signature of a function, it does so at the price of oversimplification, for a function with the type signature of

```
f : int -> int -> int
```

is actually (when taking into consideration the implicit currying):

```
// curried version pseudo-code
f: int -> (int -> int)
```

In other words, f is a function that takes an int and returns a function that takes an int and returns an int. Moreover,

```
f: int -> int -> int -> int
```

is a simplified shorthand for

```
// curried version pseudo-code
f: int -> (int -> (int -> int))
```

or, in very difficult to decode English: f is a function that takes an int and returns a function that takes an int and returns a function that takes an int and returns an int. Yikes!

Nested Functions

F# allows programmers to nest functions inside other functions. Nested functions have a number of applications, such as hiding the complexity of inner loops:

```
let sumOfDivisors n =
    let rec loop current max acc =
        if current > max then
            acc
        else
            if n % current = 0 then
                loop (current + 1) max (acc + current)
            else
                loop (current + 1) max acc
    let start = 2
    let max = n / 2     (* largest factor, apart from n, cannot be > n / 2 *)
    let minSum = 1 + n  (* 1 and n are already factors of n *)
    loop start max minSum

printfn "%d" (sumOfDivisors 10)
(* prints 18, because the sum of 10's divisors is 1 + 2 + 5 + 10 = 18 *)
```

The outer function `sumOfDivisors` makes a call to the inner function `loop`. Programmers can have an arbitrary level of nested functions as need requires.

Generic Functions

In programming, a generic function is a function that returns an indeterminate type `t` without sacrificing type safety. A generic type is different from a concrete type such as an `int` or a `string`; a generic type represents a *type to be specified later*. Generic functions are useful because they can be generalized over many different types.

Let's examine the following function:

```
let giveMeAThree x = 3
```

F# derives type information of variables from the way variables are used in an application, but F# can't constrain the value `x` to any particular concrete type, so F# generalizes `x` to the generic type `'a`:

`'a -> int`

> this function takes a generic type `'a` and returns an `int`.

When you call a generic function, the compiler substitutes a function's generic types with the data types of the values passed to the function. As a demonstration, let's use the following function:

```
let throwAwayFirstInput x y = y
```

Which has the type `'a -> 'b -> 'b`, meaning that the function takes a generic `'a` and a generic `'b` and returns a `'b`.

Here are some sample inputs and outputs in F# interactive:

```
> let throwAwayFirstInput x y = y;;

val throwAwayFirstInput : 'a -> 'b -> 'b

> throwAwayFirstInput 5 "value";;
val it : string = "value"

> throwAwayFirstInput "thrownAway" 10.0;;
val it : float = 10.0

> throwAwayFirstInput 5 30;;
val it : int = 30
```

`throwAwayFirstInput 5 "value"` calls the function with an `int` and a `string`, which substitutes `int` for `'a` and `string` for `'b`. This changes the data type of `throwAwayFirstInput` to `int -> string -> string`.

`throwAwayFirstInput "thrownAway" 10.0` calls the function with a `string` and a `float`, so the function's data type changes to `string -> float -> float`.

`throwAwayFirstInput 5 30` just happens to call the function with two `int`s, so the function's data type is incidentally `int -> int -> int`.

Generic functions are strongly typed. For example:

```
let throwAwayFirstInput x y = y
let add x y = x + y

let z = add 10 (throwAwayFirstInput "this is a string" 5)
```

The generic function `throwAwayFirstInput` is defined again, then the `add` function is defined and it has the type `int -> int -> int`, meaning that this function must be called with two `int` parameters.

Then `throwAwayFirstInput` is called, as a parameter to `add`, with two parameters on itself, the first one of type string and the second of type int. This call to `throwAwayFirstInput` ends up having the type `string -> int -> int`. Since this function has the return type `int`, the code works as expected:

```
> add 10 (throwAwayFirstInput "this is a string" 5);;
val it : int = 15
```

However, we get an error when we reverse the order of the parameters to `throwAwayFirstInput`:

```
> add 10 (throwAwayFirstInput 5 "this is a string");;

  add 10 (throwAwayFirstInput 5 "this is a string");;
  ------------------------------^^^^^^^^^^^^^^^^^^

stdin(13,31): error FS0001: This expression has type
        string
but is here used with type
        int.
```

The error message is very explicit: The `add` function takes two `int` parameters, but `throwAwayFirstInput 5 "this is a string"` has the return type `string`, so we have a type mismatch.

Later chapters will demonstrate how to use generics in creative and interesting ways.

Pattern Matching Basics

F# : Pattern Matching Basics

Pattern matching is used for control flow; it allows programmers to look at a value, test it against a series of conditions, and perform certain computations depending on whether that condition is met. While pattern matching is conceptually similar to a series of `if ... then` statements in other languages, F#'s pattern matching is much more flexible and powerful.

Pattern Matching Syntax

In high level terms, pattern matching resembles this:

```
match expr with
| pat1 -> result1
| pat2 -> result2
| pat3 when expr2 -> result3
| _ -> defaultResult
```

Each `|` defines a condition, the `->` means "if the condition is true, return this value...". The `_` is the *default pattern*, meaning that it matches anything, sort of like a wildcard.

Using a real example, it's easy to calculate the **nth** Fibonacci number using pattern matching syntax:

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fib (n - 1) + fib (n - 2)
```

We can experiment with this function in fsi:

```
 fib 1;;
val it : int = 1
> fib 2;;
val it : int = 1
> fib 5;;
val it : int = 5
> fib 10;;
val it : int = 55
```

It's possible to chain together multiple conditions which return the same value. For example:

```
> let greeting name =
    match name with
    | "Steve" | "Kristina" | "Matt" -> "Hello!"
    | "Carlos" | "Maria" -> "Hola!"
    | "Worf" -> "nuqneH!"
    | "Pierre" | "Monique" -> "Bonjour!"
    | _ -> "DOES NOT COMPUTE!";;

val greeting : string -> string

> greeting "Monique";;
val it : string = "Bonjour!"
> greeting "Pierre";;
val it : string = "Bonjour!"
> greeting "Kristina";;
val it : string = "Hello!"
> greeting "Sakura";;
val it : string = "DOES NOT COMPUTE!"
```

Alternative Pattern Matching Syntax

Pattern matching is such a fundamental feature that F# has a shorthand syntax for writing pattern matching functions using the `function` keyword:

```
let something = function
    | test1 -> value1
    | test2 -> value2
    | test3 -> value3
```

It may not be obvious, but a function defined in this way actually takes a single input. Here's a trivial example of the alternative syntax:

```
let getPrice = function
    | "banana" -> 0.79
    | "watermelon" -> 3.49
    | "tofu" -> 1.09
    | _ -> nan (* nan is a special value meaning "not a number" *)
```

Although it doesn't appear as if the function takes any parameters, it actually has the type `string -> float`, so it takes a single `string` parameter and returns a `float`. You call this function in exactly the same way that you'd call any other function:

```
> getPrice "tofu";;
val it : float = 1.09
```

```
> getPrice "banana";;
val it : float = 0.79

> getPrice "apple";;
val it : float = nan
```

You can add additional parameters to the definition:

```
let getPrice taxRate = function
    | "banana" -> 0.79 * (1.0 + taxRate)
    | "watermelon" -> 3.49 * (1.0 + taxRate)
    | "tofu" -> 1.09 * (1.0 + taxRate)
    | _ -> nan (* nan is a special value meaning "not a number" *)
val getPrice : taxRate:float -> _arg1:string -> float
```

Calling this gives:

```
> getPrice 0.10 "tofu";;
val it : float = 1.199
```

Note that the added parameters in the call come before the implicit parameter against which the match is made.

Comparison To Other Languages

F#'s pattern matching syntax is subtly different from "switch statement" structures in imperative languages, because each case in a pattern has a return value. For example, the `fib` function is equivalent to the following C#:

```
int fib(int n)
{
    switch(n)
    {
        case 0: return 0;
        case 1: return 1;
        default: return fib(n - 1) + fib(n - 2);
    }
}
```

Like all functions, pattern matches can only have one return type.

Binding Variables with Pattern Matching

Pattern matching is not a fancy syntax for a `switch` structure found in other languages, because it does not necessarily match against *values*, it matches against the *shape* of data.

F# can automatically bind values to identifiers if they match certain patterns. This can be especially useful when using the alternative pattern matching syntax, for example:

```
let rec factorial = function
    | 0 | 1 -> 1
    | n -> n * factorial (n - 1)
```

The variable `n` is a *pattern*. If the `factorial` function is called with a `5`, the `0` and `1` patterns will fail, but the last pattern will match and bind the value to the identifier `n`.

> *Note to beginners:* variable binding in pattern matching often looks strange to beginners, however it is probably the most powerful and useful feature of F#. Variable binding is used to decompose data structures into component parts and allow programmers to examine each part; however, data structure decomposition is too advanced for most F# beginners, and the concept is difficult to express using simple types like `int`s and `string`s. This book will discuss how to decompose data structures using pattern matching in later chapters.

Using Guards within Patterns

Occasionally, it's not enough to match an input against a particular value; we can add filters, or guards, to patterns using the `when` keyword:

```
let sign = function
    | 0 -> 0
    | x when x < 0 -> -1
    | x when x > 0 -> 1
```

The function above returns the sign of a number: -1 for negative numbers, +1 for positive numbers, and '0' for 0:

```
> sign -55;;
val it : int = -1

> sign 108;;
val it : int = 1

> sign 0;;
val it : int = 0
```

Variable binding is useful because it's often required to implement guards.

Pay Attention to F# Warnings

Note that F#'s pattern matching works from top to bottom: it tests a value against each pattern, and returns the value of the first pattern which matches. It is possible for programmers to make mistakes, such as placing a general case above a specific (which would prevent the specific case from ever being matched), or writing a pattern which doesn't match all possible inputs. F# is smart enough to notify the programmer of these types of errors.

Example With Incomplete Pattern Matches

```
> let getCityFromZipcode zip =
    match zip with
    | 68528 -> "Lincoln, Nebraska"
    | 90210 -> "Beverly Hills, California";;

      match zip with
  ----------^^^^

stdin(12,11): warning FS0025: Incomplete pattern matches on this expression.
For example, the value '0' will not be matched

val getCityFromZipcode : int -> string
```

While this code is valid, F# informs the programmer of the possible error. F# warns us for a reason:

```
> getCityFromZipcode 68528;;
val it : string = "Lincoln, Nebraska"
> getCityFromZipcode 32566;;
Microsoft.FSharp.Core.MatchFailureException:
Exception of type 'Microsoft.FSharp.Core.MatchFailureException' was thrown.
   at FSI_0018.getCityFromZipcode(Int32 zip)
   at <StartupCode$FSI_0020>.$FSI_0020._main()
stopped due to error
```

F# will throw an exception if a pattern isn't matched. The obvious solution to this problem is to write patterns which are complete.

On occasions when a function genuinely has a limited range of inputs, its best to adopt this style:

```
let apartmentPrices numberOfRooms =
    match numberOfRooms with
    | 1 -> 500.0
    | 2 -> 650.0
    | 3 -> 700.0
    | _ -> failwith "Only 1-, 2-, and 3- bedroom apartments available at this complex"
```

This function now matches any possible input, and will fail with an explanatory informative error message on invalid inputs (this makes sense, because who would rent a negative 42 bedroom apartment?).

Example With Unmatched Patterns

```
> let greeting name =
    match name with
    | "Steve" -> "Hello!"
    | "Carlos" -> "Hola!"
    | _ -> "DOES NOT COMPUTE!"
    | "Pierre" -> "Bonjour";;

      | "Pierre" -> "Bonjour";;
  ------^^^^^^^^^

stdin(22,7): warning FS0026: This rule will never be matched.

val greeting : string -> string
```

Since the pattern _ matches anything, and since F# evaluates patterns from top to bottom, its not possible for the code to ever reach the pattern `"Pierre"`.

Here is a demonstration of this code in fsi:

```
> greeting "Steve";;
val it : string = "Hello!"
> greeting "Ino";;
val it : string = "DOES NOT COMPUTE!"
> greeting "Pierre";;
val it : string = "DOES NOT COMPUTE!"
```

The first two lines return the correct output, because we've defined a pattern for `"Steve"` and nothing for `"Ino"`.

However, the third line is wrong. We have an entry for `"Pierre"`, but F# never reaches it. The best solution to this problem is to deliberately arrange the order of conditions from most specific to most general.

> *Note to beginners:* The code above contains an error, but it will not throw an exception. These are the worst kinds of errors to have, much worse than an error which throws an exception and crashes an app, because this error puts our program in an invalid state and silently continues on its way. An error like this might occur early in a program's life cycle, but may not show its effects for a long time (it could take minutes, days, or weeks before someone notices the buggy behavior). Ideally, we want buggy behavior to be as "close" to its source as possible, so if a program enters an invalid state, it *should* throw an exception immediately. To prevent this sort of problem it is usually a good idea to set the compiler flag that treats all warnings as errors; then the code will not compile thus preventing the problem right at the beginning.

Recursion

F# : Recursion and Recursive Functions

A recursive function is a function which calls itself. Interestingly, in contrast to many other languages, functions in F# are not recursive by default. A programmer needs to explicitly mark a function as recursive using the `rec` keyword:

```
let rec someFunction = ...
```

Examples

Factorial in F#

The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example, 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720.

In mathematics, the factorial is defined as follows:

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

Naturally, we'd calculate a factorial by hand using the following:

```
fact(6) =
        = 6 * fact(6 - 1)
        = 6 * 5 * fact(5 - 1)
        = 6 * 5 * 4 * fact(4 - 1)
        = 6 * 5 * 4 * 3 * fact(3 - 1)
        = 6 * 5 * 4 * 3 * 2 * fact(2 - 1)
        = 6 * 5 * 4 * 3 * 2 * 1 * fact(1 - 1)
        = 6 * 5 * 4 * 3 * 2 * 1 * 1
        = 720
```

In F#, the factorial function can be written concisely as follows:

```
let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)
```

But note that this function as it stands returns 1 for all negative numbers but factorial is undefined for negative numbers. This means that in real production programs you must either design the rest of the program so that factorial can never be called with a a negative number or trap negative input and throw an exception. Exceptions will be discussed in a later chapter.

Here's a complete program:

```
open System

let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)

(* // can also be written using pattern matching syntax:
let rec fact = function
    | n when n < 1 -> 1
    | n -> n * fact (n - 1) *)

Console.WriteLine(fact 6)
```

Greatest Common Divisor (GCD)

The greatest common divisor, or GCD function, calculates the largest integer number which evenly divides two other integers. For example, largest number that evenly divides 259 and 111 is 37, denoted GCD(259, 111) = 37.

Euclid discovered a remarkably simple recursive algorithm for calculating the GCD of two numbers:

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, remainder(x, y)) & \text{if } x >= y \text{ and } y > 0 \end{cases}$$

To calculate this by hand, we'd write:

```
gcd(259, 111)   = gcd(111, 259% 111)
                = gcd(111, 37)
                = gcd(37, 111% 37)
                = gcd(37, 0)
                = 37
```

In F#, we can use the % (modulus) operator to calculate the remainder of two numbers, so naturally we can define the GCD function in F# as follows:

```
open System

let rec gcd x y =
    if y = 0 then x
    else gcd y (x % y)

Console.WriteLine(gcd 259 111) // prints 37
```

Tail Recursion

Let's say we have a function A which, at some point, calls function B. When B finishes executing, the CPU must continue executing A from the point where it left off. To "remember" where to return, the function A passes a *return address* as an extra argument to B on the stack; B jumps back to the return address when it finishes executing. This means calling a function, even one that doesn't take any parameters, consumes stack space, and it's extremely easy for a recursive function to consume all of the available memory on the stack.

A *tail recursive* function is a special case of recursion in which the last instruction executed in the method is the recursive call. F# and many other functional languages can optimize tail recursive functions; since no extra work is performed after the recursive call, there is no need for the function to remember where it came from, and hence no reason to allocate additional memory on the stack.

F# optimizes tail-recursive functions by telling the CLR to drop the current stack frame before executing the target function. As a result, tail-recursive functions can recurse indefinitely without consuming stack space.

Here's non-tail recursive function:

```
> let rec count n =
    if n = 1000000 then
        printfn "done"
    else
        if n % 1000 = 0 then
            printfn "n: %i" n

        count (n + 1) (* recursive call *)
        () (* <-- This function is not tail recursive
              because it performs extra work (by
              returning unit) after
              the recursive call is invoked. *);;

val count : int -> unit

> count 0;;
n: 0
n: 1000
n: 2000
n: 3000
...
n: 58000
n: 59000
Session termination detected. Press Enter to restart.
Process is terminated due to StackOverflowException.
```

Let's see what happens if we make the function properly tail-recursive:

```
> let rec count n =
    if n = 1000000 then
        printfn "done"
    else
        if n % 1000 = 0 then
            printfn "n: %i" n

        count (n + 1) (* recursive call *);;

val count : int -> unit

> count 0;;
n: 0
n: 1000
n: 2000
n: 3000
n: 4000
...
n: 995000
n: 996000
n: 997000
n: 998000
n: 999000
done
```

If there was no check for `n = 1000000`, the function would run indefinitely. It's important to ensure that all recursive function have a *base case* to ensure they terminate eventually.

How to Write Tail-Recursive Functions

Let's imagine that, for our own amusement, we wanted to implement a multiplication function in terms of the more fundamental function of addition. For example, we know that `6 * 4` is the same as `6 + 6 + 6 + 6`, or more generally we can define multiplication recursively as `M(a, b) = a + M(a, b - 1), b > 1`. In F#, we'd write this function as:

```
let rec slowMultiply a b =
    if b > 1 then
        a + slowMultiply a (b - 1)
    else
        a
```

It may not be immediately obvious, but this function is not tail recursive. It might be more obvious if we rewrote the function as follows:

```
let rec slowMultiply a b =
    if b > 1 then
        let intermediate = slowMultiply a (b - 1) (* recursion *)
        let result = a + intermediate (* <-- additional operations *)
        result
    else a
```

The reason it is not tail recursive is because after the recursive call to `slowMultiply`, the result of the recursion has to added to `a`. Remember tail recursion needs the last operation to be the recursion.

Since the `slowMultiply` function isn't tail recursive, it throws a `StackOverFlowException` for inputs which result in very deep recursion:

```
> let rec slowMultiply a b =
    if b > 1 then
        a + slowMultiply a (b - 1)
    else
        a;;

val slowMultiply : int -> int -> int

> slowMultiply 3 9;;
val it : int = 27

> slowMultiply 2 14;;
val it : int = 28

> slowMultiply 1 100000;;

Process is terminated due to StackOverflowException.
Session termination detected. Press Enter to restart.
```

It's possible to re-write most recursive functions into their tail-recursive forms using an accumulating parameter:

```
> let slowMultiply a b =
    let rec loop acc counter =
        if counter > 1 then
            loop (acc + a) (counter - 1) (* tail recursive *)
        else
            acc
    loop a b;;

val slowMultiply : int -> int -> int

> slowMultiply 3 9;;
val it : int = 27

> slowMultiply 2 14;;
val it : int = 28

> slowMultiply 1 100000;;
val it : int = 100000
```

The accumulator parameter in the inner loop holds the state of our function throughout each recursive iteration.

Exercises

Solutions.

Faster Fib Function

The following function calculates the *nth* number in the Fibonacci sequence:

```
let rec fib = function
    | n when n=0I -> 0I
    | n when n=1I -> 1I
    | n -> fib(n - 1I) + fib(n - 2I)
```

> **Note**: The function above has the type `val fib : bigint -> bigint`. Previously, we've been using the `int` or `System.Int32` type to represent numbers, but this type has a maximum value of `2,147,483,647`. The type `bigint` is used for *arbitrary size integers* such as integers with billions of digits. The maximum value of `bigint` is constrained only by the available memory on a users machine, but for most practical computing purposes we can say this type is boundless.

The function above is neither tail-recursive nor particularly efficient with a computational complexity $O(2^n)$. The tail-recursive form of this function has a computational complexity of $O(n)$. Re-write the function above so that it's tail recursive.

You can verify the correctness of your function using the following:

```
fib(0I) = 0
fib(1I) = 1
fib(2I) = 1
fib(3I) = 2
fib(4I) = 3
fib(5I) = 5
fib(10I) = 55
fib(100I) = 354224848179261915075
```

Additional Reading

- Understanding Tail Recursion (http://blogs.msdn.com/chrsmith/archive/2008/08/07/understanding-tail-recursion.aspx)
- How can I implement a tail-recursive append? (http://stackoverflow.com/q/2867514/40516)

Higher Order Functions

F# : Higher Order Functions

A higher-order function is a function that takes another function as a parameter, or a function that returns another function as a value, or a function that does both.

Familiar Higher Order Functions

To put higher order functions in perspective, if you've ever taken a first-semester course on calculus, you're undoubtedly familiar with two functions: the limit function and the derivative function.

The limit function is defined as follows:

$$\lim_{x \to p} f(x) = L$$

The limit function, `lim`, takes another function `f(x)` as a parameter, and it returns a value `L` to represent the limit.

Similarly, the derivative function is defined as follows:

$$deriv(f(x)) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} = f'(x)$$

The derivative function, `deriv`, takes a function `f(x)` as a parameter, and it returns a completely different function `f'(x)` as a result.

In this respect, we can correctly assume the limit and derivative functions are higher-order functions. If we have a good understanding of higher-order functions in mathematics, then we can apply the same principles in F# code.

In F#, we can pass a function to another function just as if it were a literal value, and call it just like any other function. For example, here's a very trivial function:

```
let passFive f = (f 5)
```

In F# notation, `passFive` has the following type:

```
val passFive : (int -> 'a) -> 'a
```

In other words, `passFive` takes a function `f`, where `f` must take an `int` and return any generic type `'a`. Our function `passFive` has the return type `'a` because we don't know the return type of `f 5` in advance.

```
open System

let square x = x * x

let cube x = x * x * x

let sign x =
    if x > 0 then "positive"
    else if x < 0 then "negative"
    else "zero"

let passFive f = (f 5)

printfn "%A" (passFive square)  // 25
printfn "%A" (passFive cube)    // 125
printfn "%A" (passFive sign)    // "positive"
```

These functions have the following types:

```
val square : int -> int
val cube : int -> int
val sign : int -> string
val passFive : (int -> 'a) -> 'a
```

Unlike many other languages, F# makes no distinction between functions and values. We pass functions to other functions in the exact same way that we pass ints, strings, and other values.

Creating a Map Function

A map function converts one type of data to another type of data. A simple map function in F# looks like this:

```
let map item converter = converter item
```

This has the type `val map : 'a -> ('a -> 'b) -> 'b`. In other words, `map` takes two parameters: an item 'a, and a function that takes an `'a` and returns a `'b`; map returns a `'b`.

Let's examine the following code:

```
open System

let map x f = f x

let square x = x * x

let cubeAndConvertToString x =
    let temp = x * x * x
    temp.ToString()

let answer x =
    if x = true then "yes"
    else "no"

let first = map 5 square
let second = map 5 cubeAndConvertToString
let third = map true answer
```

These functions have the following signatures:

```
val map : 'a -> ('a -> 'b) -> 'b

val square : int -> int
val cubeAndConvertToString : int -> string
val answer : bool -> string

val first : int
val second : string
val third : string
```

The `first` function passes a datatype `int` and a function with the signature `(int -> int)`; this means the placeholders `'a` and `'b` in the map function both become `int`s.

The `second` function passes a datatype `int` and a function `(int -> string)`, and `map` predictably returns a `string`.

The `third` function passes a datatype `bool` and a function `(bool -> string)`, and `map` returns a `string` just as we expect.

Since our generic code is typesafe, we would get an error if we wrote:

```
let fourth = map true square
```

Because the `true` constrains our function to a type `(bool -> 'b)`, but the `square` function has the type `(int -> int)`, so it's obviously incorrect.

The Composition Function (<< operator)

In algebra, the composition function is defined as `compose(f, g, x) = f(g(x))`, denoted $f \circ g$. In F#, the composition function is defined as follows:

```
let inline (<<) f g x = f (g x)
```

Which has the somewhat cumbersome signature `val << : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`.

If I had two functions:

f(x) = x^2
g(x) = -x/2 + 5

And I wanted to model $f \circ g$, I could write:

```
open System

let f x = x*x
let g x = -x/2.0 + 5.0

let fog = f << g

Console.WriteLine(fog 0.0) // 25
Console.WriteLine(fog 1.0) // 20.25
Console.WriteLine(fog 2.0) // 16
Console.WriteLine(fog 3.0) // 12.25
Console.WriteLine(fog 4.0) // 9
Console.WriteLine(fog 5.0) // 6.25
```

Note that `fog` doesn't return a value, it returns another function whose signature is `(float -> float)`.

There's no reason why the compose function must be limited to numbers. Since it's generic, it can work with any datatype, such as `int array`s, `tuple`s, `string`s, and so on.

There also exists the `>>` operator, which similarly performs function composition, but in reverse order. It is defined as follows:

```
let inline (>>) f g x = g (f x)
```

This operator's signature is as follows: `val >> : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`.

The advantage of doing composition using the `>>` operator is that the functions in the composition are listed in the order in which they are called.

```
let gof = f >> g
```

This first applies `f` and then applies `g` on the result.

The `|>` Operator

The pipe forward operator, `|>`, is one of the most important operators in F#. The definition of the pipe forward operator is remarkably simple:

```
let inline (|>) x f = f x
```

Let's take 3 functions:

```
let square x = x * x
let add x y = x + y
let toString x = x.ToString()
```

Let's also say we had a complicated function that squared a number, added five to it, and converted it to a string? Normally, we'd write this:

```
let complexFunction x =
    toString (add 5 (square x))
```

We can improve the readability of this function somewhat using the pipe forward operator:

```
let complexFunction x =
    x |> square |> add 5 |> toString
```

`x` is piped to the `square` function, which is piped to `add 5` method, and finally to the `toString` method.

Anonymous Functions

Until now, all functions shown in this book have been named. For example, the function above is named `add`. F# allows programmers to declare nameless, or anonymous functions using the `fun` keyword.

```
let complexFunction =
    2                           (* 2 *)
    |> ( fun x -> x + 5)        (* 2 + 5 = 7 *)
    |> ( fun x -> x * x)        (* 7 * 7 = 49 *)
    |> ( fun x -> x.ToString() ) (* 49.ToString = "49" *)
```

Anonymous functions are convenient and find a use in a surprising number of places.

A Timer Function

```
open System

let duration f =
    let timer = new System.Diagnostics.Stopwatch()
    timer.Start()
    let returnValue = f()
    printfn "Elapsed Time: %i" timer.ElapsedMilliseconds
    returnValue

let rec fib = function
    | 0 -> 0
    | 1 -> 1
    | n -> fib (n - 1) + fib (n - 2)

let main() =
    printfn "fib 5: %i" (duration ( fun() -> fib 5 ))
    printfn "fib 30: %i" (duration ( fun() -> fib 30 ))

main()
```

The `duration` function has the type `val duration : (unit -> 'a) -> 'a`. This program prints:

```
Elapsed Time: 1
fib 5: 5
Elapsed Time: 5
fib 30: 832040
```

> *Note*: the actual duration to execute these functions vary from machine to machine.

Currying and Partial Functions

A fascinating feature in F# is called "currying", which means that F# does not require programmers to provide all of the arguments when calling a function. For example, let's say we have a function:

```
let add x y = x + y
```

`add` takes two integers and returns another integer. In F# notation, this is written as `val add : int -> int -> int`

We can define another function as follows:

```
let addFive = add 5
```

The `addFive` function calls the `add` function with one of its parameters, so what is the return value of this function? That's easy: `addFive` returns another function, which is waiting for the rest of its arguments. In this case, addFive returns a function that takes an `int` and returns another `int`, denoted in F# notation as `val addFive : (int -> int)`.

You call `addFive` just in the same way that you call other functions:

```
open System

let add x y = x + y

let addFive = add 5

Console.WriteLine(addFive 12) // prints 17
```

How Currying Works

The function `let add x y = x + y` has the type `val add : int -> int -> int`. F# uses the slightly unconventional arrow notation to denote function signatures for a reason: arrows notation is intrinsically connected to currying and anonymous functions. Currying works because, behind the scenes, F# converts function parameters to a style that looks like this:

```
let add = (fun x -> (fun y -> x + y) )
```

The type `int -> int -> int` is semantically equivalent to `(int -> (int -> int))`.

When you call `add` with no arguments, it returns `fun x -> fun y -> x + y` (or equivalently `fun x y -> x + y`), another function waiting for the rest of its arguments. Likewise, when you supply one argument to the function above, say 5, it returns `fun y -> 5 + y`, another function waiting for the rest of its arguments, with all occurrences of x being replaced by the argument 5.

Currying is built on the principle that each argument actually returns a separate function, which is why calling a function with only part of its parameters returns another function. The familiar F# syntax that we've seen so far, `let add x y = x + y`, is actually a kind of syntactic sugar for the explicit currying style shown above.

Two Pattern Matching Syntaxes

You may have wondered why there are two pattern matching syntaxes:

Traditional Syntax

```
let getPrice food =
    match food with
    | "banana" -> 0.79
    | "watermelon" -> 3.49
    | "tofu" -> 1.09
    | _ -> nan
```

Shortcut Syntax

```
let getPrice2 = function
    | "banana" -> 0.79
    | "watermelon" -> 3.49
    | "tofu" -> 1.09
    | _ -> nan
```

Both snippets of code are identical, but why does the shortcut syntax allow programmers to omit the `food` parameter in the function definition? The answer is related to currying: behind the scenes, the F# compiler converts the `function` keyword into the following construct:

```
let getPrice2 =
    (fun x ->
        match x with
        | "banana" -> 0.79
        | "watermelon" -> 3.49
        | "tofu" -> 1.09
        | _ -> nan)
```

In other words, F# treats the `function` keyword as an anonymous function that takes one parameter and returns one value. The `getPrice2` function actually returns an anonymous function; arguments passed to `getPrice2` are actually applied and evaluated by the anonymous function instead.

Option Types

F# : Option Types

An **option type** can hold two possible values: `Some(x)` or `None`. Option types are frequently used to represent optional values in calculations, or to indicate whether a particular computation has succeeded or failed.

Using Option Types

Let's say we have a function that divides two integers. Normally, we'd write the function as follows:

```
let div x y = x / y
```

This function works just fine, but it's not safe: it's possible to pass an invalid value into this function which results in a runtime error. Here is a demonstration in fsi:

```
> let div x y = x / y;;

val div : int -> int -> int

> div 10 5;;
val it : int = 2

> div 10 0;;
System.DivideByZeroException: Attempted to divide by zero.
   at <StartupCode$FSI_0035>.$FSI_0035._main()
stopped due to error
```

`div 10 5` executes just fine, but `div 10 0` throws a division by zero exception.

Using option types, we can return `Some(value)` on a successful calculation, or `None` if the calculation fails:

```
> let safediv x y =
    match y with
    | 0 -> None
    | _ -> Some(x/y);;

val safediv : int -> int -> int option

> safediv 10 5;;
val it : int option = Some 2

> safediv 10 0;;
val it : int option = None
```

Notice an important difference between our `div` and `safediv` functions:

```
val div : int -> int -> int
val safediv : int -> int -> int option
```

`div` returns an `int`, while `safediv` returns an `int option`. Since our `safediv` function returns a different data type, it informs clients of our function that the application has entered an invalid state.

Option types are conceptually similar to nullable types in languages like C#, however F# option types do not use the CLR `System.Nullable<T>` representation in IL due to differences in semantics.

Pattern Matching Option Types

Pattern matching option types is as easy as creating them: the same syntax used to declare an option type is used to match option types:

```
> let isFortyTwo = function
    | Some(42) -> true
    | Some(_) | None -> false;;

val isFortyTwo : int option -> bool

> isFortyTwo (Some(43));;
val it : bool = false

> isFortyTwo (Some(42));;
val it : bool = true

> isFortyTwo None;;
val it : bool = false
```

Other Functions in the `Option` Module

`val get : 'a option -> 'a`

Returns the value of a `Some` option.

`val isNone : 'a option -> bool`

Returns `true` for a `None` option, `false` otherwise.

`val isSome : 'a option -> bool`

Returns `true` for a `Some` option, `false` otherwise.

`val map : ('a -> 'b) -> 'a option -> 'b option`

Given `None`, returns `None`. Given `Some(x)`, returns `Some(f x)`, where `f` is the given mapping function.

`val iter : ('a -> unit) -> 'a option -> unit`

Applies the given function to the value of a `Some` option, does nothing otherwise.

Tuples and Records

F# : Tuples and Records

Defining Tuples

A tuple is defined as a comma separated collection of values. For example, `(10, "hello")` is a 2-tuple with the type `(int * string)`. Tuples are extremely useful for creating ad hoc data structures which group together related values. Note that the parentheses are not part of the tuple but it is often necessary to add them to ensure that the tuple only includes what you think it includes.

```
let average (a, b) =
    (a + b) / 2.0
```

This function has the type `float * float -> float`, it takes a `float * float` tuple and returns another `float`.

```
> let average (a, b) =
    let sum = a + b
    sum / 2.0;;

val average : float * float -> float

> average (10.0, 20.0);;
val it : float = 15.0
```

Notice that a tuple is considered a single argument. As a result, tuples can be used to return multiple values:

Example 1 - a function which multiplies a 3-tuple by a scalar value to return another 3-tuple.

```
> let scalarMultiply (s : float) (a, b, c) = (a * s, b * s, c * s);;

val scalarMultiply : float -> float * float * float -> float * float * float

> scalarMultiply 5.0 (6.0, 10.0, 20.0);;
val it : float * float * float = (30.0, 50.0, 100.0)
```

Example 2 - a function which reverses the input of whatever is passed into the function.

```
> let swap (a, b) = (b, a);;
val swap : 'a * 'b -> 'b * 'a

> swap ("Web", 2.0);;
val it : float * string = (2.0, "Web")

> swap (20, 30);;
val it : int * int = (30, 20)
```

Example 3 - a function which divides two numbers and returns the remainder simultaneously.

```
> let divrem x y =
    match y with
    | 0 -> None
    | _ -> Some(x / y, x % y);;

val divrem : int -> int -> (int * int) option

> divrem 100 20;; (* 100 / 20 = 5 remainder 0 *)
val it : (int * int) option = Some (5, 0)

> divrem 6 4;; (* 6 / 4 = 1 remainder 2 *)
val it : (int * int) option = Some (1, 2)

> divrem 7 0;; (* 7 / 0 throws a DivisionByZero exception *)
val it : (int * int) option = None
```

Every tuple has a property called **arity**, which is the number of arguments used to define a tuple. For example, an `int * string` tuple is made up of two parts, so it has an arity of 2, a `string * string * float` has an arity of 3, and so on.

Pattern Matching Tuples

Pattern matching on tuples is easy, because the same syntax used to declare tuple types is also used to match tuples.

Example 1

Let's say that we have a function `greeting` that prints out a custom greeting based on the specified name and/or language.

```
let greeting (name, language) =
    match (name, language) with
    | ("Steve", _) -> "Howdy, Steve"
    | (name, "English") -> "Hello, " + name
    | (name, _) when language.StartsWith("Span") -> "Hola, " + name
    | (_, "French") -> "Bonjour!"
    | _ -> "DOES NOT COMPUTE"
```

This function has type `string * string -> string`, meaning that it takes a 2-tuple and returns a string. We can test this function in fsi:

```
> greeting ("Steve", "English");;
val it : string = "Howdy, Steve"
> greeting ("Pierre", "French");;
val it : string = "Bonjour!"
> greeting ("Maria", "Spanish");;
val it : string = "Hola, Maria"
> greeting ("Rocko", "Esperanto");;
val it : string = "DOES NOT COMPUTE"
```

Example 2

We can conveniently match against the *shape* of a tuple using the alternative pattern matching syntax:

```
> let getLocation = function
    | (0, 0) -> "origin"
    | (0, y) -> "on the y-axis at y=" + y.ToString()
    | (x, 0) -> "on the x-axis at x=" + x.ToString()
    | (x, y) -> "at x=" + x.ToString() + ", y=" + y.ToString() ;;

val getLocation : int * int -> string

> getLocation (0, 0);;
val it : string = "origin"
> getLocation (0, -1);;
val it : string = "on the y-axis at y=-1"
> getLocation (5, -10);;
val it : string = "at x=5, y=-10"
> getLocation (7, 0);;
val it : string = "on the x-axis at x=7"
```

`fst` and `snd`

F# has two built-in functions, `fst` and `snd`, which return the first and second items in a 2-tuple. These functions are defined as follows:

```
let fst (a, b) = a
let snd (a, b) = b
```

They have the following types:

```
val fst : 'a * 'b -> 'a
val snd : 'a * 'b -> 'b
```

Here are a few examples in FSI:

```
> fst (1, 10);;
val it : int = 1
> snd (1, 10);;
val it : int = 10
> fst ("hello", "world");;
val it : string = "hello"
> snd ("hello", "world");;
val it : string = "world"
> fst ("Web", 2.0);;
val it : string = "Web"
> snd (50, 100);;
val it : int = 100
```

Assigning Multiple Variables Simultaneously

Tuples can be used to assign multiple values simultaneously. This is the same as tuple unpacking in Python. The syntax for doing so is:

```
let val1, val2, ... valN = (expr1, expr2, ... exprN)
```

In other words, you assign a comma-separated list of *N* values to an *N*-tuple. Here's an example in FSI:

```
> let x, y = (1, 2);;

val y : int
val x : int

> x;;
```

```
val it : int = 1

> y;;
val it : int = 2
```

The number of values being assigned must match the arity of tuple returned from the function, otherwise F# will raise an exception:

```
> let x, y = (1, 2, 3);;

  let x, y = (1, 2, 3);;
  ------------^^^^^^^^

stdin(18,13): error FS0001: Type mismatch. Expecting a
    'a * 'b
but given a
    'a * 'b * 'c.
The tuples have differing lengths of 2 and 3.
```

Tuples and the .NET Framework

From a point of view F#, all methods in the .NET Base Class Library take a single argument, which is a tuple of varying types and arity. For example:

Class	C# Function Signature	F# Function Signature
`System.String`	`String Join(String separator, String[] value)`	`val Join : (string * string array) -> string`
`System.Net.WebClient`	`void DownloadFile(String uri, String fileName)`	`val DownloadFile : (string * string) -> unit`
`System.Convert`	`String ToString(int value, int toBase)`	`val ToString : (int * int) -> string`
`System.Math`	`int DivRem(int a, int b, out int remainder)`	`val DivRem : (int * int) -> (int * int)`
`System.Int32`	`bool TryParse(String value, out int result)`	`val TryParse : string -> (bool * int)`

Some methods, such as the `System.Math.DivRem` shown above, and others such as `System.Int32.TryParse` return multiple through *output variables*. F# allows programmers to omit an output variable; using this calling convention, F# will return results of a function as a tuple, for example:

```
> System.Int32.TryParse("3");;
val it : bool * int = (true, 3)

> System.Math.DivRem(10, 7);;
val it : int * int = (1, 3)
```

Defining Records

A record is similar to a tuple, except it contains named fields. A record is defined using the syntax:

```
type recordName =
    { [ fieldName : dataType ] + }
```

> + means the element must occur one or more times.

Here's a simple record:

```
type website =
    { Title : string;
      Url : string }
```

Unlike a tuple, a record is explicitly defined as its own type using the `type` keyword, and record fields are defined as a semicolon-separated list. (In many ways, a record can be thought of as a simple class.)

A `website` record is created by specifying the record's fields as follows:

```
> let homepage = { Title = "Google"; Url = "http://www.google.com" };;
val homepage : website
```

Note that F# determines a records type by the name and type of its fields, not the order that fields are used. For example, while the record above is defined with `Title` first and `Url` second, it's perfectly legitimate to write:

```
> { Url = "http://www.microsoft.com/"; Title = "Microsoft Corporation" };;
val it : website = {Title = "Microsoft Corporation";
                    Url = "http://www.microsoft.com/";}
```

It's easy to access a record's properties using dot notation:

```
> let homepage = { Title = "Wikibooks"; Url = "http://www.wikibooks.org/" };;

val homepage : website

> homepage.Title;;
val it : string = "Wikibooks"

> homepage.Url;;
val it : string = "http://www.wikibooks.org/"
```

Cloning Records

Records are immutable types, which means that instances of records cannot be modified. However, records can be cloned conveniently using the clone syntax:

```
type coords = { X : float; Y : float }

let setX item newX =
    { item with X = newX }
```

The method `setX` has the type `coords -> float -> coords`. The `with` keyword creates a clone of `item` and set its `X` property to `newX`.

```
> let start = { X = 1.0; Y = 2.0 };;
val start : coords

> let finish = setX start 15.5;;
val finish : coords

> start;;
val it : coords = {X = 1.0;
                   Y = 2.0;}
> finish;;
val it : coords = {X = 15.5;
                   Y = 2.0;}
```

Notice that the `setX` creates a copy of the record, it doesn't actually mutate the original record instance.

Here's a more complete program:

```
type TransactionItem =
    { Name : string;
        ID : int;
        ProcessedText : string;
        IsProcessed : bool }

let getItem name id =
    { Name = name; ID = id; ProcessedText = null; IsProcessed = false }

let processItem item =
    { item with
        ProcessedText = "Done";
        IsProcessed = true }

let printItem msg item =
    printfn "%s: %A" msg item

let main() =
    let preProcessedItem = getItem "Steve" 5
    let postProcessedItem = processItem preProcessedItem

    printItem "preProcessed" preProcessedItem
    printItem "postProcessed" postProcessedItem

main()
```

This program processes an instance of the `TransactionItem` class and prints the results. This program outputs the following:

```
preProcessed: {Name = "Steve";
 ID = 5;
 ProcessedText = null;
 IsProcessed = false;}
postProcessed: {Name = "Steve";
 ID = 5;
 ProcessedText = "Done";
 IsProcessed = true;}
```

Pattern Matching Records

We can pattern match on records just as easily as tuples:

```
open System

type coords = { X : float; Y : float }

let getQuadrant = function
    | { X = 0.0; Y = 0.0 } -> "Origin"
    | item when item.X >= 0.0 && item.Y >= 0.0 -> "I"
    | item when item.X <= 0.0 && item.Y >= 0.0 -> "II"
    | item when item.X <= 0.0 && item.Y <= 0.0 -> "III"
    | item when item.X >= 0.0 && item.Y <= 0.0 -> "IV"

let testCoords (x, y) =
    let item = { X = x; Y = y }
    printfn "(%f, %f) is in quadrant %s" x y (getQuadrant item)

let main() =
    testCoords(0.0, 0.0)
    testCoords(1.0, 1.0)
    testCoords(-1.0, 1.0)
    testCoords(-1.0, -1.0)
    testCoords(1.0, -1.0)
    Console.ReadKey(true) |> ignore

main()
```

Note that pattern cases are defined with the same syntax used to create a record (as shown in the first case), or using guards (as shown in the remaining cases). Unfortunately, programmers cannot use the clone syntax in pattern cases, so a case such as `| { item with X = 0 } -> "y-axis"` will not compile.

The program above outputs:

```
(0.000000, 0.000000) is in quadrant Origin
(1.000000, 1.000000) is in quadrant I
(-1.000000, 1.000000) is in quadrant II
(-1.000000, -1.000000) is in quadrant III
(1.000000, -1.000000) is in quadrant IV
```

Lists

F# : Lists

A **list** is an ordered collection of related values, and is roughly equivalent to a linked list data structure used in many other languages. F# provides a module, `Microsoft.FSharp.Collections.List`, for common operations on lists; this module is imported automatically by F#, so the `List` module is already accessible from every F# application.

Creating Lists

Using List Literals

There are a variety of ways to create lists in F#, the most straightforward method being a semicolon-delimited sequence of values. Here's a list of numbers in fsi:

```
> let numbers = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
val numbers : int list

> numbers;;
val it : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Notice that all values in a list must have the same type:

```
> [1; 2; 3; 4; "cat"; 6; 7];;

  ------------^^^^^^

stdin(120,14): error FS0001: This expression has type
        string
but is here used with type
        int.
```

Using the `::` ("cons") Operator

It is very common to build lists up by prepending or consing a value to an existing list using the `::` operator:

```
> 1 :: 2 :: 3 :: [];;
val it : int list = [1; 2; 3]
```

> *Note*: the `[]` is an empty list. By itself, it has the type `'T list`; since it is used with `int`s, it has the type `int list`.

The `::` operator prepends items to a list, returning a new list. It is a right-associative operator with the following type:

```
val inline (::) : 'T -> 'T list -> 'T list
```

This operator does not actually mutate lists, it creates an entirely new list with the prepended element in the front. Here's an example in fsi:

```
> let x = 1 :: 2 :: 3 :: 4 :: [];;
val x : int list

> let y = 12 :: x;;
val y : int list

> x;;
val it : int list = [1; 2; 3; 4]

> y;;
val it : int list = [12; 1; 2; 3; 4]
```

Consing creates a new list, but it reuses nodes from the old list, so consing a list is an extremely efficient O(1) operation.

Using List.init

The `List` module contains a useful method, `List.init`, which has the type

```
val init : int -> (int -> 'T) -> 'T list
```

The first argument is the desired length of the new list, and the second argument is an initializer function which generates items in the list. `List.init` is used as follows:

```
> List.init 5 (fun index -> index * 3);;
val it : int list = [0; 3; 6; 9; 12]

> List.init 5 (fun index -> (index, index * index, index * index * index));;
val it : (int * int * int) list
= [(0, 0, 0); (1, 1, 1); (2, 4, 8); (3, 9, 27); (4, 16, 64)]
```

F# calls the initializer function `5` times with the index of each item in the list, starting at index `0`.

Using List Comprehensions

List comprehensions refers to special syntactic constructs in some languages used for generating lists. F# has an expressive list comprehension syntax, which comes in two forms, ranges and generators.

Ranges have the constructs `[start .. end]` and `[start .. step .. end]`. For example:

```
> [1 .. 10];;
val it : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

> [1 .. 2 .. 10];;
val it : int list = [1; 3; 5; 7; 9]

> ['a' .. 's'];;
val it : char list
= ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm'; 'n'; 'o';
   'p'; 'q'; 'r'; 's']
```

Generators have the construct `[for x in collection do ... yield expr]`, and they are much more flexible than ranges. For example:

```
> [ for a in 1 .. 10 do
        yield (a * a) ];;
val it : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]

> [ for a in 1 .. 3 do
    for b in 3 .. 7 do
        yield (a, b) ];;
val it : (int * int) list
= [(1, 3); (1, 4); (1, 5); (1, 6); (1, 7); (2, 3); (2, 4); (2, 5); (2, 6);
   (2, 7); (3, 3); (3, 4); (3, 5); (3, 6); (3, 7)]

> [ for a in 1 .. 100 do
    if a % 3 = 0 && a % 5 = 0 then yield a];;
val it : int list = [15; 30; 45; 60; 75; 90]
```

it's possible to loop over any collection, not just numbers. This example loops over a `char list`:

```
> let x = [ 'a' .. 'f' ];;

val x : char list

> [for a in x do yield [a; a; a] ];;
val it : char list list
= [['a'; 'a'; 'a']; ['b'; 'b'; 'b']; ['c'; 'c'; 'c']; ['d'; 'd'; 'd'];
   ['e'; 'e'; 'e']; ['f'; 'f'; 'f']]
```

Note that the `yield` keyword pushes a single value into a list. Another keyword, `yield!`, pushes a collection of values into the list. The `yield!` keyword is used as follows:

```
> [for a in 1 .. 5 do
    yield! [ a .. a + 3 ] ];;
val it : int list
= [1; 2; 3; 4; 2; 3; 4; 5; 3; 4; 5; 6; 4; 5; 6; 7; 5; 6; 7; 8]
```

It's possible to mix the `yield` and `yield!` keywords:

```
> [for a in 1 .. 5 do
    match a with
    | 3 -> yield! ["hello"; "world"]
    | _ -> yield a.ToString() ];;
val it : string list = ["1"; "2"; "hello"; "world"; "4"; "5"]
```

Alternative List Comprehension Syntax

The samples above use the `yield` keyword explicitly, however F# provides a slightly different arrow-based syntax for list comprehensions:

```
> [ for a in 1 .. 5 -> a * a];;
val it : int list = [1; 4; 9; 16; 25]

> [ for a in 1 .. 5 do
    for b in 1 .. 3 -> a, b];;
val it : (int * int) list
= [(1, 1); (1, 2); (1, 3); (2, 1); (2, 2); (2, 3); (3, 1); (3, 2); (3, 3);
   (4, 1); (4, 2); (4, 3); (5, 1); (5, 2); (5, 3)]
```

`->` is equivalent to the `yield` operator respectively. While it's still common to see list comprehensions expressed using `->`, this construct will not be emphasized in this book since it has been deprecated in favor of `yield`.

Pattern Matching Lists

You use the same syntax to match against lists that you use to create lists. Here's a simple program:

```
let rec sum total = function
    | [] -> total
    | hd :: tl -> sum (hd + total) tl

let main() =
    let numbers = [ 1 .. 5 ]
    let sumOfNumbers = sum 0 numbers
    printfn "sumOfNumbers: %i" sumOfNumbers

main()
```

The `sum` method has the type `val sum : int -> int list -> int`. It recursively enumerates through the list, adding each item in the list to the value `total`. Step by step, the function works as follows:

total	input	hd :: tl	sum (hd + total) tl
0	[1; 2; 3; 4; 5]	1 :: [2; 3; 4; 5]	sum (1 + 0 = 1) [2; 3; 4; 5]
1	[2; 3; 4; 5]	2 :: [3; 4; 5]	sum (2 + 1 = 3) [3; 4; 5]
3	[3; 4; 5]	3 :: [4; 5]	sum (3 + 3 = 6) [4; 5]
6	[4; 5]	4 :: [5]	sum (4 + 6 = 10) [5]
10	[5]	5 :: []	sum (5 + 10 = 15) []
15	[]	n/a	returns total

Reversing Lists

Frequently, we use recursion and pattern matching to generate new lists from existing lists. A simple example is reversing a list:

```
let reverse l =
    let rec loop acc = function
        | [] -> acc
        | hd :: tl -> loop (hd :: acc) tl
    loop [] l
```

> *Note to beginners:* the pattern seen above is very common. Often, when we iterate through lists, we want to build up a new list. To do this recursively, we use an *accumulating parameter* (which is called `acc` above) which holds our new list as we generate it. It's also very common to use a nested function, usually named `innerXXXXX` or `loop`, to hide the implementation details of the function from clients (in other words, clients should not have to pass in their own accumulating parameter).

`reverse` has the type `val reverse : 'a list -> 'a list`. You'd use this function as follows:

```
> reverse [1 .. 5];;
val it : int list = [5; 4; 3; 2; 1]
```

This simple function works because items are always prepended to the accumulating parameter `acc`, resulting in series of recursive calls as follows:

acc	input	loop (hd :: acc) tl
[]	[1; 2; 3; 4; 5]	loop (1 :: []) [2; 3; 4; 5]
[1]	[2; 3; 4; 5]	loop (2 :: [1]) [3; 4; 5]
[2; 1]	[3; 4; 5]	loop (3 :: [2; 1]) [4; 5]
[3; 2; 1]	[4; 5]	loop (4 :: [3; 2; 1]) [5]
[4; 3; 2; 1]	[5]	loop (5 :: [4; 3; 2; 1]) []
[5; 4; 3; 2; 1]	[]	returns acc

`List.rev` is a built-in function for reversing a list:

```
> List.rev [1 .. 5];;
val it : int list = [5; 4; 3; 2; 1]
```

Filtering Lists

Oftentimes, we want to filter a list for certain values. We can write a filter function as follows:

```
open System

let rec filter predicate = function
    | [] -> []
    | hd :: tl ->
        match predicate hd with
```

```
        | true -> hd::filter predicate tl
        | false -> filter predicate tl

let main() =
    let filteredNumbers = [1 .. 10] |> filter (fun x -> x % 2 = 0)
    printfn "filteredNumbers: %A" filteredNumbers

main()
```

The `filter` method has the type `val filter : ('a -> bool) -> 'a list -> 'a list`. The program above outputs:

```
filteredNumbers: [2; 4; 6; 8; 10]
```

We can make the `filter` above tail-recursive with a slight modification:

```
let filter predicate l =
    let rec loop acc = function
        | [] -> acc
        | hd :: tl ->
            match predicate hd with
            | true -> loop (hd :: acc) tl
            | false -> loop (acc) tl
    List.rev (loop [] l)
```

> *Note:* Since accumulating parameters often build up lists in reverse order, it's very common to see `List.rev` called immediately before returning a list from a function to put it in correct order.

Mapping Lists

We can write a function which maps a list to another list:

```
open System

let rec map converter = function
    | [] -> []
    | hd :: tl -> converter hd::map converter tl

let main() =
    let mappedNumbers = [1 .. 10] |> map ( fun x -> (x * x).ToString() )
    printfn "mappedNumbers: %A" mappedNumbers

main()
```

`map` has the type `val map : ('a -> 'b) -> 'a list -> 'b list`. The program above outputs:

```
mappedNumbers: ["1"; "4"; "9"; "16"; "25"; "36"; "49"; "64"; "81"; "100"]
```

A tail-recursive map function can be written as:

```
let map converter l =
    let rec loop acc = function
        | [] -> acc
        | hd :: tl -> loop (converter hd :: acc) tl
    List.rev (loop [] l)
```

Like the example above, we use the accumulating-param-and-reverse pattern to make the function tail recursive.

Using the `List` Module

Although a reverse, filter, and map method were implemented above, it's much more convenient to use F#'s built-in functions:

`List.rev` reverses a list:

```
> List.rev [1 .. 5];;
val it : int list = [5; 4; 3; 2; 1]
```

`List.filter` filters a list:

```
> [1 .. 10] |> List.filter (fun x -> x % 2 = 0);;
val it : int list = [2; 4; 6; 8; 10]
```

`List.map` maps a list from one type to another:

```
> [1 .. 10] |> List.map (fun x -> (x * x).ToString());;
val it : string list
= ["1"; "4"; "9"; "16"; "25"; "36"; "49"; "64"; "81"; "100"]
```

`List.append` and the `@` Operator

`List.append` has the type:

```
val append : 'T list -> 'T list -> 'T list
```

As you can imagine, the `append` functions appends one list to another. The `@` operator is an infix operator which performs the same function:

```
let first = [1; 2; 3;]
let second = [4; 5; 6;]
let combined1 = first @ second (* returns [1; 2; 3; 4; 5; 6] *)
let combined2 = List.append first second (* returns [1; 2; 3; 4; 5; 6] *)
```

Since lists are immutable, appending two lists together requires copying all of the elements of the lists to create a brand new list. However, since lists are immutable, it's only necessary to copy the elements of the first list; the second list does not need to be copied. Represented in memory, appending two lists can be diagrammed as follows:

We start with the following:

```
    first = 1 :: 2 :: 3 :: []

    second = 4 :: 5 :: 6 :: []
```

Appending the two lists, `first @ second`, results in the following:

```
       first = 1 :: 2 :: 3 :: []
               \_____  _____/
                     \/
    combined = 1 :: 2 :: 3 :: second
                    (copied)
```

In other words, F# prepends a copy of `first` to `second` to create the `combined` list. This hypothesis can be verified using the following in fsi:

```
> let first = [1; 2; 3;]
let second = [4; 5; 6;]
let combined = first @ second
let secondHalf = List.tail (List.tail (List.tail combined));;

val first : int list
val second : int list
val combined : int list
val secondHalf : int list

> System.Object.ReferenceEquals(second, secondHalf);;
val it : bool = true
```

The two lists `second` and `secondHalf` are literally the same object in memory, meaning F# reused the nodes from `second` when constructing the new list `combined`.

Appending two lists, `list1` and `list2` has a space and time complexity of O(list1.Length).

`List.choose`

`List.choose` has the following definition:

```
val choose : ('T -> 'U option) -> 'T list -> 'U list
```

The `choose` method is clever because it filters and maps a list simultaneously:

```
> [1 .. 10] |> List.choose (fun x ->
    match x % 2 with
    | 0 -> Some(x, x*x, x*x*x)
    | _ -> None);;
val it : (int * int * int) list
= [(2, 4, 8); (4, 16, 64); (6, 36, 216); (8, 64, 512); (10, 100, 1000)]
```

`choose` filters for items that return `Some` and maps them to another value in a single step.

`List.fold` and `List.foldBack`

`List.fold` and `List.foldBack` have the following definitions:

```
val fold : ('State -> 'T -> 'State) -> 'State -> 'T list -> 'State
val foldBack : ('T -> 'State -> 'State) -> 'T list -> 'State -> 'State
```

A "fold" operation applies a function to each element in a list, aggregates the result of the function in an accumulator variable, and returns the accumulator as the result of the fold operation. The 'State type represents the accumulated value, it is the output of one round of the calculation and input for the next round. This description makes fold operations sound more complicated, but the implementation is actually very simple:

```
(* List.fold implementation *)
let rec fold (f : 'State -> 'T -> 'State) (seed : 'State) = function
    | [] -> seed
    | hd :: tl -> fold f (f seed hd) tl

(* List.foldBack implementation *)
```

```
let rec foldBack (f : 'T -> 'State -> 'State) (items : 'T list) (seed : 'State) =
    match items with
    | [] -> seed
    | hd :: tl -> f hd (foldBack f tl seed)
```

`fold` applies a function to each element in the list from left to right, while `foldBack` applies a function to each element from right to left. Let's examine the fold functions in more technical detail using the following example:

```
let input = [ 2; 4; 6; 8; 10 ]
let f accumulator input = accumulator * input
let seed = 1
let output = List.fold f seed input
```

The value of `output` is `3840`. This table demonstrates how `output` was calculated:

accumulator	input	f accumulator input = accumulator * input
1 (seed)	2	1 * 2 = 2
2	4	2 * 4 = 8
8	6	8 * 6 = 48
48	8	48 * 8 = 384
384	10	384 * 10 = 3840 (return value)

`List.fold` passes an accumulator with an item from the list into a function. The output of the function is passed as the accumulator for the next item.

As shown above, the `fold` function processes the list from the first item to the last item in the list, or left to right. As you can imagine, `List.foldBack` works the same way, but it operates on lists from right to left. Given a fold function `f` and a list `[1; 2; 3; 4; 5]`, the fold methods transform our lists in the following ways:

```
fold: f (f (f (f (f seed 1) 2) 3) 4) 5
foldBack: f 1 (f 2 (f 3(f 4(f 5 seed))))
```

There are several other functions in the `List` module related to folding:

- `fold2` and `foldBack2`: folds two lists together simultaneously.
- `reduce` and `reduceBack`: same as `fold` and `foldBack`, except it uses the first (or last) element in the list as the seed value.
- `scan` and `scanBack`: similar to `fold` and `foldBack`, except it returns all of the intermediate values as a list rather than the final accumulated value.

Fold functions can be surprisingly useful:

Summing the numbers 1 - 100

```
let x = [ 1 .. 100 ] |> List.fold ( + ) 0 (* returns 5050 *)
```

In F#, mathematical operators are no different from functions. As shown above, we can actually pass the addition operator to the `fold` function, because the `+` operator has the definition `int -> int -> int`.

Computing a factorial

```
let factorial n = [ 1I .. n ] |> List.fold ( * ) 1I
let x = factorial 13I (* returns 6227020800I *)
```

Computing population standard deviation

```
let stddev (input : float list) =
    let sampleSize = float input.Length
    let mean = (input |> List.fold ( + ) 0.0) / sampleSize
    let differenceOfSquares =
        input |> List.fold
            ( fun sum item -> sum + Math.Pow(item - mean, 2.0) ) 0.0
    let variance = differenceOfSquares / sampleSize
    Math.Sqrt(variance)

let x = stddev [ 5.0; 6.0; 8.0; 9.0 ] (* returns 1.58113883 *)
```

`List.find` and `List.tryFind`

`List.find` and `List.tryfind` have the following types:

```
val find : ('T -> bool) -> 'T list -> 'T
val tryFind : ('T -> bool) -> 'T list -> 'T option
```

The `find` and `tryFind` methods return the first item in the list for which the search function returns `true`. They only differ as follows: if no items are found that meet the search function, `find` throws a `KeyNotFoundException`, while `tryfind` returns `None`.

The two functions are used as follows:

```
> let cities = ["Bellevue"; "Omaha"; "Lincoln"; "Papillion"; "Fremont"];;

val cities : string list =
  ["Bellevue"; "Omaha"; "Lincoln"; "Papillion"; "Fremont"]

> let findStringContaining text (items : string list) =
    items |> List.find(fun item -> item.Contains(text));;

val findStringContaining : string -> string list -> string

> let findStringContaining2 text (items : string list) =
    items |> List.tryFind(fun item -> item.Contains(text));;

val findStringContaining2 : string -> string list -> string option

> findStringContaining "Papi" cities;;
val it : string = "Papillion"

> findStringContaining "Hastings" cities;;
System.Collections.Generic.KeyNotFoundException: The given key was not present in the dictionary.
   at Microsoft.FSharp.Collections.ListModule.find[T](FastFunc`2 predicate, FSharpList`1 list)
   at <StartupCode$FSI_0007>.$FSI_0007.main@()
stopped due to error

> findStringContaining2 "Hastings" cities;;
val it : string option = None
```

Exercises

Solutions.

Pair and Unpair

Write two functions with the following definitions:

```
val pair : 'a list -> ('a * 'a) list
val unpair : ('a * 'a) list -> 'a list
```

The `pair` function should convert a list into a list of pairs as follows:

```
pair [ 1 .. 10 ] = [(1, 2); (3, 4); (5, 6); (7, 8); (9, 10)]
pair [ "one"; "two"; "three"; "four"; "five" ] = [("one", "two"); ("three", "four")]
```

The `unpair` function should convert a list of pairs back into a traditional list as follows:

```
unpair [(1, 2); (3, 4); (5, 6)] = [1; 2; 3; 4; 5; 6]
unpair [("one", "two"); ("three", "four")] = ["one"; "two"; "three"; "four"]
```

Expand a List

Write a function with the following type definition:

```
val expand : 'a list -> 'a list list
```

The `expand` function should expand a list as follows:

```
expand [ 1 .. 5 ] = [ [1; 2; 3; 4; 5]; [2; 3; 4; 5]; [3; 4; 5]; [4; 5]; [5] ]
expand [ "monkey"; "kitty"; "bunny"; "rat" ] =
    [ ["monkey"; "kitty"; "bunny"; "rat"];
      ["kitty"; "bunny"; "rat"];
      ["bunny"; "rat"];
      ["rat"] ]
```

Greatest common divisor on lists

The task is to calculate the greatest common divisor of a list of integers.
The first step is to write a function which should have the following type:

```
val gcd : int -> int -> int
```

The `gcd` function should take two integers and return their greatest common divisor. Hint: Use Euler's algorithm

```
gcd 15 25 = 5
```

The second step is to use the gcd function to calculate the greatest common divisor of an int list.

```
val gcdl : int list -> int
```

The `gcdl` function should work like this:

```
gcdl [15; 75; 20] = 5
```

If the list is empty the result shall be 0.

Basic Mergesort Algorithm

The goal of this exercise is to implement the mergesort algorithm to sort a list in F#. When I talk about sorting, it means sorting in an ascending order. If you're not familiar with the mergesort algorithm, it works like this:

- Split: Divide the list in two equally large parts
- Sort: Sort those parts
- Merge: Sort while merging (hence the name)

Note that the algorithm works recursively. It will first split the list. The next step is to sort both parts. To do that, they will be split again and so on. This will basically continue until the original list is scrambled up completely. Then recursion will do it's magic and assemble the list from the bottom. This might seem confusing at first, but you will learn a lot about how recursion works, when there's more than one function involved

The `split` function

```
val split : 'a list -> 'a list * 'a list
```

The `split` function will work like this.

```
split [2; 8; 5; 3] = ( [5; 2], [8; 3] )
```

The split's will be returned as a tuple. The `split` function doesn't need to sort the lists though.

The `merge` function
The next step is merging. We now want to merge the split's together into a sorted list assuming that both split's themselves are already sorted. The `merge` function will take a tuple of two already sorted lists and recursively create a sorted list:

```
val merge : 'a list * 'a list -> 'a list
```

Example:

```
merge ( [2; 5], [3; 8] ) = [2; 3; 5; 8]
```

It is important to notice that the `merge` function will only work if both split's are already sorted. It will make it's implementation a lot easier. Assuming both split's are sorted, we can just look at the first element of both split's and only compare which one of them is smaller. To ensure this is the case we will write one last function.

The `msort` function
You can think of it as the function organising the correct execution of the algorithm. It uses the previously implemented functions, so it's able to take a random list and return the sorted list.

```
val msort : 'a list -> 'a list
```

How to implement msort:
If the list is empty or if the list has only one element, we don't need to do anything to it and can immediately return it, because we don't need to sort it.
If that's not the case, we need to apply our algorithm to it. First `split` the list into a tuple of two, then `merge` the tuple while recursively sorting both arguments of the tuple

Sequences

F# : Sequences

Sequences, commonly called **sequence expressions**, are similar to lists: both data structures are used to represent an ordered collection of values. However, unlike lists, elements in a sequence are computed *as they are needed* (or "lazily"), rather than computed all at once. This gives sequences some interesting properties, such as the capacity to represent infinite data structures.

Defining Sequences

Sequences are defined using the syntax:

```
seq { expr }
```

Similar to lists, sequences can be constructed using ranges and comprehensions:

```
> seq { 1 .. 10 };; (* seq ranges *)
val it : seq<int> = seq [1; 2; 3; 4; ...]

> seq { 1 .. 2 .. 10 };; (* seq ranges *)
val it : seq<int> = seq [1; 3; 5; 7; ...]

> seq {10 .. -1 .. 0};; (* descending *)
val it : seq<int> = seq [10; 9; 8; 7; ...]

> seq { for a in 1 .. 10 do yield a, a*a, a*a*a };; (* seq comprehensions *)
val it : seq<int * int * int>
= seq [(1, 1, 1); (2, 4, 8); (3, 9, 27); (4, 16, 64); ...]
```

Sequences have an interesting property which sets them apart from lists: elements in the sequence are *lazily evaluated*, meaning that F# does not compute values in a sequence until the values are actually needed. This is in contrast to lists, where F# computes the value of all elements in a list on declaration. As a demonstration, compare the following:

```
> let intList =
    [ for a in 1 .. 10 do
        printfn "intList: %i" a
        yield a ]

let intSeq =
    seq { for a in 1 .. 10 do
            printfn "intSeq: %i" a
            yield a };;

val intList : int list
val intSeq : seq<int>

intList: 1
intList: 2
intList: 3
intList: 4
intList: 5
intList: 6
intList: 7
intList: 8
intList: 9
intList: 10

> Seq.item 3 intSeq;;
intSeq: 1
intSeq: 2
intSeq: 3
intSeq: 4
val it : int = 4

> Seq.item 7 intSeq;;
intSeq: 1
intSeq: 2
intSeq: 3
intSeq: 4
intSeq: 5
intSeq: 6
intSeq: 7
intSeq: 8
val it : int = 8
```

The list is created on declaration, but elements in the sequence are created as they are needed.

As a result, sequences are able to represent a data structure with an arbitrary number of elements:

```
> seq { 1I .. 1000000000000I };;
val it : seq<bigint> = seq [1I; 2I; 3I; 4I; ...]
```

The sequence above represents a list with one trillion elements in it. That does not mean the sequence actually contains one trillion elements, but it can *potentially* hold one trillion elements. By comparison, it would not be possible to create a list `[ 1I .. 1000000000000I ]` since the .NET runtime would attempt to create all one trillion elements up front, which would certainly consume all of the available memory on a system before the operation completed.

Additionally, sequences can represent an infinite number of elements:

```
> let allEvens =
    let rec loop x = seq { yield x; yield! loop (x + 2) }
    loop 0;;

> for a in (Seq.take 5 allEvens) do
    printfn "%i" a;;
0
2
4
6
8
val it : unit = ()
```

Notice the definition of `allEvens` does not terminate. The function `Seq.take` returns the first `n` elements of elements of the sequence. If we attempted to loop through all of the elements, fsi would print indefinitely.

> **Note:** sequences are implemented as state machines by the F# compiler. In reality, they manage state internally and hold only the last generated item in memory at a time. Memory usage is constant for creating and traversing sequences of any length.

Iterating Through Sequences Manually

The .NET Base Class Library (BCL) contains two interfaces in the `System.Collections.Generic (http://msdn.microsoft.com/en-us/library/system.collections.generic.aspx)` namespace:

```
type IEnumerable<'a> =
  interface
   (* Returns an enumerator that iterates through a collection *)
    member GetEnumerator<'a> : unit -> IEnumerator<'a>
  end

type IEnumerator<'a> =
  interface
   (* Advances to the next element in the sequences. Returns true if
      the enumerator was successfully advanced to the next element; false
      if the enumerator has passed the end of the collection. *)
    member MoveNext : unit -> bool

    (* Gets the current element in the collection. *)
    member Current : 'a

    (* Sets the enumerator to its initial position, which is
       before the first element in the collection. *)
    member Reset : unit -> unit
  end
```

The `seq` type is defined as follows:

```
type seq<'a> = System.Collections.Generic.IEnumerable<'a>
```

As you can see, `seq` is not a unique F# type, but rather another name for the built-in `System.Collections.Generic.IEnumerable` interface. Since `seq`/`IEnumerable` is a native .NET type, it was designed to be used in a more imperative style, which can be demonstrated as follows:

```
open System
open System.Collections

let evens = seq { 0 .. 2 .. 10 } (* returns IEnumerable<int> *)

let main() =
    let evensEnumerator = evens.GetEnumerator() (* returns IEnumerator<int> *)
    while evensEnumerator.MoveNext() do
        printfn "evensEnumerator.Current: %i" evensEnumerator.Current

    Console.ReadKey(true) |> ignore

main()
```

This program outputs:

```
evensEnumerator.Current: 0
evensEnumerator.Current: 2
evensEnumerator.Current: 4
evensEnumerator.Current: 6
evensEnumerator.Current: 8
evensEnumerator.Current: 10
```

Behind the scenes, .NET converts every `for` loop over a collection into an explicit while loop. In other words, the following two pieces of code compile down to the same bytecode:

```
let x = [1 .. 10]
for num in x do
    printfn "%i" num
```

```
let x = [1 .. 10]
let enumerator = x.GetEnumerator()
while enumerator.MoveNext() do
    let num = enumerator.Current
    printfn "%i" num
```

All collections which can be used with the `for` keyword implement the `IEnumerable<'a>` interface, a concept which will be discussed later in this book.

The Seq Module

Similar to the List modules, the `Seq` module contains a number of useful functions for operating on sequences:

val append : seq<'T> -> seq<'T> -> seq<'T>

> Appends one sequence onto another sequence.

```
> let test = Seq.append (seq{1..3}) (seq{4..7});;
val it : seq<int> = seq [1; 2; 3; 4; ...]
```

val choose : ('T -> 'U option) -> seq<'T> -> seq<'U>

> Filters and maps a sequence to another sequence.

```
> let thisworks = seq { for nm in [ Some("James"); None; Some("John") ] |> Seq.choose id -> nm.Length }
val it : seq<int> = seq [5; 4]
```

val distinct : seq<'T> -> seq<'T>

Returns a sequence that filters out duplicate entries.

```
> let dist = Seq.distinct (seq[1;2;2;6;3;2])
val it : seq<int> = seq [1; 2; 6; 3]
```

val exists : ('T -> bool) -> seq<'T> -> bool

Determines if an element exists in a sequence.

```
> let equalsTwo x = x=2
> let exist = Seq.exists equalsTwo (seq{3..9})
val equalsTwo : int -> bool
val it : bool = false
```

val filter : ('T -> bool) -> seq<'T> -> seq<'T>

Builds a new sequence consisting of elements filtered from the input sequence.

```
> Seq.filter (fun x-> x%2 = 0) (seq{0..9})
val it : seq<int> = seq [0; 2; 4; 6; ...]
```

val fold : ('State -> 'T -> 'State) -> 'State -> seq<'T> -> 'State

Repeatedly applies a function to each element in the sequence from left to right.

```
> let sumSeq sequence1 = Seq.fold (fun acc elem -> acc + elem) 0 sequence1
Seq.init 10 (fun index -> index * index)
|> sumSeq
|> printfn "The sum of the elements is %d."
>
The sum of the elements is 285.

val sumSeq : seq<int> -> int
```

Note: sequences can only be read in a forward-only manner, so there is no corresponding `foldBack` function as found in the List and Array modules.

val initInfinite : (int -> 'T) -> seq<'T>

Generates a sequence consisting of an infinite number of elements.

```
> Seq.initInfinite (fun x -> x*x)
val it : seq<int> = seq [0; 1; 4; 9; ...]
```

val map : ('T -> 'U) -> seq<'T> -> seq<'U>

Maps a sequence of type `'a` to type `'b`.

```
> Seq.map (fun x->x*x+2) (seq[3;5;4;3])
val it : seq<int> = seq [11; 27; 18; 11]
```

val item : int -> seq<'T> -> 'T

Returns the *nth* value of a sequence.

```
> Seq.item 3 (seq {for n in 2..9 do yield n})
val it : int = 5
```

val take : int -> seq<'T> -> seq<'T>

Returns a new sequence consisting of the first *n* elements of the input sequence.

```
> Seq.take 3 (seq{1..6})
val it : seq<int> = seq [1; 2; 3]
```

val takeWhile : ('T -> bool) -> seq<'T> -> seq<'T>

Return a sequence that, when iterated, yields elements of the underlying sequence while the given predicate returns `true`, and returns no further elements.

```
> let sequenciaMenorqDez = Seq.takeWhile (fun elem -> elem < 10) (seq {for i in 0..20 do yield i+1})
val sequenciaMenorqDez : seq<int>
> sequenciaMenorqDez;;
val it : seq<int> = seq [1; 2; 3; 4; ...]
```

val unfold : ('State -> ('T * 'State) option) -> 'State seed -> seq<'T>

The opposite of `fold`: this function generates a sequence as long as the generator function returns `Some`.

```
> let fibs = (0I, 1I) |> Seq.unfold (fun (a, b) -> Some(a, (b, a + b) ) );;
```

```
val fibs : seq<bigint>

> Seq.iter (fun x -> printf "%O " x) (Seq.take 20 fibs);;
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

The generator function in `unfold` expects a return type of `('T * 'State) option`. The first value of the tuple is inserted as an element into the sequence, the second value of the tuple is passed as the accumulator. The `fibs` function is clever for its brevity, but it's hard to understand if you've never seen an unfold function. The following demonstrates `unfold` in a more straightforward way:

```
> let test = 1 |> Seq.unfold (fun x ->
    if x <= 5 then Some(sprintf "x: %i" x, x + 1)
    else None );;

val test : seq<string>

> Seq.iter (fun x -> printfn "%s" x) test;;
x: 1
x: 2
x: 3
x: 4
x: 5
```

Often, it's preferable to generate sequences using `seq` comprehensions rather than the `unfold`.

Sets and Maps

F# : Sets and Maps

In addition to lists and sequences, F# provides two related immutable data structures called **sets** and **maps**. Unlike lists, sets and maps are *unordered* data structures, meaning that these collections do not preserve the order of elements as they are inserted, nor do they permit duplicates.

Sets

A set in F# is just a container for items. Sets do not preserve the order in which items are inserted, nor do they allow duplicate entries to be inserted into the collection.

Sets can be created in a variety of ways:

Adding an item to an empty set The `Set` module contains a useful function `Set.empty` which returns an empty set to start with.

Conveniently, all instances of sets have an `Add` function with the type `val Add : 'a -> Set<'a>`. In other words, our `Add` returns a new set containing our new item, which makes it easy to add items together in this fashion:

```
> Set.empty.Add(1).Add(2).Add(7);;
val it : Set<int> = set [1; 2; 7]
```

Converting lists and sequences into sets Additionally, the we can use `Set.ofList` and `Set.ofSeq` to convert an entire collection into a set:

```
> Set.ofList ["Mercury"; "Venus"; "Earth"; "Mars"; "Jupiter"; "Saturn"; "Uranus"; "Neptune"];;
val it : Set<string> = set ["Earth"; "Jupiter"; "Mars"; "Mercury"; ...]
```

The example above demonstrates the unordered nature of sets.

The `Set` Module

The FSharp.Collections.Set (https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-setmodule.html) module contains a variety of useful methods for working with sets.

`val add : 'a -> Set<'a> -> Set<'a>`

Return a new set with an element added to the set. No exception is raised if the set already contains the given element.

`val compare : Set<'a> -> Set<'a> -> int`

Compare two sets. Places sets into a total order.

`val count : Set<'a> -> int`

Return the number of elements in the set. Same as "size".

`val difference : Set<'a> -> Set<'a> -> Set<'a>`

Return a new set with the elements of the second set removed from the first. That is a set containing only those items from the first set that are not also in the second set.

```
> let a = Set.ofSeq [ 1 .. 10 ]
let b = Set.ofSeq [ 5 .. 15 ];;

val a : Set<int>
val b : Set<int>

> Set.difference a b;;
val it : Set<int> = set [1; 2; 3; 4]

> a - b;; (* The '-' operator is equivalent to Set.difference *)
val it : Set<int> = set [1; 2; 3; 4]
```

`val exists : ('a -> bool) -> Set<'a> -> bool`

Test if any element of the collection satisfies the given predicate.

`val filter : ('a -> bool) -> Set<'a> -> Set<'a>`

Return a new collection containing only the elements of the collection for which the given predicate returns "true".

`val intersect : Set<'a> -> Set<'a> -> Set<'a>`

Compute the intersection, or overlap, of the two sets.

```
> let a = Set.ofSeq [ 1 .. 10 ]
let b = Set.ofSeq [ 5 .. 15 ];;

val a : Set<int>
val b : Set<int>

> Set.iter (fun x -> printf "%O " x) (Set.intersect a b);;
5 6 7 8 9 10
```

`val map : ('a -> 'b) -> Set<'a> -> Set<'b>`

Return a new collection containing the results of applying the given function to each element of the input set.

`val contains: 'a -> Set<'a> -> bool`

Evaluates to `true` if the given element is in the given set.

`val remove : 'a -> Set<'a> -> Set<'a>`

Return a new set with the given element removed. No exception is raised if the set doesn't contain the given element.

`val count: Set<'a> -> int`

Return the number of elements in the set.

`val isSubset : Set<'a> -> Set<'a> -> bool`

Evaluates to "true" if all elements of the first set are in the second.

`val isProperSubset : Set<'a> -> Set<'a> -> bool`

Evaluates to "true" if all elements of the first set are in the second, and there is at least one element in the second set which is not in the first.

```
> let a = Set.ofSeq [ 1 .. 10 ]
let b = Set.ofSeq [ 5 .. 15 ]
let c = Set.ofSeq [ 2; 4; 5; 9 ];;

val a : Set<int>
val b : Set<int>
val c : Set<int>

> Set.isSubset c a;; (* All elements of 'c' exist in 'a' *)
val it : bool = true

> Set.isSubset c b;; (* Not all of the elements of 'c' exist in 'b' *);;
val it : bool = false
```

`val union : Set<'a> -> Set<'a> -> Set<'a>`

Compute the union of the two sets.

```
> let a = Set.ofSeq [ 1 .. 10 ]
let b = Set.ofSeq [ 5 .. 15 ];;

val a : Set<int>
val b : Set<int>

> Set.iter (fun x -> printf "%O " x) (Set.union a b);;
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 val it : unit = ()

> Set.iter (fun x -> printf "%O " x) (a + b);; (* '+' computes union *)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Examples

```
open System

let shakespeare = "O Romeo, Romeo! wherefore art thou Romeo?"
let shakespeareArray =
    shakespeare.Split([| ' '; ','; '!'; '?' |], StringSplitOptions.RemoveEmptyEntries)
let shakespeareSet = shakespeareArray |> Set.ofSeq

let main() =
    printfn "shakespeare: %A" shakespeare

    let printCollection msg coll =
        printfn "%s:" msg
        Seq.iteri (fun index item -> printfn "  %i: %O" index item) coll

    printCollection "shakespeareArray" shakespeareArray
    printCollection "shakespeareSet" shakespeareSet

    Console.ReadKey(true) |> ignore

main()
```

```
shakespeare: "O Romeo, Romeo! wherefore art thou Romeo?"
shakespeareArray:
  0: O
  1: Romeo
  2: Romeo
  3: wherefore
  4: art
  5: thou
  6: Romeo
shakespeareSet:
  0: O
  1: Romeo
  2: art
  3: thou
  4: wherefore
```

Maps

A map is a special kind of set: it associates *keys* with *values*. A map is created in a similar way to sets:

```
> let holidays =
    Map.empty. (* Start with empty Map *)
        Add("Christmas", "Dec. 25").
        Add("Halloween", "Oct. 31").
        Add("Darwin Day", "Feb. 12").
        Add("World Vegan Day", "Nov. 1");;

val holidays : Map<string,string>

> let monkeys =
    [ "Squirrel Monkey", "Simia sciureus";
        "Marmoset", "Callithrix jacchus";
        "Macaque", "Macaca mulatta";
        "Gibbon", "Hylobates lar";
        "Gorilla", "Gorilla gorilla";
        "Humans", "Homo sapiens";
        "Chimpanzee", "Pan troglodytes" ]
    |> Map.ofList;; (* Convert list to Map *)

val monkeys : Map<string,string>
```

You can use the `.[key]` to access elements in the map:

```
> holidays.["Christmas"];;
val it : string = "Dec. 25"

> monkeys.["Marmoset"];;
val it : string = "Callithrix jacchus"
```

The `Map` Module

The FSharp.Collections.Map (https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-fsharpmap-2.html) module handles map operations.

`val add : 'key -> 'a -> Map<'key,'a> -> Map<'key,'a>`

Return a new map with the binding added to the given map.

`val empty<'key,'a> : Map<'key,'a>`

Returns an empty map.

`val exists : ('key -> 'a -> bool) -> Map<'key,'a> -> bool`

Return true if the given predicate returns true for one of the bindings in the map.

`val filter : ('key -> 'a -> bool) -> Map<'key,'a> -> Map<'key,'a>`

Build a new map containing only the bindings for which the given predicate returns `true`.

`val find : 'key -> Map<'key,'a> -> 'a`

Lookup an element in the map, raising `KeyNotFoundException` if no binding exists in the map.

`val containsKey: 'key -> Map<'key,'a> -> bool`

Test if an element is in the domain of the map.

`val remove : 'key -> Map<'key,'a> -> Map<'key,'a>`

Remove an element from the domain of the map. No exception is raised if the element is not present.

`val tryFind : 'key -> Map<'key,'a> -> 'a option`

Lookup an element in the map, returning a `Some` value if the element is in the domain of the map and `None` if not.

Examples

```
open System

let capitals =
    [("Australia", "Canberra"); ("Canada", "Ottawa"); ("China", "Beijing");
        ("Denmark", "Copenhagen"); ("Egypt", "Cairo"); ("Finland", "Helsinki");
        ("France", "Paris"); ("Germany", "Berlin"); ("India", "New Delhi");
        ("Japan", "Tokyo"); ("Mexico", "Mexico City"); ("Russia", "Moscow");
        ("Slovenia", "Ljubljana"); ("Spain", "Madrid"); ("Sweden", "Stockholm");
        ("Taiwan", "Taipei"); ("USA", "Washington D.C.")]
    |> Map.ofList

let rec main() =
    Console.Write("Find a capital by country (type 'q' to quit): ")
    match Console.ReadLine() with
    | "q" -> Console.WriteLine("Bye bye")
    | country ->
        match capitals.TryFind(country) with
        | Some(capital) -> Console.WriteLine("The capital of {0} is {1}\n", country, capital)
        | None -> Console.WriteLine("Country not found.\n")
        main() (* loop again *)

main()
```

```
Find a capital by country (type 'q' to quit): Egypt
The capital of Egypt is Cairo

Find a capital by country (type 'q' to quit): Slovenia
The capital of Slovenia is Ljubljana

Find a capital by country (type 'q' to quit): Latveria
Country not found.

Find a capital by country (type 'q' to quit): USA
The capital of USA is Washington D.C.

Find a capital by country (type 'q' to quit): q
Bye bye
```

Set and Map Performance

F# sets and maps are implemented as immutable AVL trees, an efficient data structure which forms a self-balancing binary tree. AVL trees are well-known for their efficiency, in which they can search, insert, and delete elements in the tree in O(log *n*) time, where *n* is the number of elements in the tree.

Discriminated Unions

F# : Discriminated Unions

Discriminated unions, also called **tagged unions**, represent a finite, well-defined set of choices. Discriminated unions are often the tool of choice for building up more complicated data structures including linked lists and a wide range of trees.

Creating Discriminated Unions

Discriminated unions are defined using the following syntax:

```
type unionName =
    | Case1
    | Case2 of datatype
    | ...
```

By convention, union names start with a lowercase character, and union cases are written in PascalCase.

Union basics: an On/Off switch

Let's say we have a light switch. For most of us, a light switch has two possible states: the light switch can be ON, or it can be OFF. We can use an F# union to model our light switch's state as follows:

```
type switchstate =
    | On
    | Off
```

We've defined a union called `switchstate` which has two cases, `On` and `Off`. You can create and use instances of `switchstate` as follows:

```
type switchstate =
    | On
    | Off

let x = On (* creates an instance of switchstate *)
let y = Off (* creates another instance of switchstate *)

let main() =
    printfn "x: %A" x
    printfn "y: %A" y

main()
```

This program has the following types:

```
type switchstate = On | Off
val x : switchstate
val y : switchstate
```

It outputs the following:

```
x: On
y: Off
```

Notice that we create an instance of `switchstate` simply by using the name of its cases; this is because, in a literal sense, the cases of a union are constructors. As you may have guessed, since we use the same syntax for constructing objects as for matching them, we can pattern match on unions in the following way:

```
type switchstate =
    | On
    | Off

let toggle = function (* pattern matching input *)
    | On -> Off
    | Off -> On

let main() =
    let x = On
    let y = Off
    let z = toggle y

    printfn "x: %A" x
    printfn "y: %A" y
    printfn "z: %A" z
    printfn "toggle z: %A" (toggle z)

main()
```

The function `toggle` has the type `val toggle : switchstate -> switchstate`.

This program has the following output:

```
x: On
y: Off
z: On
toggle z: Off
```

Holding Data In Unions: a dimmer switch

The example above is kept deliberately simple. In fact, in many ways, the discriminated union defined above doesn't appear much different from an enum value. However, let's say we wanted to change our light switch model into a model of a dimmer switch, or in other words a light switch that allows users to adjust a lightbulb's power output from 0% to 100% power.

We can modify our union above to accommodate three possible states: On, Off, and an adjustable value between 0 and 100:

```
type switchstate =
    | On
    | Off
    | Adjustable of float
```

We've added a new case, `Adjustable of float`. This case is fundamentally the same as the others, except it takes a single `float` value in its constructor:

```
open System

type switchstate =
    | On
    | Off
    | Adjustable of float

let toggle = function
    | On -> Off
    | Off -> On
    | Adjustable(brightness) ->
        (* Matches any switchstate of type Adjustable. Binds
```

```
        the value passed into the constructor to the variable
        'brightness'. Toggles dimness around the halfway point. *)
        let pivot = 0.5
        if brightness <= pivot then
            Adjustable(brightness + pivot)
        else
            Adjustable(brightness - pivot)

let main() =
    let x = On
    let y = Off
    let z = Adjustable(0.25) (* takes a float in constructor *)

    printfn "x: %A" x
    printfn "y: %A" y
    printfn "z: %A" z
    printfn "toggle z: %A" (toggle z)

    Console.ReadKey(true) |> ignore

main()
```

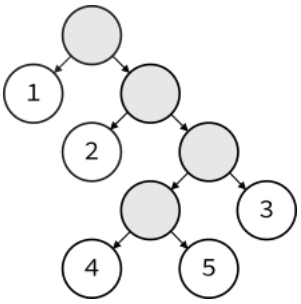
This program outputs:

```
x: On
y: Off
z: Adjustable 0.25
toggle z: Adjustable 0.75
```

Creating Trees

Discriminated unions can easily model a wide variety of trees and hierarchical data structures.

For example, let's consider the following binary tree:



Each node of our tree contains exactly two branches, and each branch can either be an integer or another tree. We can represent this tree as follows:

```
type tree =
    | Leaf of int
    | Node of tree * tree
```

We can create an instance of the tree above using the following code:

```
open System

type tree =
    | Leaf of int
    | Node of tree * tree

let simpleTree =
    Node(
        Leaf 1,
        Node(
            Leaf 2,
            Node(
                Node(
                    Leaf 4,
                    Leaf 5
                ),
                Leaf 3
            )
        )
    )

let main() =
    printfn "%A" simpleTree
    Console.ReadKey(true) |> ignore

main()
```

This program outputs the following:

```
Node (Leaf 1,Node (Leaf 2,Node (Node (Leaf 4,Leaf 5),Leaf 3)))
```

Very often, we want to recursively enumerate through all of the nodes in a tree structure. For example, if we wanted to count the total number of Leaf nodes in our tree, we can use:

```
open System
```

```
type tree =
    | Leaf of int
    | Node of tree * tree

let simpleTree =
    Node (Leaf 1, Node (Leaf 2, Node (Node (Leaf 4, Leaf 5), Leaf 3)))

let rec countLeaves = function
    | Leaf(_) -> 1
    | Node(tree1, tree2) ->
        (countLeaves tree1) + (countLeaves tree2)

let main() =
    printfn "countLeaves simpleTree: %i" (countLeaves simpleTree)
    Console.ReadKey(true) |> ignore

main()
```

This program outputs:

```
countLeaves simpleTree: 5
```

Generalizing Unions For All Datatypes

Note that our binary tree above only operates on integers. It is possible to construct unions that are generalized to operate on all possible data types. We can modify the definition of our union to the following:

```
type 'a tree =
    | Leaf of 'a
    | Node of 'a tree * 'a tree

(* The syntax above is "OCaml" style. It's common to see
   unions defined using the ".NET" style as follows which
   surrounds the type parameter with <'s and >'s after the
   type name:

type tree<'a> =
    | Leaf of 'a
    | Node of tree<'a> * tree<'a>
*)
```

We can use the union defined above to define a binary tree of any data type:

```
open System

type 'a tree =
    | Leaf of 'a
    | Node of 'a tree * 'a tree

let firstTree =
    Node(
        Leaf 1,
        Node(
            Leaf 2,
            Node(
                Node(
                    Leaf 4,
                    Leaf 5
                ),
                Leaf 3
            )
        )
    )

let secondTree =
    Node(
        Node(
            Node(
                Leaf "Red",
                Leaf "Orange"
            ),
            Node(
                Leaf "Yellow",
                Leaf "Green"
            )
        ),
        Node(
            Leaf "Blue",
            Leaf "Violet"
        )
    )

let prettyPrint tree =
    let rec loop depth tree =
        let spacer = new String(' ', depth)
        match tree with
        | Leaf(value) ->
            printfn "%s |- %A" spacer value
        | Node(tree1, tree2) ->
            printfn "%s |" spacer
            loop (depth + 1) tree1
            loop (depth + 1) tree2
    loop 0 tree

let main() =
    printfn "firstTree:"
    prettyPrint firstTree

    printfn "secondTree:"
    prettyPrint secondTree
    Console.ReadKey(true) |> ignore

main()
```

The functions above have the following types:

```
type 'a tree =
    | Leaf of 'a
    | Node of 'a tree * 'a tree

val firstTree : int tree
val secondTree : string tree
val prettyPrint : 'a tree -> unit
```

This program outputs:

```
firstTree:
 |
  |- 1
  |
   |- 2
   |
    |
     |- 4
     |- 5
    |- 3
secondTree:
 |
  |
   |
    |- "Red"
    |- "Orange"
   |
    |- "Yellow"
    |- "Green"
  |
   |- "Blue"
   |- "Violet"
```

Examples

Built-in Union Types

F# has several built-in types derived from discriminated unions, some of which have already been introduced in this tutorial. These types include:

```
type 'a list =
    | Cons of 'a * 'a list
    | Nil

type 'a option =
    | Some of 'a
    | None
```

Propositional Logic

The ML family of languages, which includes F# and its parent language OCaml, were originally designed for the development of automated theorem provers. Union types allow F# programmers to represent propositional logic remarkably concisely. To keep things simple, let's limit our propositions to four possible cases:

```
type proposition =
    | True
    | Not of proposition
    | And of proposition * proposition
    | Or of proposition * proposition
```

Let's say we had a series of propositions and wanted to determine whether they evaluate to true or false. We can easily write an eval function by recursively enumerating through a propositional statement as follows:

```
let rec eval = function
    | True -> true
    | Not(prop) -> not (eval(prop))
    | And(prop1, prop2) -> eval(prop1) && eval(prop2)
    | Or(prop1, prop2) -> eval(prop1) || eval(prop2)
```

The `eval` function has the type `val eval : proposition -> bool`.

Here is a full program using the eval function:

```
open System

type proposition =
    | True
    | Not of proposition
    | And of proposition * proposition
    | Or of proposition * proposition

let prop1 =
    (* ~t || ~~t *)
    Or(
        Not True,
        Not (Not True)
    )

let prop2 =
    (* ~(t && ~t) || ( (t || t) || ~t) *)
    Or(
```

```
        Not(
            And(
                True,
                Not True
            )
        ),
        Or(
            Or(
                True,
                True
            ),
            Not True
        )
    )

let prop3 =
    (* ~~~~~~~t *)
    Not(Not(Not(Not(Not(Not(Not True))))))

let rec eval = function
    | True -> true
    | Not(prop) -> not (eval(prop))
    | And(prop1, prop2) -> eval(prop1) && eval(prop2)
    | Or(prop1, prop2) -> eval(prop1) || eval(prop2)

let main() =
    let testProp name prop = printfn "%s: %b" name (eval prop)

    testProp "prop1" prop1
    testProp "prop2" prop2
    testProp "prop3" prop3

    Console.ReadKey(true) |> ignore

main()
```

This program outputs the following:

```
prop1: true
prop2: true
prop3: false
```

Additional Reading

Theorem Proving Examples (http://www.cl.cam.ac.uk/~jrh13/atp/index.html) (OCaml)

Mutable Data

F# : Mutable Data

All of the data types and values in F# seen so far have been immutable, meaning the values cannot be reassigned another value after they've been declared. However, F# allows programmers to create variables in the true sense of the word: variables whose values can change throughout the lifetime of the application.

`mutable` Keyword

The simplest mutable variables in F# are declared using the `mutable` keyword. Here is a sample using fsi:

```
> let mutable x = 5;;

val mutable x : int

> x;;
val it : int = 5

> x <- 10;;
val it : unit = ()

> x;;
val it : int = 10
```

As shown above, the `<-` operator is used to assign a mutable variable a new value. Notice that variable assignment actually returns `unit` as a value.

The mutable keyword is frequently used with record types to create mutable records:

```
open System

type transactionItem =
    { ID : int;
        mutable IsProcessed : bool;
        mutable ProcessedText : string; }

let getItem id =
    { ID = id;
        IsProcessed = false;
        ProcessedText = null; }

let processItems (items : transactionItem list) =
    items |> List.iter(fun item ->
```

```
        item.IsProcessed <- true
        item.ProcessedText <- sprintf "Processed %s" (DateTime.Now.ToString("hh:mm:ss"))

        Threading.Thread.Sleep(1000) (* Putting thread to sleep for 1 second to simulate
                                        processing overhead. *)
        )

let printItems (items : transactionItem list) =
        items |> List.iter (fun x -> printfn "%A" x)

let main() =
    let items = List.init 5 getItem

    printfn "Before process:"
    printItems items

    printfn "After process:"
    processItems items
    printItems items

    Console.ReadKey(true) |> ignore

main()
```

```
Before process:
{ID = 0;
 IsProcessed = false;
 ProcessedText = null;}
{ID = 1;
 IsProcessed = false;
 ProcessedText = null;}
{ID = 2;
 IsProcessed = false;
 ProcessedText = null;}
{ID = 3;
 IsProcessed = false;
 ProcessedText = null;}
{ID = 4;
 IsProcessed = false;
 ProcessedText = null;}
After process:
{ID = 0;
 IsProcessed = true;
 ProcessedText = "Processed 10:00:31";}
{ID = 1;
 IsProcessed = true;
 ProcessedText = "Processed 10:00:32";}
{ID = 2;
 IsProcessed = true;
 ProcessedText = "Processed 10:00:33";}
{ID = 3;
 IsProcessed = true;
 ProcessedText = "Processed 10:00:34";}
{ID = 4;
 IsProcessed = true;
 ProcessedText = "Processed 10:00:35";}
```

Limitations of Mutable Variables

Mutable variables are somewhat limited: before F# 4.0, mutables were inaccessible outside of the scope of the function where they are defined. Specifically, this means its not possible to reference a mutable in a subfunction of another function. Here's a demonstration in fsi:

```
> let testMutable() =
    let mutable msg = "hello"
    printfn "%s" msg

    let setMsg() =
        msg <- "world"

    setMsg()
    printfn "%s" msg;;

          msg <- "world"
  --------^^^^^^^^^^^^^^^

stdin(18,9): error FS0191: The mutable variable 'msg' is used in an invalid way. Mutable
variables may not be captured by closures. Consider eliminating this use of mutation or
using a heap-allocated mutable reference cell via 'ref' and '!'.
```

Ref cells

Ref cells get around some of the limitations of mutables. In fact, ref cells are very simple datatypes which wrap up a mutable field in a record type. Ref cells are defined by F# as follows:

```
type 'a ref = { mutable contents : 'a }
```

The F# library contains several built-in functions and operators for working with ref cells:

```
let ref v = { contents = v }      (* val ref  : 'a -> 'a ref *)
let (!) r = r.contents            (* val (!)  : 'a ref -> 'a *)
let (:=) r v = r.contents <- v    (* val (:=) : 'a ref -> 'a -> unit *)
```

The `ref` function is used to create a ref cell, the `!` operator is used to read the contents of a ref cell, and the `:=` operator is used to assign a ref cell a new value. Here is a sample in fsi:

```
> let x = ref "hello";;

val x : string ref

> x;; (* returns ref instance *)
val it : string ref = {contents = "hello";}

> !x;; (* returns x.contents *)
val it : string = "hello"

> x := "world";; (* updates x.contents with a new value *)
val it : unit = ()

> !x;; (* returns x.contents *)
val it : string = "world"
```

Since ref cells are allocated on the heap, they can be shared across multiple functions:

```
open System

let withSideEffects x =
    x := "assigned from withSideEffects function"

let refTest() =
    let msg = ref "hello"
    printfn "%s" !msg

    let setMsg() =
        msg := "world"

    setMsg()
    printfn "%s" !msg

    withSideEffects msg
    printfn "%s" !msg

let main() =
    refTest()
    Console.ReadKey(true) |> ignore

main()
```

The `withSideEffects` function has the type `val withSideEffects : string ref -> unit`.

This program outputs the following:

```
hello
world
assigned from withSideEffects function
```

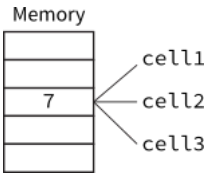
The `withSideEffects` function is named as such because it has a *side-effect*, meaning it can change the state of a variable in other functions. Ref Cells should be treated like fire. Use it cautiously when it is absolutely necessary but avoid it in general. If you find yourself using Ref Cells while translating code from C/C++, then ignore efficiency for a while and see if you can get away without Ref Cells or at worst using mutable. You would often stumble upon a more elegant and more maintanable algorithm

Aliasing Ref Cells

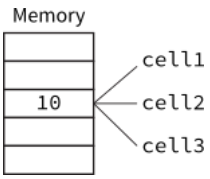
> **Note:** While imperative programming uses aliasing extensively, this practice has a number of problems. In particular it makes programs hard to follow since the state of any variable can be modified at any point elsewhere in an application. Additionally, multithreaded applications sharing mutable state are difficult to reason about since one thread can potentially change the state of a variable in another thread, which can result in a number of subtle errors related to race conditions and dead locks.

A ref cell is very similar to a C or C++ pointer. Its possible to point to two or more ref cells to the same memory address; changes at that memory address will change the state of all ref cells pointing to it. Conceptually, this process looks like this:

Let's say we have 3 ref cells looking at the same address in memory:



`cell1`, `cell2`, and `cell3` are all pointing to the same address in memory. The `.contents` property of each cell is `7`. Let's say, at some point in our program, we execute the code `cell1 := 10`, this changes the value in memory to the following:



By assigning `cell1.contents` a new value, the variables `cell2` and `cell3` were changed as well. This can be demonstrated using fsi as follows:

```
> let cell1 = ref 7;;
val cell1 : int ref
```

```
> let cell2 = cell1;;
val cell2 : int ref

> let cell3 = cell2;;
val cell3 : int ref

> !cell1;;
val it : int = 7

> !cell2;;
val it : int = 7

> !cell3;;
val it : int = 7

> cell1 := 10;;
val it : unit = ()

> !cell1;;
val it : int = 10

> !cell2;;
val it : int = 10

> !cell3;;
val it : int = 10
```

Encapsulating Mutable State

F# discourages the practice of passing mutable data between functions. Functions that rely on mutation should generally hide its implementation details behind a private function, such as the following example in FSI:

```
> let incr =
    let counter = ref 0
    fun () ->
        counter := !counter + 1
        !counter;;

val incr : (unit -> int)

> incr();;
val it : int = 1

> incr();;
val it : int = 2

> incr();;
val it : int = 3
```

Control Flow

F# : Control Flow

In all programming languages, **control flow** refers to the decisions made in code that affect the order in which statements are executed in an application. F#'s imperative control flow elements are similar to those encountered in other languages.

Imperative Programming in a Nutshell

Most programmers coming from a C#, Java, or C++ background are familiar with an imperative style of programming which uses loops, mutable data, and functions with side-effects in applications. While F# primarily encourages the use of a functional programming style, it has constructs which allow programmers to write code in a more imperative style as well. Imperative programming can be useful in the following situations:

- Interacting with many objects in the .NET Framework, most of which are inherently imperative.
- Interacting with components that depend heavily on side-effects, such as GUIs, I/O, and sockets.
- Scripting and prototyping snippets of code.
- Initializing complex data structures.
- Optimizing blocks of code where an imperative version of an algorithm is more efficient than the functional version.

`if/then` Decisions

F#'s `if/then/elif/else` construct has already been seen earlier in this book, but to introduce it more formally, the `if/then` construct has the following syntaxes:

```
(* simple if *)
if expr then
    expr

(* binary if *)
if expr then
    expr
else
    expr
```

```
(* multiple else branches *)
if expr then
    expr
elif expr then
    expr
elif expr then
    expr
...
else
    expr
```

Like all F# blocks, the scope of an `if` statement extends to any code indented under it. For example:

```
open System

let printMessage condition =
    if condition then
        printfn "condition = true: inside the 'if'"
    printfn "outside the 'if' block"

let main() =
    printMessage true
    printfn "--------"
    printMessage false
    Console.ReadKey(true) |> ignore

main()
```

This program prints:

```
condition = true: inside the 'if'
outside the 'if' block
--------
outside the 'if' block
```

Working With Conditions

F# has three boolean operators:

Symbol	Description	Example
&&	Logical AND (infix, short-circuited)	`true && false (* returns false *)`
||	Logical OR (infix, short-circuited)	`true || false (* returns true *)`
not	Logical NOT	`not false (* returns true *)`

The && and || operators are short-circuited, meaning the CLR will perform the minimum evaluation necessary to determine whether the condition will succeed or fail. For example, if the left side of an && evaluates to `false`, then there is no need to evaluate the right side; likewise, if the left side of a || evaluates to `true`, then there is no need to evaluate the right side of the expression.

Here is a demonstration of short-circuiting in F#:

```
open System

let alwaysTrue() =
    printfn "Always true"
    true

let alwaysFalse() =
    printfn "Always false"
    false

let main() =
    let testCases =
        ["alwaysTrue && alwaysFalse", fun() -> alwaysTrue() && alwaysFalse();
         "alwaysFalse && alwaysTrue", fun() -> alwaysFalse() && alwaysTrue();
         "alwaysTrue || alwaysFalse", fun() -> alwaysTrue() || alwaysFalse();
         "alwaysFalse || alwaysTrue", fun() -> alwaysFalse() || alwaysTrue();]

    testCases |> List.iter (fun (msg, res) ->
        printfn "%s: %b" msg (res())
        printfn "-------")

    Console.ReadKey(true) |> ignore

main()
```

The `alwaysTrue` and `alwaysFalse` methods return `true` and `false` respectively, but they also have a side-effect of printing a message to the console whenever the functions are evaluated.

This program outputs the following:

```
Always true
Always false
alwaysTrue && alwaysFalse: false
-------
```

```
Always false
alwaysFalse && alwaysTrue: false
-------
Always true
alwaysTrue || alwaysFalse: true
-------
Always false
Always true
alwaysFalse || alwaysTrue: true
-------
```

As you can see above, the expression `alwaysTrue && alwaysFalse` evaluates both sides of the expression. `alwaysFalse && alwaysTrue` only evaluates the left side of the expression; since the left side returns `false`, its unnecessary to evaluate the right side.

`for` Loops Over Ranges

`for` loops are traditionally used to iterate over a well-defined integer range. The syntax of a `for` loop is defined as:

```
for var = start-expr to end-expr do
    ... // loop body
```

Here's a trivial program which prints out the numbers 1 - 10:

```
let main() =
    for i = 1 to 10 do
        printfn "i: %i" i
main()
```

This program outputs:

```
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
```

This code takes input from the user to compute an average:

```
open System

let main() =
    Console.WriteLine("This program averages numbers input by the user.")
    Console.Write("How many numbers do you want to add? ")

    let mutable sum = 0
    let numbersToAdd = Console.ReadLine() |> int

    for i = 1 to numbersToAdd do
        Console.Write("Input #{0}: ", i)
        let input = Console.ReadLine() |> int
        sum <- sum + input

    let average = sum / numbersToAdd
    Console.WriteLine("Average: {0}", average)

main()
```

This program outputs:

```
This program averages numbers input by the user.
How many numbers do you want to add? 3
Input #1: 100
Input #2: 90
Input #3: 50
Average: 80
```

`for` Loops Over Collections and Sequences

Its often convenient to iterate over collections of items using the syntax:

```
for pattern in expr do
    ... // loop body
```

For example, we can print out a shopping list in fsi:

```
> let shoppingList =
    ["Tofu", 2, 1.99;
    "Seitan", 2, 3.99;
    "Tempeh", 3, 2.69;
    "Rice milk", 1, 2.95;];;

val shoppingList : (string * int * float) list

> for (food, quantity, price) in shoppingList do
```

```
    printfn "food: %s, quantity: %i, price: %g" food quantity price;;
food: Tofu, quantity: 2, price: 1.99
food: Seitan, quantity: 2, price: 3.99
food: Tempeh, quantity: 3, price: 2.69
food: Rice milk, quantity: 1, price: 2.95
```

`while` Loops

As the name suggests, `while` loops will repeat a block of code indefinitely while a particular condition is true. The syntax of a `while` loop is defined as follows:

```
while expr do
    ... // loop body
```

We use a while loop when we don't know how many times to execute a block of code. For example, lets say we wanted the user to guess a password to a secret area; the user could get the password right on the first try, or the user could try millions of passwords, we just don't know. Here is a short program that requires a user to guess a password correctly in at most 3 attempts:

```
open System

let main() =
    let password = "monkey"
    let mutable guess = String.Empty
    let mutable attempts = 0

    while password <> guess && attempts < 3 do
        Console.Write("What's the password? ")
        attempts <- attempts + 1
        guess <- Console.ReadLine()

    if password = guess then
        Console.WriteLine("You got the password right!")
    else
        Console.WriteLine("You didn't guess the password")

    Console.ReadKey(true) |> ignore


main()
```

This program outputs the following:

```
What's the password? kitty
What's the password? monkey
You got the password right!
```

Arrays

F# : Arrays

Arrays are a ubiquitous, familiar data structure used to represent a group of related, ordered values. Unlike F# data structures, arrays are mutable, meaning the values in an array can be changed after the array has been created.

Creating Arrays

Arrays are conceptually similar to lists. Naturally, arrays can be created using many of the same techniques as lists:

Array literals

```
> [| 1; 2; 3; 4; 5 |];;
val it : int array = [|1; 2; 3; 4; 5|]
```

Array comprehensions

F# supports array comprehensions using ranges and generators in the same style and format as list comprehensions:

```
> [| 1 .. 10 |];;
val it : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]

> [| 1 .. 3 .. 10 |];;
val it : int array = [|1; 4; 7; 10|]

> [| for a in 1 .. 5 do
    yield (a, a*a, a*a*a) |];;
val it : (int * int * int) array
= [|(1, 1, 1); (2, 4, 8); (3, 9, 27); (4, 16, 64); (5, 25, 125)|]
```

`System.Array` Methods

There are several methods in the `System.Array` module for creating arrays:

`val zeroCreate : int arraySize -> 'T []`

Creates an array with `arraySize` elements. Each element in the array holds the default value for the particular data type (`0` for numbers, `false` for bools, `null` for reference types).

```
> let (x : int array) = Array.zeroCreate 5;;
val x : int array

> x;;
val it : int array = [|0; 0; 0; 0; 0|]
```

`val create : int -> 'T value -> 'T []`

Creates an array with `arraySize` elements. Initializes each element in the array with `value`.

```
> Array.create 5 "Juliet";;
val it : string [] = [|"Juliet"; "Juliet"; "Juliet"; "Juliet"; "Juliet"|]
```

`val init : int arraySize -> (int index -> 'T) initializer -> 'T []`

Creates an array with `arraySize` elements. Initializes each element in the array with the `initializer` function.

```
> Array.init 5 (fun index -> sprintf "index: %i" index);;
val it : string []
= [|"index: 0"; "index: 1"; "index: 2"; "index: 3"; "index: 4"|]
```

Working With Arrays

Elements in an array are accessed by their *index*, or position in an array. Array indexes always start at `0` and end at `array.length - 1`. For example, lets take the following array:

```
let names = [| "Juliet"; "Monique"; "Rachelle"; "Tara"; "Sophia" |]
(* Indexes:        0          1           2          3         4 *)
```

This list contains 5 items. The first index is `0`, and the last index is `4`.

We can access items in the array using the `.[index]` operator, also called indexer notation. Here is the same array in fsi:

```
> let names = [| "Juliet"; "Monique"; "Rachelle"; "Tara"; "Sophia" |];;

val names : string array

> names.[2];;
val it : string = "Rachelle"

> names.[0];;
val it : string = "Juliet"
```

Instances of arrays have a `Length` property which returns the number of elements in the array:

```
> names.Length;;
val it : int = 5

> for i = 0 to names.Length - 1 do
    printfn "%s" (names.[i]);;
Juliet
Monique
Rachelle
Tara
Sophia
```

Arrays are mutable data structures, meaning we can assign elements in an array new values at any point in our program:

```
> names;;
val it : string array = [|"Juliet"; "Monique"; "Rachelle"; "Tara"; "Sophia"|]

> names.[4] <- "Kristen";;
val it : unit = ()

> names;;
val it : string array = [|"Juliet"; "Monique"; "Rachelle"; "Tara"; "Kristen"|]
```

If you try to access an element outside the range of an array, you'll get an exception:

```
> names.[-1];;
System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at <StartupCode$FSI_0029>.$FSI_0029._main()
stopped due to error
```

```
> names.[5];;
System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at <StartupCode$FSI_0030>.$FSI_0030._main()
stopped due to error
```

Array Slicing

F# supports a few useful operators which allow programmers to return "slices" or subarrays of an array using the `.[start..finish]` operator, where one of the `start` and `finish` arguments may be omitted.

```
> let names = [|"0: Juliet"; "1: Monique"; "2: Rachelle"; "3: Tara"; "4: Sophia"|];;

val names : string array

> names.[1..3];; (* Grabs items between index 1 and 3 *)
val it : string [] = [|"1: Monique"; "2: Rachelle"; "3: Tara"|]

> names.[2..];; (* Grabs items between index 2 and last element *)
val it : string [] = [|"2: Rachelle"; "3: Tara"; "4: Sophia"|]

> names.[..3];; (* Grabs items between first element and index 3 *)
val it : string [] = [|"0: Juliet"; "1: Monique"; "2: Rachelle"; "3: Tara"|]
```

Note that array slices generate a new array, rather than altering the existing array. This requires allocating new memory and copying elements from our source array into our target array. If performance is a high priority, it is generally more efficient to look at parts of an array using a few index adjustments.

Multi-dimensional Arrays

A multi-dimensional array is literally an array of arrays. Conceptually, it's not any harder to work with these types of arrays than single-dimensional arrays (as shown above). Multi-dimensional arrays come in two forms: rectangular and jagged arrays.

Rectangular Arrays

A rectangular array, which may be called a grid or a matrix, is an array of arrays, where all of the inner arrays have the same length. Here is a simple 2x3 rectangular array in fsi:

```
> Array2D.zeroCreate<int> 2 3;;
val it : int [,] = [[|0; 0; 0|]; [|0; 0; 0|]|]
```

This array has 2 rows, and each row has 3 columns. To access elements in this array, you use the `.[row,col]` operator:

```
> let grid = Array2D.init<string> 3 3 (fun row col -> sprintf "row: %i, col: %i" row col);;

val grid : string [,]

> grid;;
val it : string [,]
= [[|"row: 0, col: 0"; "row: 0, col: 1"; "row: 0, col: 2"|];
    [|"row: 1, col: 0"; "row: 1, col: 1"; "row: 1, col: 2"|];
    [|"row: 2, col: 0"; "row: 2, col: 1"; "row: 2, col: 2"|]|]

> grid.[0, 1];;
val it : string = "row: 0, col: 1"

> grid.[1, 2];;
val it : string = "row: 1, col: 2"
```

Here's a simple program to demonstrate how to use and iterate through multidimensional arrays:

```
open System

let printGrid grid =
    let maxY = (Array2D.length1 grid) - 1
    let maxX = (Array2D.length2 grid) - 1

    for row in 0 .. maxY do
        for col in 0 .. maxX do
            if grid.[row, col] = true then Console.Write("* ")
            else Console.Write("_ ")
        Console.WriteLine()

let toggleGrid (grid : bool[,]) =
    Console.WriteLine()
    Console.WriteLine("Toggle grid:")

    let row =
        Console.Write("Row: ")
        Console.ReadLine() |> int

    let col =
        Console.Write("Col: ")
        Console.ReadLine() |> int

    grid.[row, col] <- (not grid.[row, col])

let main() =
    Console.WriteLine("Create a grid:")
    let rows =
        Console.Write("Rows: ")
        Console.ReadLine() |> int

    let cols =
        Console.Write("Cols: ")
```

```
        Console.ReadLine() |> int

    let grid = Array2D.zeroCreate<bool> rows cols
    printGrid grid

    let mutable go = true
    while go do
        toggleGrid grid
        printGrid grid
        Console.Write("Keep playing (y/n)? ")
        go <- Console.ReadLine() = "y"

    Console.WriteLine("Thanks for playing")

main()
```

This program outputs the following:

```
Create a grid:
Rows: 2
Cols: 3
_ _ _
_ _ _

Toggle grid:
Row: 0
Col: 1
_ * _
_ _ _
Keep playing (y/n)? y

Toggle grid:
Row: 1
Col: 1
  *
_ * _
_ * _
Keep playing (y/n)? y

Toggle grid:
Row: 1
Col: 2
_ * _
_ * *
_
Keep playing (y/n)? n

Thanks for playing
```

In additional to the `Array2D` module, F# has an `Array3D` module to support 3-dimensional arrays as well.

> **Note** It's possible to create arrays with more than 3 dimensions using the `System.Array.CreateInstance` method, but it's generally recommended to avoid creating arrays with huge numbers of elements or dimensions, since it is very hard to manage, and also can quickly consume all of the available memory on a machine.

Jagged arrays

A jagged array is an array of arrays, except each row in the array does not necessarily need to have the same number of elements:

```
> [| for a in 1 .. 5 do yield [| 1 .. a |] |];;
val it : int array array
= [|[|1|];
    [|1; 2|];
    [|1; 2; 3|];
    [|1; 2; 3; 4|];
    [|1; 2; 3; 4; 5|]|]
```

You use the `.[index]` operator to access items in the array. Since each element contains another array, it's common to see code that resembles `.[row].[col]`:

```
> let jagged = [| for a in 1 .. 5 do yield [| 1 .. a |] |]
for arr in jagged do
    for col in arr do
        printf "%i " col
    printfn "";;

val jagged : int array array

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

> jagged.[2].[2];;
val it : int = 3

> jagged.[4].[0];;
val it : int = 1
```

> **Note:** Notice that the data type of a rectangular array is `'a[,]`, but the data type of a jagged array is `'a array array`. This results because a rectangular array stores data "flat", whereas a jagged array is literally an array of *pointers to arrays*. Since these two types of arrays are stored differently in memory, F# treats `'a[,]` and `'a array array` as two different, non-interchangeable data types. As a result, rectangular and jagged arrays have slightly different syntax for element access.
>
> Rectangular arrays are stored in a slightly more efficient manner and generally perform better than jagged arrays, although there may not be a perceptible difference in most applications. However, it's worth noting performance differences between the two in benchmark tests (http://msdn.microsoft.com/en-us/magazine/cc163995.aspx#S10).

Using the `Array` Module

There are two array modules, `System.Array` and `Microsoft.FSharp.Collections.Array`, developed by the .NET BCL designers and the creators of F# respectively. Many of the methods and functions in the F# Array module are similar to those in the List module.

`val append : 'T[] first -> 'T[] second -> 'T[]`

Returns a new array with elements consisting of the first array followed by the second array.

`val choose : ('T item -> 'U option) -> 'T[] input -> 'U[]`

Filters and maps an array, returning a new array consisting of all elements which returned `Some`.

`val copy : 'T[] input -> 'T[]`

Returns a copy of the input array.

`val fill : 'T[] input -> int start -> int end -> 'T value -> unit`

Assigns `value` to all the elements between the start and end indexes.

`val filter : ('T -> bool) -> 'T[] -> 'T[]`

Returns a new array consisting of items filtered out of the input array.

`val fold : ('State -> 'T -> 'State) -> 'State -> 'T[] input -> 'State`

Accumulates left to right over an array.

`val foldBack : ('T -> 'State -> 'State) -> 'T[] input -> 'State -> 'State`

Accumulates right to left over an array.

`val iter : ('T -> unit) -> 'T[] input -> unit`

Applies a function to all of the elements of the input array.

`val length : 'T[] -> int`

Returns the number of items in an array.

`val map : ('T -> 'U) -> 'T[] -> 'U[]`

Applies a mapping function to each element in the input array to return a new array.

`val rev : 'T[] input -> 'T[]`

Returns a new array with the items in reverse order.

`val sort : 'T[] -> 'T[]`

Sorts a copy of an array.

`val sortInPlace : 'T[] -> unit`

Sorts an array in place. Note that the `sortInPlace` method returns `unit`, indicating the `sortInPlace` mutates the original array.

`val sortBy : ('T -> 'T -> int) -> 'T[] -> 'T[]`

Sorts a copy of an array based on the sorting function.

`val sub : 'T[] -> int start -> int end -> 'T[]`

Returns a sub array based on the given start and end indexes.

Differences Between Arrays and Lists

- Lists
  - 👍 Immutable, allows new lists to share nodes with other lists.
  - 👍 List literals.
  - 👍 Pattern matching.
  - 👍 Supports mapping and folding.
  - 👎 Linear lookup time.
  - 👎 No random access to elements, just "forward-only" traversal.
- Arrays
  - 👍 Array literals.
  - 👍 Constant lookup time.
  - 👍 Good spacial locality of reference ensures efficient lookup time.

- 👍 Indexes indicate the position of each element relative to others, making arrays ideal for random access.
- 👍 Supports mapping and folding.
- 👎 Mutability prevents arrays from sharing nodes with other elements in an array.
- 👎 Not resizable.

Representation in Memory

Items in an array are represented in memory as adjacent values in memory. For example, lets say we create the following int array:

```
[| 15; 5; 21; 0; 9 |]
```

Represented in memory, our array resembles something like this:

```
Memory Location: |  100 |  104 |  108 |  112 |  116
          Value: |   15 |    5 |   21 |    0 |    9
          Index: |    0 |    1 |    2 |    3 |    4
```

Each `int` occupies 4 bytes of memory. Since our array contains 5 elements, the operating systems allocate 20 bytes of memory to hold this array (4 bytes * 5 elements = 20 bytes). The first element in the array occupies memory 100-103, the second element occupies 104-107, and so on.

We know that each element in the array is identified by its *index* or position in the array. Literally, the index is an offset: since the array starts at memory location 100, and each element in the array occupies a fixed amount of memory, the operating system can know the exact address of each element in memory using the formula:

> Start memory address of element at index n = StartPosition of array + (n * length of data type)
> End memory address of element at index n = Start memory address + length of data type

In layman's terms, this means we can access the *nth* element of an array in constant time, or in O(1) operations. This is in contrast to lists, where accessing the *nth* element requires O(n) operations.

With the understanding that elements in an array occupy adjacent memory locations, we can deduce two properties of arrays:

1. Creating arrays requires programmers to specify the size of the array up front, otherwise, the operating system won't know how many adjacent memory locations to allocate.
2. Arrays are not resizable, because memory locations before the first element or beyond the last element may hold data used by other applications. An array is only "resized" by allocating a new block of memory and copying all of the elements from the old array into the new array.

Mutable Collections

F# : Mutable Collections

The .NET BCL comes with its own suite of **mutable collections** which are found in the System.Collections.Generic (http://msdn.microsoft.com/en-us/library/system.collections.generic.aspx) namespace. These built-in collections are very similar to their immutable counterparts in F#.

`List<'T>` Class

The `List<'T> (http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx)` class represents a strongly typed list of objects that can be accessed by index. Conceptually, this makes the `List<'T>` class similar to arrays. However, unlike arrays, `Lists` can be resized and don't need to have their size specified on declaration.

.NET lists are created using the `new` keyword and calling the list's constructor as follows:

```
> open System.Collections.Generic;;
> let myList = new List<string>();;

val myList : List<string>

> myList.Add("hello");;
val it : unit = ()
> myList.Add("world");;
val it : unit = ()
> myList.[0];;
val it : string = "hello"
> myList |> Seq.iteri (fun index item -> printfn "%i: %s" index myList.[index]);;
0: hello
1: world
```

It's easy to tell that .NET lists are mutable because their `Add` methods return `unit` rather than returning another list.

Underlying Implementation

Behind the scenes, the `List<'T>` class is just a fancy wrapper for an array. When a `List<'T>` is constructed, it creates an 4-element array in memory. Adding the first 4 items is an *O(1)* operation. However, as soon as the 5th element needs to be added, the list doubles the size of the internal array, which means it has to reallocate new memory and copy elements in the existing list; this is a *O(n)* operation, where *n* is the number of items in the list.

The `List<'T>.Count` property returns the number of items currently held in the collection, the `List<'T>.Capacity` collection returns the size of the underlying array. This code sample demonstrates how the underlying array is resized:

```
open System
open System.Collections.Generic

let items = new List<string>()

let printList (l : List<_>) =
    printfn "l.Count: %i, l.Capacity: %i" l.Count l.Capacity
    printfn "Items:"
    l |> Seq.iteri (fun index item ->
        printfn "    l.[%i]: %s" index l.[index])
    printfn "-----------"

let main() =
    printList items

    items.Add("monkey")
    printList items

    items.Add("kitty")
    items.Add("bunny")
    printList items

    items.Add("doggy")
    items.Add("octopussy")
    items.Add("ducky")
    printList items

    printfn "Removing entry for \"doggy\"\n--------\n"
    items.Remove("doggy") |> ignore
    printList items

    printfn "Removing item at index 3\n--------\n"
    items.RemoveAt(3)
    printList items

    Console.ReadKey(true) |> ignore

main()
```

```
l.Count: 0, l.Capacity: 0
Items:
-----------
l.Count: 1, l.Capacity: 4
Items:
    l.[0]: monkey
-----------
l.Count: 3, l.Capacity: 4
Items:
    l.[0]: monkey
    l.[1]: kitty
    l.[2]: bunny
-----------
l.Count: 6, l.Capacity: 8
Items:
    l.[0]: monkey
    l.[1]: kitty
    l.[2]: bunny
    l.[3]: doggy
    l.[4]: octopussy
    l.[5]: ducky
-----------
Removing entry for "doggy"
--------

l.Count: 5, l.Capacity: 8
Items:
    l.[0]: monkey
    l.[1]: kitty
    l.[2]: bunny
    l.[3]: octopussy
    l.[4]: ducky
-----------
Removing item at index 3
--------

l.Count: 4, l.Capacity: 8
Items:
    l.[0]: monkey
    l.[1]: kitty
    l.[2]: bunny
    l.[3]: ducky
-----------
```

If you know the maximum size of the list beforehand, it is possible to avoid the performance hit by calling the `List<'T>(size : int)` constructor instead. The following sample demonstrates how to add 1000 items to a list without resizing the internal array:

```
> let myList = new List<int>(1000);;

val myList : List<int>

> myList.Count, myList.Capacity;;
val it : int * int = (0, 1000)
> seq { 1 .. 1000 } |> Seq.iter (fun x -> myList.Add(x));;
val it : unit = ()
> myList.Count, myList.Capacity;;
val it : int * int = (1000, 1000)
```

`LinkedList<'T>` Class

A `LinkedList<'T> (http://msdn.microsoft.com/en-us/library/he2s3bh7.aspx)` represented a doubly-linked sequence of nodes which allows efficient *O(1)* inserts and removal, supports forward and backward traversal, but its implementation prevents efficient random access. Linked lists have a few valuable methods:

```
(* Prepends an item to the LinkedList *)
val AddFirst : 'T -> LinkedListNode<'T>

(* Appends an items to the LinkedList *)
val AddLast : 'T -> LinkedListNode<'T>

(* Adds an item before a LinkedListNode *)
val AddBefore : LinkedListNode<'T> -> 'T -> LinkedListNode<'T>

(* Adds an item after a LinkedListNode *)
val AddAfter : LinkedListNode<'T> -> 'T -> LinkedListNode<'T>
```

Note that these methods return a `LinkedListNode<'T>`, not a new `LinkedList<'T>`. Adding nodes actually mutates the `LinkedList` object:

```
> open System.Collections.Generic;;
> let items = new LinkedList<string>();;

val items : LinkedList<string>

> items.AddLast("AddLast1");;
val it : LinkedListNode<string>
= System.Collections.Generic.LinkedListNode`1[System.String]
    {List = seq ["AddLast1"];
     Next = null;
     Previous = null;
     Value = "AddLast1";}
> items.AddLast("AddLast2");;
val it : LinkedListNode<string>
= System.Collections.Generic.LinkedListNode`1[System.String]
    {List = seq ["AddLast1"; "AddLast2"];
     Next = null;
     Previous = System.Collections.Generic.LinkedListNode`1[System.String];
     Value = "AddLast2";}
> let firstItem = items.AddFirst("AddFirst1");;

val firstItem : LinkedListNode<string>

> let addAfter = items.AddAfter(firstItem, "addAfter");;

val addAfter : LinkedListNode<string>

> let addBefore = items.AddBefore(addAfter, "addBefore");;

val addBefore : LinkedListNode<string>

> items |> Seq.iter (fun x -> printfn "%s" x);;
AddFirst1
addBefore
addAfter
AddLast1
AddLast2
```

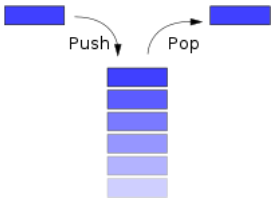
The `Stack<'T>` and `Queue<'T>` classes are special cases of a linked list (they can be thought of as linked lists with restrictions on where you can add and remove items).

`Stack<'T>` Class

A `Stack<'T> (http://msdn.microsoft.com/en-us/library/3278tedw.aspx)` only allows programmers prepend/push and remove/pop items from the front of a list, which makes it a *last in, first out* (LIFO) data structure. The `Stack<'T>` class can be thought of as a mutable version of the F# list.

```
> stack.Push("First");; (* Adds item to front of the list *)
val it : unit = ()
> stack.Push("Second");;
val it : unit = ()
> stack.Push("Third");;
val it : unit = ()
> stack.Pop();; (* Returns and removes item from front of the list *)
val it : string = "Third"
> stack.Pop();;
val it : string = "Second"
> stack.Pop();;
val it : string = "First"
```

A stack of coins could be represented with this data structure. If we stacked coins one on top another, the first coin in the stack is at the bottom of the stack, and the last coin in the stack appears at the top. We remove coins from top to bottom, so the last coin added to the stack is the first one removed.



`Queue<'T>` Class

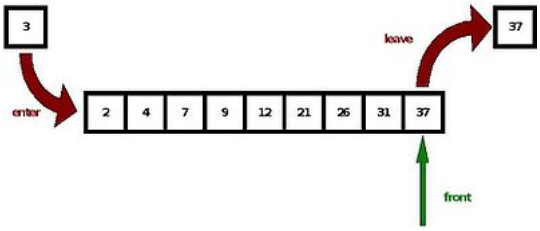
A `Queue<'T> (http://msdn.microsoft.com/en-us/library/7977ey2c.aspx)` only allows programmers to append/enqueue to the rear of a list and remove/dequeue from the front of a list, which makes it a *first in, first out* (FIFO) data structure.

```
> let queue = new Queue<string>();;

val queue : Queue<string>

> queue.Enqueue("First");; (* Adds item to the rear of the list *)
val it : unit = ()
> queue.Enqueue("Second");;
val it : unit = ()
> queue.Enqueue("Third");;
val it : unit = ()
> queue.Dequeue();; (* Returns and removes item from front of the queue *)
val it : string = "First"
> queue.Dequeue();;
val it : string = "Second"
> queue.Dequeue();;
val it : string = "Third"
```

A line of people might be represented by a queue: people add themselves to the rear of the line, and are removed from the front. The first person to stand in line is the first person to be served.



`HashSet<'T>`, and `Dictionary<'TKey, 'TValue>` Classes

The `HashSet<'T> (http://msdn.microsoft.com/en-us/library/bb359438.aspx)` and `Dictionary<'TKey, 'TValue> (http://msdn.microsoft.com/en-us/library/xfhwa508.aspx)` classes are mutable analogs of the F# set and map data structures and contain many of the same functions.

Using the HashSet<'T>

```
open System
open System.Collections.Generic

let nums_1to10 = new HashSet<int>()
let nums_5to15 = new HashSet<int>()

let main() =
    let printCollection msg targetSet =
        printf "%s: " msg
        targetSet |> Seq.sort |> Seq.iter(fun x -> printf "%O " x)
        printfn ""

    let addNums min max (targetSet : ICollection<_>) =
        seq { min .. max } |> Seq.iter(fun x -> targetSet.Add(x))

    addNums 1 10 nums_1to10
    addNums 5 15 nums_5to15

    printCollection "nums_1to10 (before)" nums_1to10
    printCollection "nums_5to15 (before)" nums_5to15

    nums_1to10.IntersectWith(nums_5to15) (* mutates nums_1to10 *)
    printCollection "nums_1to10 (after)" nums_1to10
    printCollection "nums_5to15 (after)" nums_5to15

    Console.ReadKey(true) |> ignore

main()
```

```
nums_1to10 (before): 1 2 3 4 5 6 7 8 9 10
nums_5to15 (before): 5 6 7 8 9 10 11 12 13 14 15
nums_1to10 (after): 5 6 7 8 9 10
nums_5to15 (after): 5 6 7 8 9 10 11 12 13 14 15
```

Using the Dictionary<'TKey, 'TValue>

```
> open System.Collections.Generic;;
> let dict = new Dictionary<string, string>();;

val dict : Dictionary<string,string>

> dict.Add("Garfield", "Jim Davis");;
val it : unit = ()
> dict.Add("Farside", "Gary Larson");;
val it : unit = ()
> dict.Add("Calvin and Hobbes", "Bill Watterson");;
val it : unit = ()
> dict.Add("Peanuts", "Charles Schultz");;
val it : unit = ()
> dict.["Farside"];; (* Use the '.[key]' operator to retrieve items *)
val it : string = "Gary Larson"
```

Differences Between .NET BCL and F# Collections

The major difference between the collections built into the .NET BCL and their F# analogs is, of course, mutability. The mutable nature of BCL collections dramatically affects their implementation and time-complexity:

.NET Data Structure	Insert	Remove	Lookup	F# Data Structure	Insert	Remove	Lookup
`List`	*O(1)* / *O(n)**	*O(n)*	*O(1)* (by index) / *O(n)* (linear)	No built-in equivalent			
`LinkedList`	*O(1)*	*O(1)*	*O(n)*	No built-in equivalent			
`Stack`	*O(1)*	*O(1)*	*O(n)*	`List`	*O(1)*	n/a	*O(n)*
`Queue`	*O(1)*	*O(1)*	*O(n)*	No built-in equivalent			
`HashSet`	*O(1)* / *O(n)**	*O(1)*	*O(1)*	`Set`	*O(log n)*	*O(log n)*	*O(log n)*
`Dictionary`	*O(1)* / *O(n)**	*O(1)*	*O(1)*	`Map`	*O(log n)*	*O(log n)*	*O(log n)*

* These classes are built on top of internal arrays. They may take a performance hit as the internal arrays are periodically resized when adding items.

Note: the Big-O notation above refers to the time-complexity of the insert/remove/retrieve operations relative to the number of items in the data structure, not the relative amount of time required to evaluate the operations relative to other data structures. For example, accessing arrays by index vs. accessing dictionaries by key have the same time complexity, *O(1)*, but the operations do not necessarily occur in the same amount of time.

Input and Output

F# : Basic I/O

Input and output, also called **I/O**, refers to any kind of communication between two hardware devices or between the user and the computer. This includes printing text out to the console, reading and writing files to disk, sending data over a socket, sending data to the printer, and a wide variety of other common tasks.

This page is not intended to provide an exhaustive look at .NET's I/O methods (readers seeking exhaustive references are encouraged to review the excellent documentation on the System.IO namespace on MSDN (http://msdn.microsoft.com/en-us/library/system.io.aspx)). This page will provide a cursory overview of some of the basic methods available to F# programmers for printing and working with files.

Working with the Console

With F#

By now, you're probably familiar with the `printf`, `printfn`, `sprintf` and its variants in the Printf module (http://msdn.microsoft.com/en-us/library/ee370560.aspx). However, just to describe these methods more formally, these methods are used for printf-style printing and formatting using % markers as placeholders:

Print methods take a format string and a series of arguments, for example:

```
> sprintf "Hi, I'm %s and I'm a %s" "Juliet" "Scorpio";;
val it : string = "Hi, I'm Juliet and I'm a Scorpio"
```

Methods in the `Printf` module are type-safe. For example, attempting to use substitute an `int` placeholder with a string results in a compilation error:

```
> sprintf "I'm %i years old" "kitty";;

  sprintf "I'm %i years old" "kitty";;
  -------------------------^^^^^^^^

stdin(17,28): error FS0001: The type 'string' is not compatible with any of the types
byte,int16,int32,int64,sbyte,uint16,uint32,uint64,nativeint,unativeint, arising from
the use of a printf-style format string.
```

According to the F# documentation, % placeholders consist of the following:

```
%[flags][width][.precision][type]
```

[flags] (optional)

Valid flags are:

0: add zeros instead of spaces to make up the required width
'-': left justify the result within the width specified
'+': add a '+' character if the number is positive (to match a '-' sign for negatives)
' ': add an extra space if the number is positive (to match a '-' sign for negatives)

[width] (optional)

The optional width is an integer indicating the minimal width of the result. For instance, %6d prints an integer, prefixing it with spaces to fill at least 6 characters. If width is '*', then an extra integer argument is taken to specify the corresponding width.

any number
'*':

[.precision] (optional)

Represents the number of digits after a floating point number.

```
> sprintf "%.2f" 12345.67890;;
val it : string = "12345.68"

> sprintf "%.7f" 12345.67890;;
val it : string = "12345.6789000"
```

[type] (required)

The following placeholder types are interpreted as follows:

```
     %b:         bool, formatted as "true" or "false"
     %s:         string, formatted as its unescaped contents
     %d, %i:     any basic integer type formatted as a decimal integer, signed if the basic integer type is signed.
     %u:         any basic integer type formatted as an unsigned decimal integer
     %x, %X, %o: any basic integer type formatted as an unsigned hexadecimal
                 (a-f)/Hexadecimal (A-F)/Octal integer

     %e, %E, %f, %F, %g, %G:
                 any basic floating point type (float,float32) formatted
                 using a C-style floating point format specifications, i.e

     %e, %E: Signed value having the form [-]d.dddde[sign]ddd where
                 d is a single decimal digit, dddd is one or more decimal
                 digits, ddd is exactly three decimal digits, and sign
                 is + or -

     %f:     Signed value having the form [-]dddd.dddd, where dddd is one
                 or more decimal digits. The number of digits before the
                 decimal point depends on the magnitude of the number, and
                 the number of digits after the decimal point depends on
                 the requested precision.

     %g, %G: Signed value printed in f or e format, whichever is
                 more compact for the given value and precision.


    %M:      System.Decimal value

    %O:      Any value, printed by boxing the object and using it's ToString method(s)

    %A:      Any value, printed by using Microsoft.FSharp.Text.StructuredFormat.Display.any_to_string with the default layout settings

    %a:      A general format specifier, requires two arguments:
                 (1) a function which accepts two arguments:
                     (a) a context parameter of the appropriate type for the
                         given formatting function (e.g. an #System.IO.TextWriter)
                     (b) a value to print
                         and which either outputs or returns appropriate text.

                 (2) the particular value to print


    %t:      A general format specifier, requires one argument:
                 (1) a function which accepts a context parameter of the
                     appropriate type for the given formatting function (e.g.
                     an #System.IO.TextWriter)and which either outputs or returns
                     appropriate text.

  Basic integer types are:
     byte,sbyte,int16,uint16,int32,uint32,int64,uint64,nativeint,unativeint
  Basic floating point types are:
     float, float32
```

Programmers can print to the console using the `printf` method, however F# recommends reading console input using the `System.Console.ReadLine()` method.

With .NET

.NET includes its own notation (http://msdn.microsoft.com/en-us/library/26etazsy.aspx) for format specifiers. .NET format strings are untyped. Additionally, .NET's format strings are designed to be extensible, meaning that a programmer can implement their own custom format strings. Format placeholders have the following form:

```
{index[, length][:formatString]}
```

For example, using the `String.Format` method in fsi:

```
> System.String.Format("Hi, my name is {0} and I'm a {1}", "Juliet", "Scorpio");;
val it : string = "Hi, my name is Juliet and I'm a Scorpio"

> System.String.Format("|{0,-50}|", "Left justified");;
val it : string = "|Left justified                                  |"

> System.String.Format("|{0,50}|", "Right justified");;
val it : string = "|                                   Right justified|"

> System.String.Format("|{0:yyyy-MMM-dd}|", System.DateTime.Now);;
val it : string = "|2009-Apr-06|"
```

See Number Format Strings (http://msdn.microsoft.com/en-us/library/427bttx3.aspx), Date and Time Format Strings (http://msdn.microsoft.com/en-us/library/97x6twsz.aspx), and Enum Format Strings (http://msdn.microsoft.com/en-us/library/c3s1ez6e.aspx) for a comprehensive reference on format specifiers for .NET.

Programmers can read and write to the console using the System.Console class (http://msdn.microsoft.com/en-us/library/system.console.aspx):

```
open System

let main() =
    Console.Write("What's your name? ")
    let name = Console.ReadLine()
    Console.Write("Hello, {0}", name)

main()
```

System.IO Namespace

The System.IO (http://msdn.microsoft.com/en-us/library/system.io.aspx) namespace contains a variety of useful classes for performing basic I/O.

Files and Directories

The following classes are useful for interrogating the host filesystem:

- The System.IO.File (http://msdn.microsoft.com/en-us/library/system.io.file.aspx) class exposes several useful members for creating, appending, and deleting files.
- System.IO.Directory (http://msdn.microsoft.com/en-us/library/system.io.directory.aspx) exposes methods for creating, moving, and deleting directories.
- System.IO.Path (http://msdn.microsoft.com/en-us/library/system.io.path.aspx) performs operations on strings which represent file paths.
- System.IO.FileSystemWatcher (http://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher.aspx) which allows users to listen to a directory for changes.

Streams

A stream is a sequence of bytes. .NET provides some classes which allow programmers to work with streams including:

- System.IO.StreamReader (http://msdn.microsoft.com/en-us/library/system.io.streamreader.aspx) which is used to read characters from a stream.
- System.IO.StreamWriter (http://msdn.microsoft.com/en-us/library/system.io.streamwriter.aspx) which is used to write characters to a stream.
- System.IO.MemoryStream (http://msdn.microsoft.com/en-us/library/system.io.memorystream.aspx) which creates an in-memory stream of bytes.

Exception Handling

F# : Exception Handling

When a program encounters a problem or enters an invalid state, it will often respond by throwing an exception. Left to its own devices, an uncaught exception will crash an application. Programmers write **exception handling** code to rescue an application from an invalid state.

Try/With

Let's look at the following code:

```
let getNumber msg = printf msg; int32(System.Console.ReadLine())

let x = getNumber("x = ")
let y = getNumber("y = ")
printfn "%i + %i = %i" x y (x + y)
```

This code is syntactically valid, and it has the correct types. However, it can fail at run time if we give it a bad input:

This program outputs the following:

```
x = 7
y = monkeys!
------------
FormatException was unhandled. Input string was not in a correct format.
```

The string `monkeys` does not represent a number, so the conversion fails with an exception. We can handle this exception using F#'s `try... with`, a special kind of pattern matching construct:

```
let getNumber msg =
    printf msg;
    try
        int32(System.Console.ReadLine())
    with
        | :? System.FormatException -> System.Int32.MinValue

let x = getNumber("x = ")
let y = getNumber("y = ")
printfn "%i + %i = %i" x y (x + y)
```

This program outputs the following:

```
x = 7
y = monkeys!
7 + -2147483648 = -2147483641
```

It is, of course, wholly possible to catch multiple types of exceptions in a single `with` block. For example, according to the MSDN documentation (http://msdn.microsoft.com/en-us/library/b3h1hf19.aspx), the `System.Int32.Parse(s : string)` method will throw three types of exceptions:

`ArgumentNullException (http://msdn.microsoft.com/en-us/library/system.argumentnullexception.aspx)`

> Occurs when `s` is a null reference.

`FormatException (http://msdn.microsoft.com/en-us/library/system.formatexception.aspx)`

> Occurs when `s` does not represent a numeric input.

`OverflowException (http://msdn.microsoft.com/en-us/library/system.overflowexception.aspx)`

> Occurs when `s` represents number greater than or less than `Int32.MaxValue` or `Int32.MinValue` (i.e. the number cannot be represented with a 32-bit signed integer).

We can catch all of these exceptions by adding additional match cases:

```
let getNumber msg =
    printf msg;
    try
        int32(System.Console.ReadLine())
    with
        | :? System.FormatException -> -1
        | :? System.OverflowException -> System.Int32.MinValue
        | :? System.ArgumentNullException -> 0
```

Its not necessary to have an exhaustive list of match cases on exception types, as the uncaught exception will simply move to the next method in the stack trace.

Raising Exceptions

The code above demonstrates how to recover from an invalid state. However, when designing F# libraries, its often useful to throw exceptions to notify users that the program encountered some kind of invalid input. There are several standard functions for raising exceptions:

```
(* General failure *)
val failwith : string -> 'a

(* General failure with formatted message *)
val failwithf : StringFormat<'a, 'b> -> 'a

(* Raise a specific exception *)
val raise : #exn -> 'a

(* Bad input *)
val invalidArg : string -> string -> 'a
```

For example:

```
type 'a tree =
    | Node of 'a * 'a tree * 'a tree
    | Empty

let rec add x = function
    | Empty -> Node(x, Empty, Empty)
    | Node(y, left, right) ->
        if x > y then Node(y, left, add x right)
        else if x < y then Node(y, add x left, right)
        else failwithf "Item '%A' has already been added to tree" x
```

Try/Finally

Normally, an exception will cause a function to exit immediately. However, a `finally` block will always execute, even if the code throws an exception:

```
let tryWithFinallyExample f =
    try
        printfn "tryWithFinallyExample: outer try block"
        try
            printfn "tryWithFinallyExample: inner try block"
            f()
        with
            | exn ->
                printfn "tryWithFinallyExample: inner with block"
                reraise() (* raises the same exception we just caught *)
    finally
        printfn "tryWithFinally: outer finally block"

let catchAllExceptions f =
    try
        printfn "-------------"
        printfn "catchAllExceptions: try block"
        tryWithFinallyExample f
    with
```

```
        | exn ->
            printfn "catchAllExceptions: with block"
            printfn "Exception message: %s" exn.Message

let main() =
    catchAllExceptions (fun () -> printfn "Function executed successfully")
    catchAllExceptions (fun () -> failwith "Function executed with an error")

main()
```

This program will output the following:

```
-------------
catchAllExceptions: try block
tryWithFinallyExample: outer try block
tryWithFinallyExample: inner try block
Function executed successfully
tryWithFinally: outer finally block
-------------
catchAllExceptions: try block
tryWithFinallyExample: outer try block
tryWithFinallyExample: inner try block
tryWithFinallyExample: inner with block
tryWithFinally: outer finally block
catchAllExceptions: with block
Exception message: Function executed with an error
```

Notice that our finally block executed in spite of the exception. Finally blocks are used most commonly to clean up resources, such as closing an open file handle or closing a database connection (even in the event of an exception, we *do not* want to leave file handles or database connections open):

```
open System.Data.SqlClient
let executeScalar connectionString sql =
    let conn = new SqlConnection(connectionString)
    try
        conn.Open() (* this line can throw an exception *)
        let comm = new SqlCommand(sql, conn)
        comm.ExecuteScalar() (* this line can throw an exception *)
    finally
        (* finally block guarantees our SqlConnection is closed, even if our sql statement fails *)
        conn.Close()
```

use Statement

Many objects in the .NET framework implement the System.IDisposable (http://msdn.microsoft.com/en-us/library/system.idisposable.aspx) interface, which means the objects have a special method called `Dispose` to guarantee deterministic cleanup of unmanaged resources. It's considered a best practice to call `Dispose` on these types of objects as soon as they are no longer needed.

Traditionally, we'd use a `try/finally` block in this fashion:

```
let writeToFile fileName =
    let sw = new System.IO.StreamWriter(fileName : string)
    try
        sw.Write("Hello ")
        sw.Write("World!")
    finally
        sw.Dispose()
```

However, this can be occasionally bulky and cumbersome, especially when dealing with many objects which implement the IDisposable interface. F# provides the keyword `use` as syntactic sugar for the pattern above. An equivalent version of the code above can be written as follows:

```
let writeToFile fileName =
    use sw = new System.IO.StreamWriter(fileName : string)
    sw.Write("Hello ")
    sw.Write("World!")
```

The scope of a `use` statement is identical to the scope of a `let` statement. F# will automatically call `Dispose()` on an object when the identifier goes out of scope.

Defining New Exceptions

F# allows us to easily define new types of exceptions using the `exception` declaration. Here's an example using fsi:

```
> exception ReindeerNotFoundException of string

let reindeer =
    ["Dasher"; "Dancer"; "Prancer"; "Vixen"; "Comet"; "Cupid"; "Donner"; "Blitzen"]

let getReindeerPosition name =
    match List.tryFindIndex (fun x -> x = name) reindeer with
    | Some(index) -> index
    | None -> raise (ReindeerNotFoundException(name));;

exception ReindeerNotFoundException of string
val reindeer : string list
val getReindeerPosition : string -> int

> getReindeerPosition "Comet";;
val it : int = 4

> getReindeerPosition "Donner";;
val it : int = 6
```

```
> getReindeerPosition "Rudolf";;
FSI_0033+ReindeerNotFoundExceptionException: Rudolf
   at FSI_0033.getReindeerPosition(String name)
   at <StartupCode$FSI_0036>.$FSI_0036._main()
stopped due to error
```

We can pattern match on our new existing exception type just as easily as any other exception:

```
> let tryGetReindeerPosition name =
    try
        getReindeerPosition name
    with
        | ReindeerNotFoundException(s) ->
            printfn "Got ReindeerNotFoundException: %s" s
            -1;;

val tryGetReindeerPosition : string -> int

> tryGetReindeerPosition "Comet";;
val it : int = 4

> tryGetReindeerPosition "Rudolf";;
Got ReindeerNotFoundException: Rudolf
val it : int = -1
```

Exception Handling Constructs

Construct	Kind	Description
`raise` *expr*	F# library function	Raises the given exception
`failwith` *expr*	F# library function	Raises the `System.Exception` exception
`try` *expr* `with` *rules*	F# expression	Catches expressions matching the pattern rules
`try` *expr* `finally` *expr*	F# expression	Execution the `finally` expression both when the computation is successful and when an exception is raised
`| :? ArgumentException`	F# pattern rule	A rule matching the given .NET exception type
`| :? ArgumentException as e`	F# pattern rule	A rule matching the given .NET exception type, binding the name e to the exception object value
`| Failure(msg) ->` *expr*	F# pattern rule	A rule matching the given data-carrying F# exception
`| exn ->` *expr*	F# pattern rule	A rule matching any exception, binding the name `exn` to the exception object value
`| exn when` *expr* `->` *expr*	F# pattern rule	A rule matching the exception under the given condition, binding the name `exn` to the exception object value

Operator Overloading

F# : Operator Overloading

Operator overloading allows programmers to provide new behavior for the default operators in F#. In practice, programmers overload operators to provide a simplified syntax for objects which can be combined mathematically.

Using Operators

You've already used operators:

```
let sum = x + y
```

Here **+** is example of using a mathematical addition operator.

Operator Overloading

Operators are functions with special names, enclosed in brackets. They must be defined as static class members. Here's an example on declaring **+** operator on complex numbers:

```
type Complex =
    {   Re: double
        Im: double }
    static member ( + ) (left: Complex, right: Complex) =
        { Re = left.Re + right.Re; Im = left.Im + right.Im }
```

In FSI, we can add two complex numbers as follows:

```
> let first = { Re = 1.0; Im = 7.0 };;
val first : Complex
```

```
> let second = { Re = 2.0; Im = -10.5 };;
val second : Complex

> first + second;;
val it : Complex = {Re = 3.0;
                    Im = -3.5;}
```

Defining New Operators

In addition to overloading existing operators, its possible to define new operators. The names of custom operators can only be one or more of the following characters:

```
!%&*+-./<=>?@^|~
```

F# supports two types of operators: infix operators and prefix operators.

Infix operators

An infix operator takes two arguments, with the operator appearing in between both arguments (i.e. `arg1 {op} arg2`). We can define our own infix operators using the syntax:

```
let (op) arg1 arg2 = ...
```

In addition to mathematical operators, F# has a variety of infix operators defined as part of its library, for example:

```
let inline (|>) x f = f x
let inline (::) hd tl = Cons(hd, tl)
let inline (:=) (x : 'a ref) value = x.contents <- value
```

Let's say we're writing an application which performs a lot of regex matching and replacing. We can match text using Perl-style operators by defining our own operators as follows:

```
open System.Text.RegularExpressions

let (=~) input pattern =
    Regex.IsMatch(input, pattern)

let main() =
    printfn "cat =~ dog: %b" ("cat" =~ "dog")
    printfn "cat =~ cat|dog: %b" ("cat" =~ "cat|dog")
    printfn "monkey =~ monk*: %b" ("monkey" =~ "monk*")

main()
```

This program outputs the following:

```
cat =~ dog: false
cat =~ cat|dog: true
monkey =~ monk*: true
```

Prefix Operators

Prefix operators take a single argument which appears to the right side of the operator (`{op}argument`). You've already seen how the `!` operator is defined for ref cells:

```
type 'a ref = { mutable contents : 'a }
let (!) (x : 'a ref) = x.contents
```

Let's say we're writing a number crunching application, and we wanted to define some operators that work on lists of numbers. We might define some prefix operators in fsi as follows:

```
> let ( !+ ) l = List.reduce ( + ) l
let ( !- ) l = List.reduce ( - ) l
let ( !* ) l = List.reduce ( * ) l
let ( !/ ) l = List.reduce ( / ) l;;

val ( !+ ) : int list -> int
val ( !- ) : int list -> int
val ( !* ) : int list -> int
val ( !/ ) : int list -> int

> !* [2; 3; 5];;
val it : int = 30

> !+ [2; 3; 5];;
val it : int = 10

> !- [2; 3; 7];;
val it : int = -8

> !/ [100; 10; 2];;
val it : int = 5
```

Classes

F# : Classes and Objects

In the real world, an **object** is a "real" thing. A cat, person, computer, and a roll of duct tape are all "real" things in the tangible sense. When we think about these things, we can broadly describe them in terms of a number of attributes:

- Properties: a person has a name, a cat has four legs, computers have a price tag, duct tape is sticky.
- Behaviors: a person reads the newspaper, cats sleep all day, computers crunch numbers, duct tape attaches things to other things.
- Types/group membership: an employee is a type of person, a cat is a pet, a Dell and Mac are types of computers, duct tape is part of the broader family of adhesives.

In the programming world, an "object" is, in the simplest of terms, a model of something in the real world. Object-oriented programming (OOP) exists because it allows programmers to model real-world entities and simulate their interactions in code. Just like their real-world counterparts, objects in computer programming have properties and behaviors, and can be classified according to their type.

While we can certainly create objects that represents cats, people, and adhesives, objects can also represent less concrete things, such as a bank account or a business rule.

Although the scope of OOP has expanded to include some advanced concepts such as design patterns and the large-scale architecture of applications, this page will keep things simple and limit the discussion of OOP to data modeling.

Defining an Object

Before an object is created, its properties and functions should be defined. You define properties and methods of an object in a *class*. There are actually two different syntaxes for defining a class: an implicit syntax and an explicit syntax.

Implicit Class Construction

Implicit class syntax is defined as follows:

```
type TypeName optional-type-arguments arguments [ as ident ] =
    [ inherit type { as base } ]
    [ let-binding | let-rec bindings ] *
    [ do-statement ] *
    [ abstract-binding | member-binding | interface-implementation ] *
```

> *Elements in brackets are optional, elements followed by a * may appear zero or more times.*

This syntax above is not as daunting as it looks. Here's a simple class written in implicit style:

```
type Account(number : int, holder : string) = class
    let mutable amount = 0m

    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount

    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount - value
end
```

The code above defines a class called `Account`, which has three properties and two methods. Let's take a closer look at the following:

```
type Account(number : int, holder : string) = class
```

The underlined code is called the class *constructor*. A constructor is a special kind of function used to initialize the fields in an object. In this case, our constructor defines two values, `number` and `holder`, which can be accessed anywhere in our class. You create an instance of `Account` by using the `new` keyword and passing the appropriate parameters into the constructor as follows:

```
let bob = new Account(123456, "Bob's Saving")
```

Additionally, let's look at the way a member is defined:

```
member x.Deposit(value) = amount <- amount + value
```

The `x` above is an alias for the object currently in scope. Most OO languages provide an implicit `this` or `self` variable to access the object in scope, but F# requires programmers to create their own alias.

After we can create an instance of our `Account`, we can access its properties using `.propertyName` notation. Here's an example in FSI:

```
> let printAccount (x : Account) =
    printfn "x.Number: %i, x.Holder: %s, x.Amount: %M" x.Number x.Holder x.Amount;;
```

```
val printAccount : Account -> unit

> let bob = new Account(123456, "Bob's Savings");;

val bob : Account

> printAccount bob;;
x.Number: 123456, x.Holder: Bob's Savings, x.Amount: 0
val it : unit = ()

> bob.Deposit(100M);;
val it : unit = ()

> printAccount bob;;
x.Number: 123456, x.Holder: Bob's Savings, x.Amount: 100
val it : unit = ()

> bob.Withdraw(29.95M);;
val it : unit = ()

> printAccount bob;;
x.Number: 123456, x.Holder: Bob's Savings, x.Amount: 70.05
```

Example

Let's use this class in a real program:

```
open System

type Account(number : int, holder : string) = class
    let mutable amount = 0m

    member x.Number = number
    member x.Holder = holder
    member x.Amount = amount

    member x.Deposit(value) = amount <- amount + value
    member x.Withdraw(value) = amount <- amount - value
end

let homer = new Account(12345, "Homer")
let marge = new Account(67890, "Marge")

let transfer amount (source : Account) (target : Account) =
    source.Withdraw amount
    target.Deposit amount

let printAccount (x : Account) =
    printfn "x.Number: %i, x.Holder: %s, x.Amount: %M" x.Number x.Holder x.Amount

let main() =
    let printAccounts() =
        [homer; marge] |> Seq.iter printAccount

    printfn "\nInializing account"
    homer.Deposit 50M
    marge.Deposit 100M
    printAccounts()

    printfn "\nTransferring $30 from Marge to Homer"
    transfer 30M marge homer
    printAccounts()

    printfn "\nTransferring $75 from Homer to Marge"
    transfer 75M homer marge
    printAccounts()

main()
```

The program has the following types:

```
type Account =
  class
    new : number:int * holder:string -> Account
    member Deposit : value:decimal -> unit
    member Withdraw : value:decimal -> unit
    member Amount : decimal
    member Holder : string
    member Number : int
  end
val homer : Account
val marge : Account
val transfer : decimal -> Account -> Account -> unit
val printAccount : Account -> unit
```

The program outputs the following:

```
Initializing account
x.Number: 12345, x.Holder: Homer, x.Amount: 50
x.Number: 67890, x.Holder: Marge, x.Amount: 100

Transferring $30 from Marge to Homer
x.Number: 12345, x.Holder: Homer, x.Amount: 80
x.Number: 67890, x.Holder: Marge, x.Amount: 70

Transferring $75 from Homer to Marge
x.Number: 12345, x.Holder: Homer, x.Amount: 5
x.Number: 67890, x.Holder: Marge, x.Amount: 145
```

Example using the do keyword

The do keyword is used for post-constructor initialization. For example, to create an object which represents a stock, it is necessary to pass in the stock symbol and initialize the rest of the properties in our constructor:

```
open System
open System.Net

type Stock(symbol : string) = class
    let url =
        "http://download.finance.yahoo.com/d/quotes.csv?s=" + symbol + "&f=sl1d1t1c1ohgv&e=.csv"

    let mutable _symbol = String.Empty
    let mutable _current = 0.0
    let mutable _open = 0.0
    let mutable _high = 0.0
    let mutable _low = 0.0
    let mutable _volume = 0

    do
    (* We initialize our object in the do block *)

        let webClient = new WebClient()

        (* Data comes back as a comma-separated list, so we split it
           on each comma *)
        let data = webClient.DownloadString(url).Split([|','|])

        _symbol <- data.[0]
        _current <- float data.[1]
        _open <- float data.[5]
        _high <- float data.[6]
        _low <- float data.[7]
        _volume <- int data.[8]

    member x.Symbol = _symbol
    member x.Current = _current
    member x.Open = _open
    member x.High = _high
    member x.Low = _low
    member x.Volume = _volume
end

let main() =
    let stocks =
        ["msft"; "noc"; "yhoo"; "gm"]
        |> Seq.map (fun x -> new Stock(x))

    stocks |> Seq.iter (fun x -> printfn "Symbol: %s (%F)" x.Symbol x.Current)

main()
```

This program has the following types:

```
type Stock =
  class
    new : symbol:string -> Stock
    member Current : float
    member High : float
    member Low : float
    member Open : float
    member Symbol : string
    member Volume : int
  end
```

This program outputs the following (your outputs will vary):

```
Symbol: "MSFT" (19.130000)
Symbol: "NOC" (43.240000)
Symbol: "YHOO" (12.340000)
Symbol: "GM" (3.660000)
```

> **Note:** It's possible to have any number of do statements in a class definition, although there's no particular reason to need more than one.

Explicit Class Definition

Classes written in explicit style follow this format:

```
type TypeName =
    [ inherit type ]
    [ val-definitions ]
    [ new ( optional-type-arguments arguments ) [ as ident ] =
      { field-initialization }
      [ then constructor-statements ]
    ] *
    [ abstract-binding | member-binding | interface-implementation ] *
```

Here's a class defined using the explicit syntax:

```
type Line = class
    val X1 : float
    val Y1 : float
    val X2 : float
    val Y2 : float

    new (x1, y1, x2, y2) =
        { X1 = x1; Y1 = y1;
            X2 = x2; Y2 = y2}

    member x.Length =
        let sqr x = x * x
```

```
        sqrt(sqr(x.X1 - x.X2) + sqr(x.Y1 - x.Y2) )
end
```

Each `val` defines a field in our object. Unlike other object-oriented languages, F# does not implicitly initialize fields in a class to any value. Instead, F# requires programmers to define a constructor and explicitly initialize each field in their object with a value.

We can perform some post-constructor processing using a `then` block as follows:

```
type Line = class
    val X1 : float
    val Y1 : float
    val X2 : float
    val Y2 : float

    new (x1, y1, x2, y2) as this =
        { X1 = x1; Y1 = y1;
            X2 = x2; Y2 = y2;}
        then
            printfn "Line constructor: {(%F, %F), (%F, %F)}, Line.Length: %F"
                this.X1 this.Y1 this.X2 this.Y2 this.Length

    member x.Length =
        let sqr x = x * x
        sqrt(sqr(x.X1 - x.X2) + sqr(x.Y1 - x.Y2) )
end
```

Notice that we have to add an alias after our constructor, `new (x1, y1, x2, y2) as this`), to access the fields of our object being constructed. Each time we create a `Line` object, the constructor will print to the console. We can demonstrate this using fsi:

```
> let line = new Line(1.0, 1.0, 4.0, 2.5);;

val line : Line

Line constructor: {(1.000000, 1.000000), (4.000000, 2.500000)}, Line.Length: 3.354102
```

Example Using Two Constructors

Since the constructor is defined explicitly, we have the option to provide more than one constructor.

```
open System
open System.Net

type Car = class
    val used : bool
    val owner : string
    val mutable mileage : int

    (* first constructor *)
    new (owner) =
        { used = false;
            owner = owner;
            mileage = 0 }

    (* another constructor *)
    new (owner, mileage) =
        { used = true;
            owner = owner;
            mileage = mileage }
end

let main() =
    let printCar (c : Car) =
        printfn "c.used: %b, c.owner: %s, c.mileage: %i" c.used c.owner c.mileage

    let stevesNewCar = new Car("Steve")
    let bobsUsedCar = new Car("Bob", 83000)
    let printCars() =
        [stevesNewCar; bobsUsedCar] |> Seq.iter printCar

    printfn "\nCars created"
    printCars()

    printfn "\nSteve drives all over the state"
    stevesNewCar.mileage <- stevesNewCar.mileage + 780
    printCars()

    printfn "\nBob commits odometer fraud"
    bobsUsedCar.mileage <- 0
    printCars()

main()
```

This program has the following types:

```
type Car =
  class
    val used: bool
    val owner: string
    val mutable mileage: int
    new : owner:string * mileage:int -> Car
    new : owner:string -> Car
  end
```

Notice that our `val` fields are included in the public interface of our class definition.

This program outputs the following:

```
Cars created
c.used: false, c.owner: Steve, c.mileage: 0
c.used: true, c.owner: Bob, c.mileage: 83000

Steve drives all over the state
c.used: false, c.owner: Steve, c.mileage: 780
c.used: true, c.owner: Bob, c.mileage: 83000

Bob commits odometer fraud
c.used: false, c.owner: Steve, c.mileage: 780
c.used: true, c.owner: Bob, c.mileage: 0
```

Differences Between Implicit and Explicit Syntaxes

As you've probably guessed, the major difference between the two syntaxes is related to the constructor: the explicit syntax forces a programmer to provide explicit constructor(s), whereas the implicit syntax fuses the primary constructor with the class body. However, there are a few other subtle differences:

- The explicit syntax does not allow programmers to declare `let` and `do` bindings.
- Even though `val` fields can be used in the implicit syntax, they must have the attribute `[<DefaultValue>]` and be mutable. It is more convenient to use `let` bindings in this case. Public `member` accessors can be added when they need to be public.
- In the implicit syntax, the primary constructor parameters are visible throughout the whole class body. By using this feature, it is not necessary to write code that copies constructor parameters to instance members.
- While both syntaxes support multiple constructors, when you declare additional constructors with the implicit syntax, they must call the primary constructor. In the explicit syntax all constructors are declared with new() and there is no primary constructor that needs to be referenced from others.

Class with primary (implicit) constructor

```
// The class body acts as a constructor
type Car1(make : string, model : string) = class
    // x.Make and x.Model are property getters
    // (explained later in this chapter)
    // Notice how they can access the
    // constructor parameters directly
    member x.Make = make
    member x.Model = model

    // This is an extra constructor.
    // It calls the primary constructor
    new () = Car1("default make", "default model")
end
```

Class with only explicit constructors

```
type Car2 = class
    // In this case, we need to declare
    // all fields and their types explicitly
    val private make : string
    val private model : string

    // Notice how field access differs
    // from parameter access
    member x.Make = x.make
    member x.Model = x.model

    // Two constructors
    new (make : string, model : string) = {
        make = make
        model = model
    }
    new () = {
        make = "default make"
        model = "default model"
    }
end
```

In general, its up to the programmer to use the implicit or explicit syntax to define classes. However, the implicit syntax is used more often in practice as it tends to result in shorter, more readable class definitions.

Class Inference

F#'s `#light` syntax allows programmers to omit the `class` and `end` keywords in class definitions, a feature commonly referred to as *class inference* or *type kind inference (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785727)*. For example, there is no difference between the following class definitions:

Class Inference

```
type Product(make : string, model : string) =
    member x.Make = make
    member x.Model = model
```

Class Explicit

```
type Car(make : string, model : string) = class
    member x.Make = make
    member x.Model = model
end
```

Both classes compile down to the same bytecode, but the code using class inference allows us to omit a few unnecessary keywords.

Class inference and class explicit styles are considered acceptable. At the very least, when writing F# libraries, don't define half of your classes using class inference and the other half using class explicit style—pick one style and use it consistently for all of your classes throughout your project.

Class Members

Instance and Static Members

There are two types of members you can add to an object:

- Instance members, which can only be called from an object instance created using the `new` keyword.
- Static members, which are not associated with any object instance.

The following class has a static method and an instance method:

```
type SomeClass(prop : int) = class
    member x.Prop = prop
    static member SomeStaticMethod = "This is a static method"
end
```

We invoke a static method using the form `className.methodName`. We invoke instance methods by creating an instance of our class and calling the methods using `classInstance.methodName`. Here is a demonstration in fsi:

```
> SomeClass.SomeStaticMethod;; (* invoking static method *)
val it : string = "This is a static method"

> SomeClass.Prop;; (* doesn't make sense, we haven't created an object instance yet *)

  SomeClass.Prop;; (* doesn't make sense, we haven't created an object instance yet *)
  ^^^^^^^^^^^^^^^

stdin(78,1): error FS0191: property 'Prop' is not static.

> let instance = new SomeClass(5);;

val instance : SomeClass

> instance.Prop;; (* now we have an instance, we can call our instance method *)
val it : int = 5

> instance.SomeStaticMethod;; (* can't invoke static method from instance *)

  instance.SomeStaticMethod;; (* can't invoke static method from instance *)
  ^^^^^^^^^^^^^^^^^^^^^^^^^^

stdin(81,1): error FS0191: property 'SomeStaticMethod' is static.
```

We can, of course, invoke instance methods from objects passed *into* static methods, for example, let's say we add a `Copy` method to our object defined above:

```
type SomeClass(prop : int) = class
    member x.Prop = prop
    static member SomeStaticMethod = "This is a static method"
    static member Copy (source : SomeClass) = new SomeClass(source.Prop)
end
```

We can experiment with this method in fsi:

```
> let instance = new SomeClass(10);;

val instance : SomeClass

> let shallowCopy = instance;; (* copies pointer to another symbol *)

val shallowCopy : SomeClass

> let deepCopy = SomeClass.Copy instance;; (* copies values into a new object *)

val deepCopy : SomeClass

> open System;;

> Object.ReferenceEquals(instance, shallowCopy);;
val it : bool = true

> Object.ReferenceEquals(instance, deepCopy);;
val it : bool = false
```

`Object.ReferenceEquals` is a static method on the `System.Object` class which determines whether two objects instances are the same object. As shown above, our `Copy` method takes an instance of `SomeClass` and accesses its `Prop` property.

When should you use static methods rather than instance methods?

When the designers of the .NET framework were designing the `System.String` class, they had to decide where the `Length` method should go. They had the option of making the property an instance method (`s.Length`) or making it static (`String.GetLength(s)`). The .NET designers chose to make `Length` an instance method because it is an intrinsic property of strings.

On the other hand, the `String` class also has several static methods, including `String.Concat` which takes a list of string and concatenates them all together. Concatenating strings is instance-agnostic, its does not depend on the instance members of any *particular* strings.

The following general principles apply to all OO languages:

- Instance members should be used to access properties intrinsic to an object, such as `stringInstance.Length`.
- Instance methods should be used when they depend on state of a *particular* object instance, such as `stringInstance.Contains`.
- Instance methods should be used when its expected that programmers will want to override the method in a derived class.
- Static methods should not depend on a particular instance of an object, such as `Int32.TryParse`.
- Static methods should return the same value as long as the inputs remain the same.
- Constants, which are values that don't change for any class instance, should be declared as a static members, such as `System.Boolean.TrueString`.

Getters and Setters

Getters and setters are special functions which allow programmers to read and write to members using a convenient syntax. Getters and setters are written using this format:

```
    member alias.PropertyName
        with get() = some-value
        and set(value) = some-assignment
```

Here's a simple example using fsi:

```
> type IntWrapper() = class
    let mutable num = 0

    member x.Num
        with get() = num
        and set(value) = num <- value
end;;

type IntWrapper =
  class
    new : unit -> IntWrapper
    member Num : int
    member Num : int with set
  end

> let wrapper = new IntWrapper();;

val wrapper : IntWrapper

> wrapper.Num;;
val it : int = 0

> wrapper.Num <- 20;;
val it : unit = ()

> wrapper.Num;;
val it : int = 20
```

Getters and setters are used to expose private members to outside world. For example, our `Num` property allows users to read/write to the internal `num` variable. Since getters and setters are glorified functions, we can use them to sanitize input before writing the values to our internal variables. For example, we can modify our `IntWrapper` class to constrain our to values between 0 and 10 by modifying our class as follows:

```
type IntWrapper() = class
    let mutable num = 0

    member x.Num
        with get() = num
        and set(value) =
            if value > 10 || value < 0 then
                raise (new Exception("Values must be between 0 and 10"))
            else
                num <- value
end
```

We can use this class in fsi:

```
> let wrapper = new IntWrapper();;

val wrapper : IntWrapper

> wrapper.Num <- 5;;
val it : unit = ()

> wrapper.Num;;
val it : int = 5

> wrapper.Num <- 20;;
System.Exception: Values must be between 0 and 10
   at FSI_0072.IntWrapper.set_Num(Int32 value)
   at <StartupCode$FSI_0076>.$FSI_0076._main()
stopped due to error
```

Adding Members to Records and Unions

Its just as easy to add members to records and union types as well.

Record example:

```
> type Line =
    { X1 : float; Y1 : float;
      X2 : float; Y2 : float }
    with
        member x.Length =
            let sqr x = x * x
            sqrt(sqr(x.X1 - x.X2) + sqr(x.Y1 - x.Y2))

        member x.ShiftH amount =
            { x with X1 = x.X1 + amount; X2 = x.X2 + amount }

        member x.ShiftV amount =
            { x with Y1 = x.Y1 + amount; Y2 = x.Y2 + amount };;

type Line =
  {X1: float;
   Y1: float;
   X2: float;
   Y2: float;}
  with
    member ShiftH : amount:float -> Line
    member ShiftV : amount:float -> Line
    member Length : float
  end
```

```
> let line = { X1 = 1.0; Y1 = 2.0; X2 = 5.0; Y2 = 4.5 };;

val line : Line

> line.Length;;
val it : float = 4.716990566

> line.ShiftH 10.0;;
val it : Line = {X1 = 11.0;
                 Y1 = 2.0;
                 X2 = 15.0;
                 Y2 = 4.5;}

> line.ShiftV -5.0;;
val it : Line = {X1 = 1.0;
                 Y1 = -3.0;
                 X2 = 5.0;
                 Y2 = -0.5;}
```

Union example

```
> type shape =
    | Circle of float
    | Rectangle of float * float
    | Triangle of float * float
    with
        member x.Area =
            match x with
            | Circle(r) -> Math.PI * r * r
            | Rectangle(b, h) -> b * h
            | Triangle(b, h) -> b * h / 2.0

        member x.Scale value =
            match x with
            | Circle(r) -> Circle(r + value)
            | Rectangle(b, h) -> Rectangle(b + value, h + value)
            | Triangle(b, h) -> Triangle(b + value, h + value);;

type shape =
  | Circle of float
  | Rectangle of float * float
  | Triangle of float * float
  with
    member Scale : value:float -> shape
    member Area : float
  end

> let mycircle = Circle(5.0);;

val mycircle : shape

> mycircle.Area;;
val it : float = 78.53981634

> mycircle.Scale(7.0);;
val it : shape = Circle 12.0
```

Generic classes

Classes which take generic types can be created:

```
type 'a GenericWrapper(initialVal : 'a) = class
    let mutable internalVal = initialVal

    member x.Value
        with get() = internalVal
        and set(value) = internalVal <- value
end
```

We can use this class in FSI as follows:

```
> let intWrapper = new GenericWrapper<_>(5);;

val intWrapper : int GenericWrapper

> intWrapper.Value;;
val it : int = 5

> intWrapper.Value <- 20;;
val it : unit = ()

> intWrapper.Value;;
val it : int = 20

> intWrapper.Value <- 2.0;; (* throws an exception *)

  intWrapper.Value <- 2.0;; (* throws an exception *)
  --------------------^^^^

stdin(156,21): error FS0001: This expression has type
    float
but is here used with type
    int.

> let boolWrapper = new GenericWrapper<_>(true);;

val boolWrapper : bool GenericWrapper

> boolWrapper.Value;;
val it : bool = true
```

Generic classes help programmers generalize classes to operate on multiple different types. They are used in fundamentally the same way as all other generic types already seen in F#, such as Lists, Sets, Maps, and union types.

Pattern Matching Objects

While it's not possible to match objects based on their *structure* in quite the same way that we do for lists and union types, F# allows programmers to match on *types* using the syntax:

```
match arg with
| :? type1 -> expr
| :? type2 -> expr
```

Here's an example which uses type-testing:

```
type Cat() = class
    member x.Meow() = printfn "Meow"
end

type Person(name : string) = class
    member x.Name = name
    member x.SayHello() = printfn "Hi, I'm %s" x.Name
end

type Monkey() = class
    member x.SwingFromTrees() = printfn "swinging from trees"
end

let handlesAnything (o : obj) =
    match o with
    | null -> printfn "<null>"
    | :? Cat as cat -> cat.Meow()
    | :? Person as person -> person.SayHello()
    | _ -> printfn "I don't recognize type '%s'" (o.GetType().Name)

let main() =
    let cat = new Cat()
    let bob = new Person("Bob")
    let bill = new Person("Bill")
    let phrase = "Hello world!"
    let monkey = new Monkey()

    handlesAnything cat
    handlesAnything bob
    handlesAnything bill
    handlesAnything phrase
    handlesAnything monkey
    handlesAnything null

main()
```

This program outputs:

```
Meow
Hi, I'm Bob
Hi, I'm Bill
I don't recognize type 'String'
I don't recognize type 'Monkey'
<null>
```

Inheritance

F# : Inheritance

Many object-oriented languages use **inheritance** extensively in the .NET BCL to construct class hierarchies.

Subclasses

A subclass is, in the simplest terms, a class derived from a class which has already been defined. A subclass inherits its members from a base class in addition to adding its own members. A subclass is defined using the `inherit` keyword as shown below:

```
type Person(name) =
    member x.Name = name
    member x.Greet() = printfn "Hi, I'm %s" x.Name

type Student(name, studentID : int) =
    inherit Person(name)

    let mutable _GPA = 0.0

    member x.StudentID = studentID
    member x.GPA
        with get() = _GPA
        and set value = _GPA <- value

type Worker(name, employer : string) =
    inherit Person(name)

    let mutable _salary = 0.0

    member x.Salary
        with get() = _salary
```

```
        and set value = _salary <- value

    member x.Employer = employer
```

Our simple class hierarchy looks like this:

```
System.Object (* All classes descend from  *)
 - Person
   - Student
   - Worker
```

The `Student` and `Worker` subclasses both inherit the `Name` and `Greet` methods from the `Person` base class. This can be demonstrated in fsi:

```
> let somePerson, someStudent, someWorker =
    new Person("Juliet"), new Student("Monique", 123456), new Worker("Carla", "Awesome Hair Salon");;

val someWorker : Worker
val someStudent : Student
val somePerson : Person

> somePerson.Name, someStudent.Name, someWorker.Name;;
val it : string * string * string = ("Juliet", "Monique", "Carla")

> someStudent.StudentID;;
val it : int = 123456

> someWorker.Employer;;
val it : string = "Awesome Hair Salon"

> someWorker.ToString();; (* ToString method inherited from System.Object *)
val it : string = "FSI_0002+Worker"
```

.NET's object model supports *single-class inheritance*, meaning that a subclass is limited to one base class. In other words, its not possible to create a class which derives from `Student` and `Employee` simultaneously.

Overriding Methods

Occasionally, you may want a derived class to change the default behavior of methods inherited from the base class. For example, the output of the `.ToString()` method above isn't very useful. We can override that behavior with a different implementation using the `override`:

```
type Person(name) =
    member x.Name = name
    member x.Greet() = printfn "Hi, I'm %s" x.Name

    override x.ToString() = x.Name    (* The ToString() method is inherited from System.Object *)
```

We've overridden the default implementation of the `ToString()` method, causing it to print out a person's name.

Methods in F# are not overridable by default. If you expect users will want to override methods in a derived class, you have to declare your method as overridable using the `abstract` and `default` keywords as follows:

```
type Person(name) =
    member x.Name = name

    abstract Greet : unit -> unit
    default x.Greet() = printfn "Hi, I'm %s" x.Name

type Quebecois(name) =
    inherit Person(name)

    override x.Greet() = printfn "Bonjour, je m'appelle %s, eh." x.Name
```

Our class `Person` provides a `Greet` method which may be overridden in derived classes. Here's an example of these two classes in fsi:

```
> let terrance, phillip = new Person("Terrance"), new Quebecois("Phillip");;

val terrance : Person
val phillip : Quebecois

> terrance.Greet();;
Hi, I'm Terrance
val it : unit = ()

> phillip.Greet();;
Bonjour, je m'appelle Phillip, eh.
```

Abstract Classes

An abstract class is one which provides an incomplete implementation of an object, and requires a programmer to create subclasses of the abstract class to fill in the rest of the implementation. For example, consider the following:

```
[<AbstractClass>]
type Shape(position : Point) =
    member x.Position = position
    override x.ToString() =
        sprintf "position = {%i, %i}, area = %f" position.X position.Y (x.Area())

    abstract member Draw : unit -> unit
    abstract member Area : unit -> float
```

The first thing you'll notice is the `AbstractClass` attribute, which tells the compiler that our class has some abstract members. Additionally, you notice two abstract members, `Draw` and `Area` don't have an implementation, only a type signature.

We can't create an instance of `Shape` because the class hasn't been fully implemented. Instead, we have to derive from `Shape` and override the `Draw` and `Area` methods with a concrete implementation:

```
type Circle(position : Point, radius : float) =
    inherit Shape(position)

    member x.Radius = radius
    override x.Draw() = printfn "(Circle)"
    override x.Area() = Math.PI * radius * radius

type Rectangle(position : Point, width : float, height : float) =
    inherit Shape(position)

    member x.Width = width
    member x.Height = height
    override x.Draw() = printfn "(Rectangle)"
    override x.Area() = width * height

type Square(position : Point, width : float) =
    inherit Shape(position)

    member x.Width = width
    member x.ToRectangle() = new Rectangle(position, width, width)
    override x.Draw() = printfn "(Square)"
    override x.Area() = width * width

type Triangle(position : Point, sideA : float, sideB : float, sideC : float) =
    inherit Shape(position)

    member x.SideA = sideA
    member x.SideB = sideB
    member x.SideC = sideC

    override x.Draw() = printfn "(Triangle)"
    override x.Area() =
        (* Heron's formula *)
        let a, b, c = sideA, sideB, sideC
        let s = (a + b + c) / 2.0
        Math.Sqrt(s * (s - a) * (s - b) * (s - c) )
```

Now we have several different implementations of the `Shape` class. We can experiment with these in fsi:

```
> let position = { X = 0; Y = 0 };;

val position : Point

> let circle, rectangle, square, triangle =
    new Circle(position, 5.0),
    new Rectangle(position, 2.0, 7.0),
    new Square(position, 10.0),
    new Triangle(position, 3.0, 4.0, 5.0);;

val triangle : Triangle
val square : Square
val rectangle : Rectangle
val circle : Circle

> circle.ToString();;
val it : string = "Circle, position = {0, 0}, area = 78.539816"

> triangle.ToString();;
val it : string = "Triangle, position = {0, 0}, area = 6.000000"

> square.Width;;
val it : float = 10.0

> square.ToRectangle().ToString();;
val it : string = "Rectangle, position = {0, 0}, area = 100.000000"

> rectangle.Height, rectangle.Width;;
val it : float * float = (7.0, 2.0)
```

Working With Subclasses

Up-casting and Down-casting

A *type cast* is an operation which changes the type of an object from one type to another. This is not the same as a map function, because a type cast does not return an instance of a new object, it returns the same instance of an object with a different type.

For example, let's say `B` is a subclass of `A`. If we have an instance of `B`, we are able to cast as an instance of `A`. Since `A` is upward in the class hierarchy, we call this an up-cast. We use the `:>` operator to perform upcasts:

```
> let regularString = "Hello world";;

val regularString : string

> let upcastString = "Hello world" :> obj;;

val upcastString : obj

> regularString.ToString();;
val it : string = "Hello world"

> regularString.Length;;
val it : int = 11
```

```
> upcastString.ToString();; (* type obj has a .ToString method *)
val it : string = "Hello world"

> upcastString.Length;; (* however, obj does not have Length method *)

  upcastString.Length;; (* however, obj does not have Length method *)
  ------------^^^^^^^

stdin(24,14): error FS0039: The field, constructor or member 'Length' is not defined.
```

Up-casting is considered "safe", because a derived class is guaranteed to have all of the same members as an ancestor class. We can, if necessary, go in the opposite direction: we can down-cast from an ancestor class to a derived class using the `:?>` operator:

```
> let intAsObj = 20 :> obj;;

val intAsObj : obj

> intAsObj, intAsObj.ToString();;
val it : obj * string = (20, "20")

> let intDownCast = intAsObj :?> int;;

val intDownCast : int

> intDownCast, intDownCast.ToString();;
val it : int * string = (20, "20")

> let stringDownCast = intAsObj :?> string;; (* boom! *)

val stringDownCast : string

System.InvalidCastException: Unable to cast object of type 'System.Int32' to type 'System.String'.
   at <StartupCode$FSI_0067>.$FSI_0067._main()
stopped due to error
```

Since `intAsObj` holds an `int` boxed as an `obj`, we can downcast to an `int` as needed. However, we cannot downcast to a `string` because it is an incompatible type. Down-casting is considered "unsafe" because the error isn't detectable by the type-checker, so an error with a down-cast always results in a runtime exception.

Up-casting example

```
open System

type Point = { X : int; Y : int }

[<AbstractClass>]
type Shape() =
    override x.ToString() =
        sprintf "%s, area = %f" (x.GetType().Name) (x.Area())

    abstract member Draw : unit -> unit
    abstract member Area : unit -> float

type Circle(radius : float) =
    inherit Shape()

    member x.Radius = radius
    override x.Draw() = printfn "(Circle)"
    override x.Area() = Math.PI * radius * radius

type Rectangle(width : float, height : float) =
    inherit Shape()

    member x.Width = width
    member x.Height = height
    override x.Draw() = printfn "(Rectangle)"
    override x.Area() = width * height

type Square(width : float) =
    inherit Shape()

    member x.Width = width
    member x.ToRectangle() = new Rectangle(width, width)
    override x.Draw() = printfn "(Square)"
    override x.Area() = width * width

type Triangle(sideA : float, sideB : float, sideC : float) =
    inherit Shape()

    member x.SideA = sideA
    member x.SideB = sideB
    member x.SideC = sideC

    override x.Draw() = printfn "(Triangle)"
    override x.Area() =
        (* Heron's formula *)
        let a, b, c = sideA, sideB, sideC
        let s = (a + b + c) / 2.0
        Math.Sqrt(s * (s - a) * (s - b) * (s - c) )

let shapes =
        [(new Circle(5.0) :> Shape);
            (new Circle(12.0) :> Shape);
            (new Square(10.5) :> Shape);
            (new Triangle(3.0, 4.0, 5.0) :> Shape);
            (new Rectangle(5.0, 2.0) :> Shape)]
        (* Notice we have to cast each object as a Shape *)

let main() =
    shapes
    |> Seq.iter (fun x -> printfn "x.ToString: %s" (x.ToString()) )

main()
```

This program has the following types:

```
type Point =
  {X: int;
   Y: int;}

type Shape =
  class
    abstract member Area : unit -> float
    abstract member Draw : unit -> unit
    new : unit -> Shape
    override ToString : unit -> string
  end

type Circle =
  class
    inherit Shape
    new : radius:float -> Circle
    override Area : unit -> float
    override Draw : unit -> unit
    member Radius : float
  end

type Rectangle =
  class
    inherit Shape
    new : width:float * height:float -> Rectangle
    override Area : unit -> float
    override Draw : unit -> unit
    member Height : float
    member Width : float
  end

type Square =
  class
    inherit Shape
    new : width:float -> Square
    override Area : unit -> float
    override Draw : unit -> unit
    member ToRectangle : unit -> Rectangle
    member Width : float
  end

type Triangle =
  class
    inherit Shape
    new : sideA:float * sideB:float * sideC:float -> Triangle
    override Area : unit -> float
    override Draw : unit -> unit
    member SideA : float
    member SideB : float
    member SideC : float
  end

val shapes : Shape list
```

This program outputs:

```
x.ToString: Circle, area = 78.539816
x.ToString: Circle, area = 452.389342
x.ToString: Square, area = 110.250000
x.ToString: Triangle, area = 6.000000
x.ToString: Rectangle, area = 10.000000
```

Public, Private, and Protected Members

Interfaces

F# : Interfaces

An object's **interface** refers to all of the public members and functions that a function exposes to consumers of the object. For example, take the following:

```
type Monkey(name : string, birthday : DateTime) =
    let mutable _birthday = birthday
    let mutable _lastEaten = DateTime.Now
    let mutable _foodsEaten = [] : string list

    member this.Speak() = printfn "Ook ook!"
    member this.Name = name
    member this.Birthday
        with get() = _birthday
        and set(value) = _birthday <- value

    member internal this.UpdateFoodsEaten(food) = _foodsEaten <- food :: _foodsEaten
    member internal this.ResetLastEaten() = _lastEaten <- DateTime.Now
    member this.IsHungry = (DateTime.Now - _lastEaten).TotalSeconds >= 5.0
    member this.GetFoodsEaten() = _lastEaten
    member this.Feed(food) =
        this.UpdateFoodsEaten(food)
        this.ResetLastEaten()
        this.Speak()
```

This class contains several public, private, and internal members. However, consumers of this class can only access the public members; when a consumer uses this class, they see the following interface:

```
type Monkey =
  class
    new : name:string * birthday:DateTime -> Monkey
    member Feed : food:string -> unit
    member GetFoodsEaten : unit -> DateTime
    member Speak : unit -> unit
    member Birthday : DateTime
    member IsHungry : bool
    member Name : string
    member Birthday : DateTime with set
  end
```

Notice the `_birthday`, `_lastEaten`, `_foodsEaten`, `UpdateFoodsEaten`, and `ResetLastEaten` members are inaccessible to the outside world, so they are not part of this object's public interface.

All interfaces you've seen so far have been intrinsically tied to a specific object. However, F# and many other OO languages allow users to define interfaces as stand-alone types, allowing us to effectively *separate an object's interface from its implementation*.

Defining Interfaces

According to the F# specification (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785736), interfaces are defined with the following syntax:

```
type type-name =
   interface
       inherits-decl
       member-defns
   end
```

> **Note:** The interface/end tokens can be omitted when using the #light syntax option, in which case Type Kind Inference (§10.1 (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785727)) is used to determine the kind of the type. The presence of any non-abstract members or constructors means a type is not an interface type.

For example:

```
type ILifeForm = (* .NET convention recommends the prefix 'I' on all interfaces *)
    abstract Name : string
    abstract Speak : unit -> unit
    abstract Eat : unit -> unit
```

Using Interfaces

Since they only define a set of public method signatures, users need to create an object to *implement* the interface. Here are three classes which implement the ILifeForm interface in fsi:

```
> type ILifeForm =
    abstract Name : string
    abstract Speak : unit -> unit
    abstract Eat : unit -> unit

type Dog(name : string, age : int) =
    member this.Age = age

    interface ILifeForm with
        member this.Name = name
        member this.Speak() = printfn "Woof!"
        member this.Eat() = printfn "Yum, doggy biscuits!"

type Monkey(weight : float) =
    let mutable _weight = weight

    member this.Weight
        with get() = _weight
        and set(value) = _weight <- value

    interface ILifeForm with
        member this.Name = "Monkey!!!"
        member this.Speak() = printfn "Ook ook"
        member this.Eat() = printfn "Bananas!"

type Ninja() =
    interface ILifeForm with
        member this.Name = "Ninjas have no name"
        member this.Speak() = printfn "Ninjas are silent, deadly killers"
        member this.Eat() =
            printfn "Ninjas don't eat, they wail on guitars because they're totally sweet";;

type ILifeForm =
  interface
    abstract member Eat : unit -> unit
    abstract member Speak : unit -> unit
    abstract member Name : string
  end
type Dog =
  class
    interface ILifeForm
    new : name:string * age:int -> Dog
    member Age : int
  end
type Monkey =
  class
    interface ILifeForm
    new : weight:float -> Monkey
    member Weight : float
    member Weight : float with set
  end
type Ninja =
```

```
  class
    interface ILifeForm
    new : unit -> Ninja
  end
```

Typically, we call an interface an *abstraction*, and any class which implements the interface as a *concrete implementation*. In the example above, `ILifeForm` is an abstraction, whereas `Dog`, `Monkey`, and `Ninja` are concrete implementations.

Its worth noting that interfaces only define instance members signatures on objects. In other words, they cannot define static member signatures or constructor signatures.

What are interfaces used for and how to use them?

Interfaces are a mystery to newbie programmers (after all, what's the point of creating a type with no implementation?), however they are essential to many object-oriented programming techniques. Interfaces allow programmers to generalize functions to all classes which implement a particular functionality, which is described by the interface, even if those classes don't necessarily descend from one another.

For example, the `Dog`, `Monkey`, and `Ninja` classes defined above contain behavior that is shared, which we defined in the `ILifeForm` interface. As the code shows, *how* the individual classes talk or eat is not defined, but for each class that implements the interface we know that they can eat and speak and have a name. Now we can write a method that accepts just the interface `ILifeForm` and we need not worry about how it is implemented, if it is implemented (it always is, the compiler takes care of that) or what type of object it really is. Any other class that implements the same interface, regardless of its other methods, is then automatically supported by this method as well.

```
let letsEat (lifeForm: ILifeForm) = lifeForm.Eat()
```

Note that in F#, interfaces are implemented explicitly, whereas in C# they are often implemented implicitly. As a consequence, to call a function or method that expects an interface, you have to make an explicit cast:

```
let myDog = Dog()
letsEat (myDog :> ILifeForm)
```

You can simplify this by letting the compiler help find the proper interface by using the _ placeholder:

```
let myDog = Dog()
letsEat (myDog :> _)
```

Implementing Interfaces with Object Expressions

Interfaces are extremely useful for sharing snippets of implementation logic between other classes, however it can be very cumbersome to define and implement a new class for ad hoc interfaces. Object expressions allow users to implement interfaces on anonymous classes using the following syntax (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785612):

```
{ new ty0 [ args-expr ] [ as base-ident ] [ with
      val-or-member-defns end ]

  interface ty1 with [
      val-or-member-defns1
   end ]

  …

  interface tyn with [
      val-or-member-defnsn
  end ] }
```

Using a concrete example, the .NET BCL has a method called `System.Array.Sort<T>(T array, IComparer<T>)` (http://msdn.microsoft.com/en-us/library/bzw8611x.aspx), where `IComparer<T>` (http://msdn.microsoft.com/en-us/library/6zzyats9.aspx) exposes a single method called `Compare`. Let's say we wanted to sort an array on an ad hoc basis using this method; rather than litter our code with one-time use classes, we can use object expressions to define anonymous classes on the fly:

```
> open System
open System.Collections.Generic

type person = { name : string; age : int }

let people =
    [|{ name = "Larry"; age = 20 };
      { name = "Moe"; age = 30 };
      { name = "Curly"; age = 25 } |]

let sortAndPrint msg items (comparer : System.Collections.Generic.IComparer<person>) =
    Array.Sort(items, comparer)
    printf "%s: " msg
    Seq.iter (fun x -> printf "(%s, %i) " x.name x.age) items
    printfn ""

(* sorting by age *)
sortAndPrint "age" people { new IComparer<person> with member this.Compare(x, y) = x.age.CompareTo(y.age) }

(* sorting by name *)
sortAndPrint "name" people { new IComparer<person> with member this.Compare(x, y) = x.name.CompareTo(y.name) }

(* sorting by name descending *)
sortAndPrint "name desc" people { new IComparer<person> with member this.Compare(x, y) = y.name.CompareTo(x.name) };;

type person =
  { name: string;
```

```
   age: int; }
val people : person array
val sortAndPrint : string -> person array -> IComparer<person> -> unit

age: (Larry, 20) (Curly, 25) (Moe, 30)
name: (Curly, 25) (Larry, 20) (Moe, 30)
name desc: (Moe, 30) (Larry, 20) (Curly, 25)
```

Implementing Multiple Interfaces

Unlike inheritance, its possible to implement multiple interfaces:

```
open System

type Person(name : string, age : int) =
    member this.Name = name
    member this.Age = age

    (* IComparable is used for ordering instances *)
    interface IComparable<Person> with
        member this.CompareTo(other) =
            (* sorts by name, then age *)
            match this.Name.CompareTo(other.Name) with
            | 0 -> this.Age.CompareTo(other.Age)
            | n -> n

    (* Used for comparing this type against other types *)
    interface IEquatable<string> with
        member this.Equals(othername) = this.Name.Equals(othername)
```

Its just as easy to implement multiple interfaces in object expressions as well.

Interface Hierarchies

Interfaces can extend other interfaces in a kind of interface hierarchy. For example:

```
type ILifeForm =
    abstract member location : System.Drawing.Point

type 'a IAnimal =   (* interface with generic type parameter *)
    inherit ILifeForm
    inherit System.IComparable<'a>
    abstract member speak : unit -> unit

type IFeline =
    inherit IAnimal<IFeline>
    abstract member purr : unit -> unit
```

When users create a concrete implementation of `IFeline`, they are required to provide implementations for all of the methods defined in the `IAnimal`, `IComparable`, and `ILifeForm` interfaces.

> **Note:** Interface hierarchies are occasionally useful, however deep, complicated hierarchies can be cumbersome to work with.

Examples

Generalizing a function to many classes

```
open System
type ILifeForm =
    abstract Name : string
    abstract Speak : unit -> unit
    abstract Eat : unit -> unit

type Dog(name : string, age : int) =
    member this.Age = age

    interface ILifeForm with
        member this.Name = name
        member this.Speak() = printfn "Woof!"
        member this.Eat() = printfn "Yum, doggy biscuits!"

type Monkey(weight : float) =
    let mutable _weight = weight

    member this.Weight
        with get() = _weight
        and set(value) = _weight <- value

    interface ILifeForm with
        member this.Name = "Monkey!!!"
        member this.Speak() = printfn "Ook ook"
        member this.Eat() = printfn "Bananas!"

type Ninja() =
    interface ILifeForm with
        member this.Name = "Ninjas have no name"
        member this.Speak() = printfn "Ninjas are silent, deadly killers"
        member this.Eat() =
            printfn "Ninjas don't eat, they wail on guitars because they're totally sweet"

let lifeforms =
    [(new Dog("Fido", 7) :> ILifeForm);
     (new Monkey(500.0) :> ILifeForm);
     (new Ninja() :> ILifeForm)]

let handleLifeForm (x : ILifeForm) =
```

```
    printfn "Handling lifeform '%s'" x.Name
    x.Speak()
    x.Eat()
    printfn ""

let main() =
    printfn "Processing...\n"
    lifeforms |> Seq.iter handleLifeForm
    printfn "Done."

main()
```

This program has the following types:

```
type ILifeForm =
  interface
    abstract member Eat : unit -> unit
    abstract member Speak : unit -> unit
    abstract member Name : string
  end

type Dog =
  class
    interface ILifeForm
    new : name:string * age:int -> Dog
    member Age : int
  end

type Monkey =
  class
    interface ILifeForm
    new : weight:float -> Monkey
    member Weight : float
    member Weight : float with set
  end

type Ninja =
  class
    interface ILifeForm
    new : unit -> Ninja
  end

val lifeforms : ILifeForm list
val handleLifeForm : ILifeForm -> unit
val main : unit -> unit
```

This program outputs the following:

```
Processing...

Handling lifeform 'Fido'
Woof!
Yum, doggy biscuits!

Handling lifeform 'Monkey!!!'
Ook ook
Bananas!

Handling lifeform 'Ninjas have no name'
Ninjas are silent, deadly killers
Ninjas don't eat, they wail on guitars because they're totally sweet

Done.
```

Using interfaces in generic type definitions

We can constrain generic types in class and function definitions to particular interfaces. For example, let's say that we wanted to create a binary tree which satisfies the following property: each node in a binary tree has two children, `left` and `right`, where all of the child nodes in `left` are less than all of its parent nodes, and all of the child nodes in `right` are greater than all of its parent nodes.

We can implement a binary tree with these properties defining a binary tree which constrains our tree to the `IComparable<T>` interface.

> **Note:** .NET has a number of interfaces defined in the BCL, including the very important `IComparable<T> interface (http://msdn.microsoft.com/en-us/library/4d7sx9hd.aspx)`. IComparable exposes a single method, objectInstance.CompareTo(otherInstance) (http://msdn.microsoft.com/en-us/library/43hc6wht.aspx), which should return 1, -1, or 0 when the `objectInstance` is greater than, less than, or equal to `otherInstance` respectively. Many classes in the .NET framework already implement IComparable, including all of the numeric data types, strings, and datetimes.

For example, using fsi:

```
> open System

type tree<'a> when 'a :> IComparable<'a> =
    | Nil
    | Node of 'a * 'a tree * 'a tree

let rec insert (x : #IComparable<'a>) = function
    | Nil -> Node(x, Nil, Nil)
    | Node(y, l, r) as node ->
        if x.CompareTo(y) = 0 then node
        elif x.CompareTo(y) = -1 then Node(y, insert x l, r)
        else Node(y, l, insert x r)

let rec contains (x : #IComparable<'a>) = function
    | Nil -> false
    | Node(y, l, r) as node ->
        if x.CompareTo(y) = 0 then true
        elif x.CompareTo(y) = -1 then contains x l
        else contains x r;;
```

```
type tree<'a> when 'a :> IComparable<'a>> =
  | Nil
  | Node of 'a * tree<'a> * tree<'a>
val insert : 'a -> tree<'a> -> tree<'a> when 'a :> IComparable<'a>
val contains : #IComparable<'a> -> tree<'a> -> bool when 'a :> IComparable<'a>

> let x =
    let rnd = new Random()
    [ for a in 1 .. 10 -> rnd.Next(1, 100) ]
    |> Seq.fold (fun acc x -> insert x acc) Nil;;

val x : tree<int>

> x;;
val it : tree<int>
= Node
    (25,Node (20,Node (6,Nil,Nil),Nil),
     Node
       (90,
        Node
          (86,Node (65,Node (50,Node (39,Node (32,Nil,Nil),Nil),Nil),Nil),Nil),
        Nil))

> contains 39 x;;
val it : bool = true

> contains 55 x;;
val it : bool = false
```

Simple dependency injection

Dependency injection refers to the process of supplying an external dependency to a software component. For example, let's say we had a class which, in the event of an error, sends an email to the network administrator, we might write some code like this:

```
type Processor() =
    (* ... *)
    member this.Process items =
        try
            (* do stuff with items *)
        with
            | err -> (new Emailer()).SendMsg("admin@company.com", "Error! " + err.Message)
```

The `Process` method creates an instance of `Emailer`, so we can say that the `Processor` class *depends* on the `Emailer` class.

Let's say we're testing our `Processor` class, and we don't want to be sending emails to the network admin all the time. Rather than comment out the lines of code we don't want to run while we test, its much easier to substitute the `Emailer` dependency with a dummy class instead. We can achieve that by passing in our dependency through the constructor:

```
type IFailureNotifier =
    abstract Notify : string -> unit

type Processor(notifier : IFailureNotifier) =
    (* ... *)
    member this.Process items =
        try
            // do stuff with items
        with
            | err -> notifier.Notify(err.Message)

(* concrete implementations of IFailureNotifier *)

type EmailNotifier() =
    interface IFailureNotifier with
        member Notify(msg) = (new Emailer()).SendMsg("admin@company.com", "Error! " + msg)

type DummyNotifier() =
    interface IFailureNotifier with
        member Notify(msg) = () // swallow message

type LogfileNotifier(filename : string) =
    interface IFailureNotifer with
        member Notify(msg) = System.IO.File.AppendAllText(filename, msg)
```

Now, we create a processor and pass in the kind of FailureNotifier we're interested in. In test environments, we'd use `new Processor(new DummyNotifier());` in production, we'd use `new Processor(new EmailNotifier())` or `new Processor(new LogfileNotifier(@"C:\log.txt"))`.

To demonstrate dependency injection using a somewhat contrived example, the following code in fsi shows how to hot swap one interface implementation with another:

```
> #time;;

--> Timing now on

> type IAddStrategy =
    abstract add : int -> int -> int

type DefaultAdder() =
    interface IAddStrategy with
        member this.add x y = x + y

type SlowAdder() =
    interface IAddStrategy with
        member this.add x y =
            let rec loop acc = function
                | 0 -> acc
                | n -> loop (acc + 1) (n - 1)
            loop x y
```

```
type OffByOneAdder() =
    interface IAddStrategy with
        member this.add x y = x + y - 1

type SwappableAdder(adder : IAddStrategy) =
    let mutable _adder = adder
    member this.Adder
        with get() = _adder
        and set(value) = _adder <- value

    member this.Add x y = this.Adder.add x y;;

type IAddStrategy =
  interface
    abstract member add : int -> (int -> int)
  end
type DefaultAdder =
  class
    interface IAddStrategy
    new : unit -> DefaultAdder
  end
type SlowAdder =
  class
    interface IAddStrategy
    new : unit -> SlowAdder
  end
type OffByOneAdder =
  class
    interface IAddStrategy
    new : unit -> OffByOneAdder
  end
type SwappableAdder =
  class
    new : adder:IAddStrategy -> SwappableAdder
    member Add : x:int -> (int -> int)
    member Adder : IAddStrategy
    member Adder : IAddStrategy with set
  end

Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0

> let myAdder = new SwappableAdder(new DefaultAdder());;

val myAdder : SwappableAdder

Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0

> myAdder.Add 10 1000000000;;
Real: 00:00:00.001, CPU: 00:00:00.015, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 1000000010

> myAdder.Adder <- new SlowAdder();;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()

> myAdder.Add 10 1000000000;;
Real: 00:00:01.085, CPU: 00:00:01.078, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 1000000010

> myAdder.Adder <- new OffByOneAdder();;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()

> myAdder.Add 10 1000000000;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : int = 1000000009
```

Events

F# : Events

Events allow objects to communicate with one another through a kind of synchronous message passing. Events are simply hooks to other functions: objects register callback functions to an event, and these callbacks will be executed when (and if) the event is triggered by some object.

For example, let's say we have a clickable button which exposed an event called `Click`. We can register a block of code, something like `fun () -> printfn "I've been clicked!"`, to the button's click event. When the click event is triggered, it will execute the block of code we've registered. If we wanted to, we could register an indefinite number of callback functions to the click event—the button doesn't care what code is trigged by the callbacks or how many callback functions are registered to its click event, it blindly executes whatever functions are hooked to its click event.

Event-driven programming is natural in GUI code, as GUIs tend to consist of controls which react and respond to user input. Events are, of course, useful in non-GUI applications as well. For example, if we have an object with mutable properties, we may want to notify another object when those properties change.

Defining Events

Events are created and used through F#'s Event class (http://msdn.microsoft.com/en-us/library/ee370608(VS.100).aspx). To create an event, use the `Event` constructor as follows:

```
type Person(name : string) =
    let mutable _name = name;
    let nameChanged = new Event<string>()

    member this.Name
```

```
        with get() = _name
        and set(value) = _name <- value
```

To allow listeners to hook onto our event, we need to expose the `nameChanged` field as a public member using the event's `Publish` property:

```
type Person(name : string) =
    let mutable _name = name;
    let nameChanged = new Event<unit>() (* creates event *)

    member this.NameChanged = nameChanged.Publish (* exposed event handler *)

    member this.Name
        with get() = _name
        and set(value) =
            _name <- value
            nameChanged.Trigger() (* invokes event handler *)
```

Now, any object can listen to the changes on the person method. By convention and Microsoft's recommendation (http://msdn.microsoft.com/en-us/library/ms229012.aspx), events are usually named *`Verb`* or *`VerbPhrase`*, as well as adding tenses like *`Verbed`* and *`Verbing`* to indicate post- and pre-events.

Adding Callbacks to Event Handlers

Its very easy to add callbacks to event handlers. Each event handler has the type `IEvent<'T>` which exposes several methods:

`val Add : event:('T -> unit) -> unit`

> Connect a listener function to the event. The listener will be invoked when the event is fired.

`val AddHandler : 'del -> unit`

> Connect a handler delegate object to the event. A handler can be later removed using RemoveHandler. The listener will be invoked when the event is fired.

`val RemoveHandler : 'del -> unit`

> Remove a listener delegate from an event listener store.

Here's an example program:

```
type Person(name : string) =
    let mutable _name = name;
    let nameChanged = new Event<unit>() (* creates event *)

    member this.NameChanged = nameChanged.Publish (* exposed event handler *)

    member this.Name
        with get() = _name
        and set(value) =
            _name <- value
            nameChanged.Trigger() (* invokes event handler *)

let p = new Person("Bob")
p.NameChanged.Add(fun () -> printfn "-- Name changed! New name: %s" p.Name)

printfn "Event handling is easy"
p.Name <- "Joe"

printfn "It handily decouples objects from one another"
p.Name <- "Moe"

p.NameChanged.Add(fun () -> printfn "-- Another handler attached to NameChanged!")

printfn "It's also causes programs behave non-deterministically."
p.Name <- "Bo"

printfn "The function NameChanged is invoked effortlessly."
```

This program outputs the following:

```
Event handling is easy
-- Name changed! New name: Joe
It handily decouples objects from one another
-- Name changed! New name: Moe
It's also causes programs behave non-deterministically.
-- Name changed! New name: Bo
-- Another handler attached to NameChanged!
The function NameChanged is invoked effortlessly.
```

> **Note:** When multiple callbacks are connected to a single event, they are executed in the order they are added. However, in practice, you should **not** write code with the expectation that events will trigger in a particular order, as doing so can introduce complex dependencies between functions. Event-driven programming is often non-deterministic and fundamentally stateful, which can occasionally be at odds with the spirit of functional programming. Its best to write callback functions which do not modify state, and do not depend on the invocation of any prior events.

Working with EventHandlers Explicitly

Adding and Removing Event Handlers

The code above demonstrates how to use the `IEvent<'T>.add` method. However, occasionally we need to remove callbacks. To do so, we need to work with the `IEvent<'T>.AddHandler` and `IEvent<'T>.RemoveHandler` methods, as well as .NET's built-in System.Delegate (http://msdn.microsoft.com/en-us/library/system.delegate.aspx) type.

The function `person.NameChanged.AddHandler` has the type `val AddHandler : Handler<'T> -> unit`, where `Handler<'T>` inherits from `System.Delegate`. We can create an instance of `Handler` as follows:

```
type Person(name : string) =
    let mutable _name = name;
    let nameChanged = new Event<unit>() (* creates event *)

    member this.NameChanged = nameChanged.Publish (* exposed event handler *)

    member this.Name
        with get() = _name
        and set(value) =
            _name <- value
            nameChanged.Trigger() (* invokes event handler *)


let p = new Person("Bob")

let person_NameChanged =
    new Handler<unit>(fun sender eventargs -> printfn "-- Name changed! New name: %s" p.Name)

p.NameChanged.AddHandler(person_NameChanged)

printfn "Event handling is easy"
p.Name <- "Joe"

printfn "It handily decouples objects from one another"
p.Name <- "Moe"

p.NameChanged.RemoveHandler(person_NameChanged)
p.NameChanged.Add(fun () -> printfn "-- Another handler attached to NameChanged!")

printfn "It's also causes programs behave non-deterministically."
p.Name <- "Bo"

printfn "The function NameChanged is invoked effortlessly."
```

This program outputs the following:

```
Event handling is easy
-- Name changed! New name: Joe
It handily decouples objects from one another
-- Name changed! New name: Moe
It's also causes programs behave non-deterministically.
-- Another handler attached to NameChanged!
The function NameChanged is invoked effortlessly.
```

Defining New Delegate Types

F#'s event handling model is a little different from the rest of .NET. If we want to expose F# events to different languages like C# or VB.NET, we can define a custom delegate type which compiles to a .NET delegate using the `delegate` keyword, for example:

```
type NameChangingEventArgs(oldName : string, newName : string) =
    inherit System.EventArgs()

    member this.OldName = oldName
    member this.NewName = newName

type NameChangingDelegate = delegate of obj * NameChangingEventArgs -> unit
```

The convention `obj * NameChangingEventArgs` corresponds to the .NET naming guidelines which recommend that all events have the type `val eventName : (sender : obj * e : #EventArgs) -> unit`.

Use existing .NET WPF Event and Delegate Types

Try using existing .NET WPF Event and Delegate, example, ClickEvent and RoutedEventHandler. Create F# Windows Application .NET project with referring these libraries (PresentationCore PresentationFramework System.Xaml WindowsBase). The program will display a button in a window. Clicking the button will display the button's content as string.

```
open System.Windows
open System.Windows.Controls
open System.Windows.Input
open System
[<EntryPoint>] [<STAThread>]              // STAThread is Single-Threading-Apartment which is required by WPF
let main argv =
    let b = new Button(Content="Button")  // b is a Button with "Button" as content
    let f(sender:obj)(e:RoutedEventArgs) = // (#3) f is a fun going to handle the Button.ClickEvent
                                           //   f signature must be curried, not tuple as govened by Delegate-RoutedEventHandler.
                                           //   that means f(sender:obj,e:RoutedEventArgs) will not work.
        let b = sender:?>Button            // sender will have Button-type.  Convert it to Button into b.
        MessageBox.Show(b.Content:?>string) // Retrieve the content of b which is obj.
                                           //    Convert it to string and display by <code>Messagebox.Show</code>
            |> ignore                      // ignore the return because f-signature requires: obj->RoutedEventArgs->unit


    let d = new RoutedEventHandler(f)      // (#2) d will have type-RoutedEventHandler,
                                           //      RoutedEventHandler is a kind of delegate to handle Button.ClickEvent.
                                           //      The f must have signature governed by RoutedEventHandler.
    b.AddHandler(Button.ClickEvent,d)      // (#1) attach a RountedEventHandler-d for Button.ClickEvent
    let w = new Window(Visibility=Visibility.Visible,Content=b)  // create a window-w have a Button-b
```

```
                                                    // which will show the content of b when clicked
    (new Application()).Run(w)      // create new Application() running the Window-w.
```

- (#1) To attach a handler to a control for an event: `b.AddHandler(Button.ClickEvent,d)`
- (#2) Create a delegate/handler using a function: `let d = new RoutedEventHandler(f)`
- (#3) Create a function with specific signature defined by the delegate: `let f(sender:obj)(e:RoutedEventArgs) = ....`
- b is the control.
- AddHandler is attach.
- Button.ClickEvent is the event.
- d is delegate/handler. It is a layer to make sure the signature is correct
- f is the real function/program provided to the delegate.
- Rule#1: b must have this event Button.ClickEvent: b is type-Button-object. ClickEvent is a static property of type-ButtonBase which is inherited by type-Button. So Button-type will also have this static property ClickEvent.
- Rule#2: d must be the handler of ClickEvent: ClickEvent is type-RoutedEvent. RoutedEvent's handler is RoutedEventHandler, just adding Handler at end. RoutedEventHandler is a defined delegate in .NET library. To create d, just let d = new RoutedEventHandler(f), where f is function.
- Rule#3: f must have signature obeying delegate-d's definition: Check .NET library, RoutedEventHandler is a delegate of C#-signature: `void RoutedEventHandler(object sender, RoutedEventArgs e)`. So f must have same signature. Present the signature in F# is `(obj * RountedEventHandler) -> unit`

Passing State To Callbacks

Events can pass state to callbacks with minimal effort. Here is a simple program which reads a file in blocks of characters:

```
open System

type SuperFileReader() =
    let progressChanged = new Event<int>()

    member this.ProgressChanged = progressChanged.Publish

    member this.OpenFile (filename : string, charsPerBlock) =
        use sr = new System.IO.StreamReader(filename)
        let streamLength = int64 sr.BaseStream.Length
        let sb = new System.Text.StringBuilder(int streamLength)
        let charBuffer = Array.zeroCreate<char> charsPerBlock

        let mutable oldProgress = 0
        let mutable totalCharsRead = 0
        progressChanged.Trigger(0)
        while not sr.EndOfStream do
            (* sr.ReadBlock returns number of characters read from stream *)
            let charsRead = sr.ReadBlock(charBuffer, 0, charBuffer.Length)
            totalCharsRead <- totalCharsRead + charsRead

            (* appending chars read from buffer *)
            sb.Append(charBuffer, 0, charsRead) |> ignore

            let newProgress = int(decimal totalCharsRead / decimal streamLength * 100m)
            if newProgress > oldProgress then
                progressChanged.Trigger(newProgress) // passes newProgress as state to callbacks
                oldProgress <- newProgress

        sb.ToString()

let fileReader = new SuperFileReader()
fileReader.ProgressChanged.Add(fun percent ->
    printfn "%i percent done..." percent)

let x = fileReader.OpenFile(@"C:\Test.txt", 50)
printfn "%s[...]" x.[0 .. if x.Length <= 100 then x.Length - 1 else 100]
```

This program has the following types:

```
type SuperFileReader =
  class
    new : unit -> SuperFileReader
    member OpenFile : filename:string * charsToRead:int -> string
    member ProgressChanged : IEvent<int>
  end
val fileReader : SuperFileReader
val x : string
```

Since our event has the type `IEvent<int>`, we can pass `int` data as state to listening callbacks. This program outputs the following:

```
0 percent done...
4 percent done...
9 percent done...
14 percent done...
19 percent done...
24 percent done...
29 percent done...
34 percent done...
39 percent done...
44 percent done...
49 percent done...
53 percent done...
58 percent done...
63 percent done...
68 percent done...
73 percent done...
78 percent done...
83 percent done...
88 percent done...
93 percent done...
98 percent done...
100 percent done...
In computer programming, event-driven programming or event-based programming is a programming paradig[...]
```

Retrieving State from Callers

A common idiom in event-driven programming is pre- and post-event handling, as well as the ability to cancel events. Cancellation requires two-way communication between an event handler and a listener, which we can easily accomplish through the use of ref cells or mutable members:

```
type Person(name : string) =
    let mutable _name = name;
    let nameChanging = new Event<string * bool ref>()
    let nameChanged = new Event<unit>()

    member this.NameChanging = nameChanging.Publish
    member this.NameChanged = nameChanged.Publish

    member this.Name
        with get() = _name
        and set(value) =
            let cancelChange = ref false
            nameChanging.Trigger(value, cancelChange)

            if not !cancelChange then
                _name <- value
                nameChanged.Trigger()

let p = new Person("Bob")

p.NameChanging.Add(fun (name, cancel) ->
    let exboyfriends = ["Steve"; "Mike"; "Jon"; "Seth"]
    if List.exists (fun forbiddenName -> forbiddenName = name) exboyfriends then
        printfn "-- No %s's allowed!" name
        cancel := true
    else
        printfn "-- Name allowed")

p.NameChanged.Add(fun () ->
    printfn "-- Name changed to %s" p.Name)

let tryChangeName newName =
    printfn "Attempting to change name to '%s'" newName
    p.Name <- newName

tryChangeName "Joe"
tryChangeName "Moe"
tryChangeName "Jon"
tryChangeName "Thor"
```

This program has the following types:

```
type Person =
  class
    new : name:string -> Person
    member Name : string
    member NameChanged : IEvent<unit>
    member NameChanging : IEvent<string * bool ref>
    member Name : string with set
  end
val p : Person
val tryChangeName : string -> unit
```

This program outputs the following:

```
Attempting to change name to 'Joe'
-- Name allowed
-- Name changed to Joe
Attempting to change name to 'Moe'
-- Name allowed
-- Name changed to Moe
Attempting to change name to 'Jon'
-- No Jon's allowed!
Attempting to change name to 'Thor'
-- Name allowed
-- Name changed to Thor
```

If we need to pass a significant amount of state to listeners, then its recommended to wrap the state in an object as follows:

```
type NameChangingEventArgs(newName : string) =
    inherit System.EventArgs()

    let mutable cancel = false
    member this.NewName = newName
    member this.Cancel
        with get() = cancel
        and set(value) = cancel <- value

type Person(name : string) =
    let mutable _name = name;
    let nameChanging = new Event<NameChangingEventArgs>()
    let nameChanged = new Event<unit>()

    member this.NameChanging = nameChanging.Publish
    member this.NameChanged = nameChanged.Publish

    member this.Name
        with get() = _name
        and set(value) =
            let eventArgs = new NameChangingEventArgs(value)
            nameChanging.Trigger(eventArgs)

            if not eventArgs.Cancel then
                _name <- value
                nameChanged.Trigger()

let p = new Person("Bob")
```

```
p.NameChanging.Add(fun e ->
    let exboyfriends = ["Steve"; "Mike"; "Jon"; "Seth"]
    if List.exists (fun forbiddenName -> forbiddenName = e.NewName) exboyfriends then
        printfn "-- No %s's allowed!" e.NewName
        e.Cancel <- true
    else
        printfn "-- Name allowed")

(* ... rest of program ... *)
```

By convention, custom event parameters should inherit from `System.EventArgs (http://msdn.microsoft.com/en-us/library/system.eventargs.aspx)`, and should have the suffix `EventArgs`.

Using the Event Module

F# allows users to pass event handlers around as first-class values in fundamentally the same way as all other functions. The Event module (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/FSharp.Core/Microsoft.FSharp.Control.Event.html) has a variety of functions for working with event handlers:

`val choose : ('T -> 'U option) -> IEvent<'del,'T> -> IEvent<'U> (requires delegate and 'del :> Delegate)`

Return a new event which fires on a selection of messages from the original event. The selection function takes an original message to an optional new message.

`val filter : ('T -> bool) -> IEvent<'del,'T> -> IEvent<'T> (requires delegate and 'del :> Delegate)`

Return a new event that listens to the original event and triggers the resulting event only when the argument to the event passes the given function.

`val listen : ('T -> unit) -> IEvent<'del,'T> -> unit (requires delegate and 'del :> Delegate)`

Run the given function each time the given event is triggered.

`val map : ('T -> 'U) -> IEvent<'del,'T> -> IEvent<'U> (requires delegate and 'del :> Delegate)`

Return a new event which fires on a selection of messages from the original event. The selection function takes an original message to an optional new message.

`val merge : IEvent<'del1,'T> -> IEvent<'del2,'T> -> IEvent<'T> (requires delegate and 'del1 :> Delegate and delegate and 'del2 :> Delegate)`

Fire the output event when either of the input events fire.

`val pairwise : IEvent<'del,'T> -> IEvent<'T * 'T> (requires delegate and 'del :> Delegate)`

Return a new event that triggers on the second and subsequent triggerings of the input event. The Nth triggering of the input event passes the arguments from the N-1th and Nth triggering as a pair. The argument passed to the N-1th triggering is held in hidden internal state until the Nth triggering occurs. You should ensure that the contents of the values being sent down the event are not mutable. Note that many EventArgs types are mutable, e.g. MouseEventArgs, and each firing of an event using this argument type may reuse the same physical argument obejct with different values. In this case you should extract the necessary information from the argument before using this combinator.

`val partition : ('T -> bool) -> IEvent<'del,'T> -> IEvent<'T> * IEvent<'T> (requires delegate and 'del :> Delegate`

Return a new event that listens to the original event and triggers the first resulting event if the application of the predicate to the event arguments returned true, and the second event if it returned false.

`val scan : ('U -> 'T -> 'U) -> 'U -> IEvent<'del,'T> -> IEvent<'U> (requires delegate and 'del :> Delegate)`

Return a new event consisting of the results of applying the given accumulating function to successive values triggered on the input event. An item of internal state records the current value of the state parameter. The internal state is not locked during the execution of the accumulation function, so care should be taken that the input IEvent not triggered by multiple threads simultaneously.

`val split : ('T -> Choice<'U1,'U2>) -> IEvent<'del,'T> -> IEvent<'U1> * IEvent<'U2> (requires delegate and 'del :> Delegate)`

Return a new event that listens to the original event and triggers the first resulting event if the application of the function to the event arguments returned a Choice2Of1, and the second event if it returns a Choice2Of2.

Take the following snippet:

```
p.NameChanging.Add(fun (e : NameChangingEventArgs) ->
    let exboyfriends = ["Steve"; "Mike"; "Jon"; "Seth"]
    if List.exists (fun forbiddenName -> forbiddenName = e.NewName) exboyfriends then
        printfn "-- No %s's allowed!" e.NewName
        e.Cancel <- true)
```

We can rewrite this in a more functional style as follows:

```
p.NameChanging
    |> Event.filter (fun (e : NameChangingEventArgs) ->
        let exboyfriends = ["Steve"; "Mike"; "Jon"; "Seth"]
        List.exists (fun forbiddenName -> forbiddenName = e.NewName) exboyfriends )
    |> Event.listen (fun e ->
```

```
        printfn "-- No %s's allowed!" e.NewName
        e.Cancel <- true)
```

Modules and Namespaces

F# : Modules and Namespaces

Modules and Namespaces are primarily used for grouping and organizing code.

Defining Modules

No code is required to define a module. If a codefile does not contain a leading `namespace` or `module` declaration, F# code will implicitly place the code in a module, where the name of the module is the same as the file name with the first letter capitalized.

To access code in another module, simply use `.` notation: `moduleName.member`. Notice that this notation is similar to the syntax used to access static members—this is not a coincidence. F# modules are compiled as classes which only contain static members, values, and type definitions.

Let's create two files:

DataStructures.fs

```
type 'a Stack =
    | EmptyStack
    | StackNode of 'a * 'a Stack

let rec getRange startNum endNum =
    if startNum > endNum then EmptyStack
    else StackNode(startNum, getRange (startNum+1) endNum)
```

Program.fs

```
let x =
    DataStructures.StackNode(1,
        DataStructures.StackNode(2,
            DataStructures.StackNode(3, DataStructures.EmptyStack)))

let y = DataStructures.getRange 5 10

printfn "%A" x
printfn "%A" y
```

This program outputs:

```
StackNode (1,StackNode (2,StackNode (3,EmptyStack)))
StackNode
  (5,
   StackNode
     (6,StackNode (7,StackNode (8,StackNode (9,StackNode (10,EmptyStack))))))
```

> **Note:** Remember, order of compilation matters in F#. Dependencies must come before dependents, so `DataStructures.fs` comes before `Program.fs` when compiling this program.

Like all modules, we can use the `open` keyword to give us access to the methods inside a module without fully qualifying the naming of the method. This allows us to revise `Program.fs` as follows:

```
open DataStructures

let x = StackNode(1, StackNode(2, StackNode(3, EmptyStack)))
let y = getRange 5 10

printfn "%A" x
printfn "%A" y
```

Submodules

Its very easy to create submodules using the `module` keyword:

```
(* DataStructures.fs *)

type 'a Stack =
    | EmptyStack
    | StackNode of 'a * 'a Stack

module StackOps =
    let rec getRange startNum endNum =
        if startNum > endNum then EmptyStack
        else StackNode(startNum, getRange (startNum+1) endNum)
```

Since the `getRange` method is under another module, the fully qualified name of this method is `DataStructures.StackOps.getRange`. We can use it as follows:

```
(* Program.fs *)

open DataStructures

let x =
    StackNode(1, StackNode(2, StackNode(3, EmptyStack)))

let y = StackOps.getRange 5 10

printfn "%A" x
printfn "%A" y
```

F# allows us to create a module and a type having the same name, for example the following code is perfectly acceptable:

```
type 'a Stack =
    | EmptyStack
    | StackNode of 'a * 'a Stack

module Stack =
    let rec getRange startNum endNum =
        if startNum > endNum then EmptyStack
        else StackNode(startNum, getRange (startNum+1) endNum)
```

> **Note:** Its possible to nest submodules inside other submodules. However, as a general principle, its best to avoid creating complex module hierarchies. Functional programming libraries tend to be very "flat" with nearly all functionality accessible in the first 2 or 3 levels of a hierarchy. This is in contrast to many other OO languages which encourage programmers to create deeply nested class libraries, where functionality might be buried 8 or 10 levels down the hierarchy.

Extending Types and Modules

F# supports extension methods, which allow programmers to add new static and instance methods to classes and modules without inheriting from them.

Extending a Module

The `Seq` module contains several pairs of methods:

- `iter`/`iteri`
- `map`/`mapi`

`Seq` has a `forall` member, but does not have a corresponding `foralli` function, which includes the index of each sequence element. We add this missing method to the module simply by creating another module with the same name. For example, using fsi:

```
> module Seq =
    let foralli f s =
        s
        |> Seq.mapi (fun i x -> i, x)        (* pair item with its index *)
        |> Seq.forall (fun (i, x) -> f i x) (* apply item and index to function *)

let isPalindrome (input : string) =
    input
    |> Seq.take (input.Length / 2)
    |> Seq.foralli (fun i x -> x = input.[input.Length - i - 1]);;

module Seq = begin
  val foralli : (int -> 'a -> bool) -> seq<'a> -> bool
end
val isPalindrome : string -> bool

> isPalindrome "hello";;
val it : bool = false
> isPalindrome "racecar";;
val it : bool = true
```

Extending a Type

The `System.String (http://msdn.microsoft.com/en-us/library/system.string.aspx)` has many useful methods, but let's say we thought it was missing a few important functions, `Reverse` and `IsPalindrome`. Since this class is marked as `sealed` or `NotInheritable`, we can't create a derived version of this class. Instead, we create a module with the new methods we want. Here's an example in fsi which demonstrates how to add new static and instance methods to the `String` class:

```
> module Seq =
    let foralli f s =
        s
        |> Seq.mapi (fun i x -> i, x)        (* pair item with its index *)
        |> Seq.forall (fun (i, x) -> f i x) (* apply item and index to function *)

module StringExtensions =
    type System.String with
        member this.IsPalindrome =
            this
            |> Seq.take (this.Length / 2)
            |> Seq.foralli (fun i x -> this.[this.Length - i - 1] = x)

        static member Reverse(s : string) =
            let chars : char array =
                let temp = Array.zeroCreate s.Length
                let charsToTake = if temp.Length % 2 <> 0 then (temp.Length + 1) / 2 else temp.Length / 2

                s
                |> Seq.take charsToTake
```

```
                |> Seq.iteri (fun i x ->
                    temp.[i] <- s.[temp.Length - i - 1]
                    temp.[temp.Length - i - 1] <- x)
                temp
            new System.String(chars)

open StringExtensions;;

module Seq = begin
  val foralli : (int -> 'a -> bool) -> seq<'a> -> bool
end
module StringExtensions = begin
end

> "hello world".IsPalindrome;;
val it : bool = false
> System.String.Reverse("hello world");;
val it : System.String = "dlrow olleh"
```

Module Signatures

By default, all members in a module are accessible from outside the module. However, a module often contain members which should not be accessible outside itself, such as helper functions. One way to expose only a subset of a module's members is by creating a signature file for that module. (Another way is to apply .Net CLR access modifiers of public, internal, or private to individual declarations).

Signature files have the same name as their corresponding module, but end with a ".fsi" extension (f-sharp interface). Signature files always come before their implementation files, which have a corresponding ".fs" extension. For example:

DataStructures.fsi

```
type 'a stack =
    | EmptyStack
    | StackNode of 'a * 'a stack

module Stack =
    val getRange : int -> int -> int stack
    val hd : 'a stack -> 'a
    val tl : 'a stack -> 'a stack
    val fold : ('a -> 'b -> 'a) -> 'a -> 'b stack -> 'a
    val reduce : ('a -> 'a -> 'a) -> 'a stack -> 'a
```

DataStructures.fs

```
type 'a stack =
    | EmptyStack
    | StackNode of 'a * 'a stack

module Stack =
    (* helper functions *)
    let internal_head_tail = function
        | EmptyStack -> failwith "Empty stack"
        | StackNode(hd, tail) -> hd, tail

    let rec internal_fold_left f acc = function
        | EmptyStack -> acc
        | StackNode(hd, tail) -> internal_fold_left f (f acc hd) tail

    (* public functions *)
    let rec getRange startNum endNum =
        if startNum > endNum then EmptyStack
        else StackNode(startNum, getRange (startNum+1) endNum)

    let hd s = internal_head_tail s |> fst
    let tl s = internal_head_tail s |> snd
    let fold f seed stack = internal_fold_left f seed stack
    let reduce f stack = internal_fold_left f (hd stack) (tl stack)
```

Program.fs

```
open DataStructures

let x = Stack.getRange 1 10
printfn "%A" (Stack.hd x)
printfn "%A" (Stack.tl x)
printfn "%A" (Stack.fold ( * ) 1 x)
printfn "%A" (Stack.reduce ( + ) x)
(* printfn "%A" (Stack.internal_head_tail x) *) (* will not compile *)
```

Since `Stack.internal_head_tail` is not defined in our interface file, the method is marked `private` and no longer accessible outside of the `DataStructures` module.

Module signatures are useful to building a code library's skeleton, however they have a few caveats. If you want to expose a class, record, or union in a module through a signature, then the signature file must expose *all* of the objects members, records fields, and union's cases. Additionally, the signature of the function defined in the module and it's corresponding signature in the signature file must match *exactly*. Unlike OCaml, F# does not allow a function in a module with the generic type `'a -> 'a -> 'a` to be restricted to `int -> int -> int` in the signature file.

Defining Namespaces

A namespace is a hierarchical categorization of modules, classes, and other namespaces. For example, the `System.Collections (http://msdn.microsoft.com/en-us/library/system.collections.aspx)` namespace groups together all of the collections and data structures in the .NET BCL, whereas the `System.Security.Cryptography (http://msdn.microsoft.com/en-us/library/system.security.cryptography.aspx)` namespace groups together all classes which provide cryptographic services.

Namespaces are primarily used to avoid name conflicts. For example, let's say we were writing an application incorporated code from several different vendors. If Vendor A and Vendor B both have a class called `Collections.Stack`, and we wrote the code `let s = new Stack()`, how would the compiler know whether which stack we intended to create? Namespaces can eliminate this ambiguity by adding one more layer of grouping to our code.

Code is grouped under a namespace using the `namespace` keyword:

DataStructures.fsi

```
namespace Princess.Collections
    type 'a stack =
        | EmptyStack
        | StackNode of 'a * 'a stack

    module Stack =
        val getRange : int -> int -> int stack
        val hd : 'a stack -> 'a
        val tl : 'a stack -> 'a stack
        val fold : ('a -> 'b -> 'a) -> 'a -> 'b stack -> 'a
        val reduce : ('a -> 'a -> 'a) -> 'a stack -> 'a
```

DataStructures.fs

```
namespace Princess.Collections

    type 'a stack =
        | EmptyStack
        | StackNode of 'a * 'a stack

    module Stack =
        (* helper functions *)
        let internal_head_tail = function
            | EmptyStack -> failwith "Empty stack"
            | StackNode(hd, tail) -> hd, tail

        let rec internal_fold_left f acc = function
            | EmptyStack -> acc
            | StackNode(hd, tail) -> internal_fold_left f (f acc hd) tail

        (* public functions *)
        let rec getRange startNum endNum =
            if startNum > endNum then EmptyStack
            else StackNode(startNum, getRange (startNum+1) endNum)

        let hd s = internal_head_tail s |> fst
        let tl s = internal_head_tail s |> snd
        let fold f seed stack = internal_fold_left f seed stack
        let reduce f stack = internal_fold_left f (hd stack) (tl stack)
```

Program.fs

```
open Princess.Collections

let x = Stack.getRange 1 10
printfn "%A" (Stack.hd x)
printfn "%A" (Stack.tl x)
printfn "%A" (Stack.fold ( * ) 1 x)
printfn "%A" (Stack.reduce ( + ) x)
```

Where is the `DataStructures` Module?

You may have expected the code in `Program.fs` above to open `Princess.Collections.DataStructures` rather than `Princess.Collections`. According to the F# spec (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785747), F# treats *anonymous implementation files* (which are files without a leading `module` or `namespace` declaration) by putting all code in an implicit module which matches the code's filename. Since we have a leading `namespace` declaration, F# does not create the implicit module.

.NET does not permit users to create functions or values outside of classes or modules. As a consequence, we cannot write the following code:

```
namespace Princess.Collections

    type 'a stack =
        | EmptyStack
        | StackNode of 'a * 'a stack

    let somefunction() = 12   (* <--- functions not allowed outside modules *)

    (* ... *)
```

If we prefer to have a module called `DataStructures`, we can write this:

```
namespace Princess.Collections
    module DataStructures
        type 'a stack =
            | EmptyStack
            | StackNode of 'a * 'a stack

        let somefunction() = 12

        (* ... *)
```

Or equivalently, we define a module and place it a namespace simultaneously using:

```
module Princess.Collections.DataStructures
    type 'a stack =
```

```
        | EmptyStack
        | StackNode of 'a * 'a stack

    let somefunction() = 12

    (* ... *)
```

Adding to Namespace from Multiple Files

Unlike modules and classes, any file can contribute to a namespace. For example:

DataStructures.fs

```
namespace Princess.Collections

    type 'a stack =
        | EmptyStack
        | StackNode of 'a * 'a stack

    module Stack =
        (* helper functions *)
        let internal_head_tail = function
            | EmptyStack -> failwith "Empty stack"
            | StackNode(hd, tail) -> hd, tail

        let rec internal_fold_left f acc = function
            | EmptyStack -> acc
            | StackNode(hd, tail) -> internal_fold_left f (f acc hd) tail

        (* public functions *)
        let rec getRange startNum endNum =
            if startNum > endNum then EmptyStack
            else StackNode(startNum, getRange (startNum+1) endNum)

        let hd s = internal_head_tail s |> fst
        let tl s = internal_head_tail s |> snd
        let fold f seed stack = internal_fold_left f seed stack
        let reduce f stack = internal_fold_left f (hd stack) (tl stack)
```

MoreDataStructures.fs

```
namespace Princess.Collections

    type 'a tree when 'a :> System.IComparable<'a> =
        | EmptyTree
        | TreeNode of 'a * 'a tree * 'a tree

    module Tree =
        let rec insert (x : #System.IComparable<'a>) = function
            | EmptyTree -> TreeNode(x, EmptyTree, EmptyTree)
            | TreeNode(y, l, r) as node ->
                match x.CompareTo(y) with
                | 0 -> node
                | 1 -> TreeNode(y, l, insert x r)
                | -1 -> TreeNode(y, insert x l, r)
                | _ -> failwith "CompareTo returned illegal value"
```

Since we have a leading namespace declaration in both files, F# does not create any implicit modules. The `'a stack`, `'a tree`, `Stack`, and `Tree` types are all accessible through the `Princess.Collections` namespace:

Program.fs

```
open Princess.Collections

let x = Stack.getRange 1 10
let y =
    let rnd = new System.Random()
    [ for a in 1 .. 10 -> rnd.Next(0, 100) ]
    |> Seq.fold (fun acc x -> Tree.insert x acc) EmptyTree

printfn "%A" (Stack.hd x)
printfn "%A" (Stack.tl x)
printfn "%A" (Stack.fold ( * ) 1 x)
printfn "%A" (Stack.reduce ( + ) x)
printfn "%A" y
```

Controlling Class and Module Accessibility

Unlike modules, there is no equivalent to a signature file for namespaces. Instead, the visibility of classes and submodules is controlled through standard accessibility modifiers (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785709):

```
namespace Princess.Collections

    type 'a tree when 'a :> System.IComparable<'a> =
        | EmptyTree
        | TreeNode of 'a * 'a tree * 'a tree

    (* InvisibleModule is only accessible by classes or
       modules inside the Princess.Collections namespace*)
    module private InvisibleModule =
        let msg = "I'm invisible!"

    module Tree =
        (* InvisibleClass is only accessible by methods
           inside the Tree module *)
        type private InvisibleClass() =
            member x.Msg() = "I'm invisible too!"
```

```
        let rec insert (x : #System.IComparable<'a>) = function
            | EmptyTree -> TreeNode(x, EmptyTree, EmptyTree)
            | TreeNode(y, l, r) as node ->
                match x.CompareTo(y) with
                | 0 -> node
                | 1 -> TreeNode(y, l, insert x r)
                | -1 -> TreeNode(y, insert x l, r)
                | _ -> failwith "CompareTo returned illegal value"
```

Units of Measure

F# : Units of Measure

Units of measure allow programmers to annotate floats and integers with statically-typed unit metadata. This can be handy when writing programs which manipulate floats and integers representing specific units of measure, such as kilograms, pounds, meters, newtons, pascals, etc. F# will verify that units are used in places where the programmer intended. For example, the F# compiler will throw an error if a `float<m/s>` is used where it expects a `float<kg>`.

Use Cases

Statically Checked Type Conversions

Units of measure are invaluable to programmers who work in scientific research, they add an extra layer of protection to guard against conversion related errors. To cite a famous case study, NASA's $125 million Mars Climate Orbiter (http://mars.jpl.nasa.gov/msp98/news/mco991110.html) project ended in failure when the orbiter dipped 90 km closer to Mars than originally intended, causing it to tear apart and disintegrate spectacularly in the Mars atmosphere. A post mortem analysis narrowed down the root cause of the problem to a conversion error in the orbiter's propulsion systems used to lower the spacecraft into orbit: NASA passed data to the systems in metric units, but the software expected data in Imperial units. Although there were many contributing project-management errors which resulted in the failed mission, this software bug in particular could have been prevented if the software engineers had used a type-system powerful enough to detect unit-related errors.

Decorating Data With Contextual Information

In an article Making Code Look Wrong (http://www.joelonsoftware.com/articles/Wrong.html), Joel Spolsky describes a scenario in which, during the design of Microsoft Word and Excel, programmers at Microsoft were required to track the position of objects on a page using two non-interchangeable coordinate systems:

> In WYSIWYG word processing, you have scrollable windows, so every coordinate has to be interpreted as either relative to the window or relative to the page, and that makes a big difference, and keeping them straight is pretty important. [...]
> The compiler won't help you if you assign one to the other and Intellisense won't tell you bupkis. But they are semantically different; they need to be interpreted differently and treated differently and some kind of conversion function will need to be called if you assign one to the other or you will have a *runtime* bug. If you're lucky. [...]
> In Excel's source code you see a lot of `rw` and `col` and when you see those you know that they refer to rows and columns. Yep, they're both integers, but it never makes sense to assign between them. In Word, I'm told, you see a lot of `xl` and `xw`, where `xl` means "horizontal coordinates relative to the layout" and `xw` means "horizontal coordinates relative to the window." Both ints. Not interchangeable. In both apps you see a lot of `cb` meaning "count of bytes." Yep, it's an int again, but you know so much more about it just by looking at the variable name. It's a count of bytes: a buffer size. And if you see `xl = cb`, well, blow the Bad Code Whistle, that is obviously wrong code, because even though `xl` and `cb` are both integers, it's completely crazy to set a horizontal offset in pixels to a count of bytes.

In short, Microsoft depends on coding conventions to encode contextual data about a variable, and they depend on code reviews to enforce correct usage of a variable from its context. This works in practice, but its still possible for incorrect code to work its way it the product without the bug being detected for months.

If Microsoft were using a language with units of measure, they could have defined their own `rw`, `col`, `xw`, `xl`, and `cb` units of measure so that an assignment of the form `int<xl> = int<cb>` not only fails visual inspection, it doesn't even compile.

Defining Units

New units of measure are defined using the `Measure` attribute:

```
[<Measure>]
type m (* meter *)

[<Measure>]
type s (* second *)
```

Additionally, we can define types measures which are derived from existing measures as well:

```
[<Measure>] type m                  (* meter *)
[<Measure>] type s                  (* second *)
[<Measure>] type kg                 (* kilogram *)
[<Measure>] type N = (kg * m)/(s^2) (* Newtons *)
[<Measure>] type Pa = N/(m^2)       (* Pascals *)
```

> **Important:** Units of measure look like a data type, but they aren't. .NET's type system does not support the behaviors that units of measure have, such as being able to square, divide, or raise datatypes to powers. This functionality is provided by the F# static type checker at compile time, but units are erased from compiled code. Consequently, it is not possible to determine value's unit at runtime.

We can create instances of float and integer data which represent these units using the same notation we use with generics:

```
> let distance = 100.0<m>
let time = 5.0<s>
let speed = distance / time;;

val distance : float<m> = 100.0
val time : float<s> = 5.0
val speed : float<m/s> = 20.0
```

Notice the that F# automatically derives a new unit, `m/s`, for the value `speed`. Units of measure will multiply, divide, and cancel as needed depending on how they are used. Using these properties, it's very easy to convert between two units:

```
[<Measure>] type C
[<Measure>] type F

let to_fahrenheit (x : float<C>) = x * (9.0<F>/5.0<C>) + 32.0<F>
let to_celsius (x : float<F>) = (x - 32.0<F>) * (5.0<C>/9.0<F>)
```

Units of measure are statically checked at compile time for proper usage. For example, if we use a measure where it isn't expected, we get a compilation error:

```
> [<Measure>] type m
[<Measure>] type s

let speed (x : float<m>) (y : float<s>) = x / y;;

[<Measure>]
type m
[<Measure>]
type s
val speed : float<m> -> float<s> -> float<m/s>

> speed 20.0<m> 4.0<s>;; (* should get a speed *)
val it : float<m/s> = 5.0

> speed 20.0<m> 4.0<m>;; (* boom! *)

  speed 20.0<m> 4.0<m>;;
  --------------^^^^^^

stdin(39,15): error FS0001: Type mismatch. Expecting a
    float<s>
but given a
    float<m>.
The unit of measure 's' does not match the unit of measure 'm'
```

Units can be defined for integral types too:

```
> [<Measure>] type col
[<Measure>] type row
let colOffset (a : int<col>) (b : int<col>) = a - b
let rowOffset (a : int<row>) (b : int<row>) = a - b;;

[<Measure>]
type col
[<Measure>]
type row
val colOffset : int<col> -> int<col> -> int<col>
val rowOffset : int<row> -> int<row> -> int<row>
```

Dimensionless Values

A value without a unit is *dimensionless*. Dimensionless values are represented implicitly by writing them out without units (i.e. `7.0`, `-14`, `200.5`), or they can be represented explicitly using the `<1>` type (i.e. `7.0<1>`, `-14<1>`, `200.5<1>`).

We can convert dimensionless units to a specific measure by multiplying by `1<`*`targetMeasure`*`>`. We can convert a measure back to a dimensionless unit by passing it to the built-in `float` or `int` methods:

```
[<Measure>] type m

(* val to_meters : (x : float<'u>) -> float<'u m> *)
let to_meters x = x * 1<m>

(* val of_meters : (x : float<m>) -> float *)
let of_meters (x : float<m>) = float x
```

Alternatively, its often easier (and safer) to divide away unneeded units:

```
let of_meters (x : float<m>) = x / 1.0<m>
```

Generalizing Units of Measure

Since measures and dimensionless values are (or appear to be) generic types, we can write functions which operate on both transparently:

```
> [<Measure>] type m
[<Measure>] type kg

let vanillaFloats = [10.0; 15.5; 17.0]
let lengths = [ for a in [2.0; 7.0; 14.0; 5.0] -> a * 1.0<m> ]
let masses = [ for a in [155.54; 179.01; 135.90] -> a * 1.0<kg> ]
```

```
let densities = [ for a in [0.54; 1.0; 1.1; 0.25; 0.7] -> a * 1.0<kg/m^3> ]

let average (l : float<'u> list) =
    let sum, count = l |> List.fold (fun (sum, count) x -> sum + x, count + 1.0<_>) (0.0<_>, 0.0<_>)
    sum / count;;

[<Measure>]
type m
[<Measure>]
type kg
val vanillaFloats : float list = [10.0; 15.5; 17.0]
val lengths : float<m> list = [2.0; 7.0; 14.0; 5.0]
val masses : float<kg> list = [155.54; 179.01; 135.9]
val densities : float<kg/m ^ 3> list = [0.54; 1.0; 1.1; 0.25; 0.7]
val average : float<'u> list -> float<'u>

> average vanillaFloats, average lengths, average masses, average densities;;
val it : float * float<m> * float<kg> * float<kg/m ^ 3> =
  (14.16666667, 7.0, 156.8166667, 0.718)
```

Since units are erased from compiled code, they are not considered a real data type, so they can't be used *directly* as a type parameter in generic functions and classes. For example, the following code will not compile:

```
> type triple<'a> = { a : float<'a>; b : float<'a>; c : float<'a>};;

  type triple<'a> = { a : float<'a>; b : float<'a>; c : float<'a>};;
  ----------------------------^^

stdin(40,31): error FS0191: Expected unit-of-measure parameter, not type parameter.
Explicit unit-of-measure parameters must be marked with the [<Measure>] attribute
```

F# does not infer that `'a` is a unit of measure above, possibly because the following code *appears* correct, but it can be used in non-sensical ways:

```
type quad<'a> = { a : float<'a>; b : float<'a>; c : float<'a>; d : 'a}
```

The type `'a` can be a unit of measure *or* a data type, but not both at the same time. F#'s type checker assumes `'a` is a type parameter *unless otherwise specified*. We can use the `[<Measure>]` attribute to change the `'a` to a unit of measure:

```
> type triple<[<Measure>] 'a> = { a : float<'a>; b : float<'a>; c : float<'a>};;

type triple<[<Measure>] 'a> =
  {a: float<'a>;
   b: float<'a>;
   c: float<'a>;}

> { a = 7.0<kg>; b = -10.5<_>; c = 0.5<_> };;
val it : triple<kg> = {a = 7.0;
                       b = -10.5;
                       c = 0.5;}
```

F# PowerPack

The F# PowerPack (FSharp.PowerPack.dll) includes a number of predefined units of measure for scientific applications. These are available in the following modules:

- Microsoft.FSharp.Math.SI (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/fsharp.powerpack/Microsoft.FSharp.Math.SI.html) - a variety of predefined measures in the International System of Units (SI).
- Microsoft.FSharp.Math.PhysicalConstants (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/fsharp.powerpack/Microsoft.FSharp.Math.PhysicalConstants.html) - Fundamental physical constants with units of measure.

External Resources

- Andrew Kennedy's 4-part tutorial on units of measure:
  - Part 1: Introducing Units (http://blogs.msdn.com/andrewkennedy/archive/2008/08/29/units-of-measure-in-f-part-one-introducing-units.aspx)
  - Part 2: Unit Conversions (http://blogs.msdn.com/andrewkennedy/archive/2008/09/02/units-of-measure-in-f-part-two-unit-conversions.aspx)
  - Part 3: Generic Units (http://blogs.msdn.com/andrewkennedy/archive/2008/09/04/units-of-measure-in-f-part-three-generic-units.aspx)
  - Part 4: Parameterized Types (http://blogs.msdn.com/andrewkennedy/archive/2009/06/09/units-of-measure-in-f_2300_-part-four-parameterized-types.aspx)
- F# Units of Measure (http://msdn.microsoft.com/en-us/library/dd233243(VS.100).aspx) (MSDN)

Caching

F# : Caching

Caching is often useful to re-use data which has already been computed. F# provides a number of built-in techniques to cache data for future use.

Partial Functions

F# automatically caches the value of any function which takes no parameters. When F# comes across a function with no parameters, F# will only evaluate the function once and reuse its value everytime the function is accessed. Compare the following:

```
let isNebraskaCity_bad city =
    let cities =
        printfn "Creating cities Set"
        ["Bellevue"; "Omaha"; "Lincoln"; "Papillion"]
        |> Set.ofList

    cities.Contains(city)

let isNebraskaCity_good =
    let cities =
        printfn "Creating cities Set"
        ["Bellevue"; "Omaha"; "Lincoln"; "Papillion"]
        |> Set.ofList

    fun city -> cities.Contains(city)
```

Both functions accept and return the same values, but they have very different behavior. Here's a comparison of the output in fsi:

isNebraskaCity_bad

```
> let isNebraskaCity_bad city =
    let cities =
        printfn "Creating cities Set"
        ["Bellevue"; "Omaha"; "Lincoln"; "Papillion"]
        |> Set.of_list

    cities.Contains(city);;

val isNebraskaCity_bad : string -> bool

> isNebraskaCity_bad "Lincoln";;
Creating cities Set
val it : bool = true

> isNebraskaCity_bad "Washington";;
Creating cities Set
val it : bool = false

> isNebraskaCity_bad "Bellevue";;
Creating cities Set
val it : bool = true

> isNebraskaCity_bad "St. Paul";;
Creating cities Set
val it : bool = false
```

isNebraskaCity_good

```
> let isNebraskaCity_good =
    let cities =
        printfn "Creating cities Set"
        ["Bellevue"; "Omaha"; "Lincoln"; "Papillion"]
        |> Set.of_list

    fun city -> cities.Contains(city);;
Creating cities Set

val isNebraskaCity_good : (string -> bool)

> isNebraskaCity_good "Lincoln";;
val it : bool = true

> isNebraskaCity_good "Washington";;
val it : bool = false

> isNebraskaCity_good "Bellevue";;
val it : bool = true

> isNebraskaCity_good "St. Paul";;
val it : bool = false
```

The implementation of `isNebraskaCity_bad` forces F# to re-create the internal set on each call. On the other hand, `isNebraskaCity_good` is a value initialized to the function `fun city -> cities.Contains(city)`, so it creates its internal set once and reuses it for all successive calls.

> **Note:** Internally, `isNebraskaCity_bad` is compiled as a static function which constructs a set on every call. `isNebraskaCity_good` is compiled as a static readonly property, where the value is initialized in a static constructor.

This distinction is often subtle, but it can have a huge impact on an application's performance.

Memoization

"Memoization" is a fancy word meaning that computed values are stored in a lookup table rather than recomputed on every successive call. Long-running pure functions (i.e. functions which have no side-effects) are good candidates for memoization. Consider the recursive definition for computing Fibonacci numbers:

```
> #time;;

--> Timing now on

> let rec fib n =
    if n = 0I then 0I
    elif n = 1I then 1I
    else fib (n - 1I) + fib(n - 2I);;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0

val fib : Math.bigint -> Math.bigint

> fib 35I;;
Real: 00:00:23.557, CPU: 00:00:23.515, GC gen0: 2877, gen1: 3, gen2: 0
val it : Math.bigint = 9227465I
```

Beyond `fib 35I`, the runtime of the function becomes unbearable. Each recursive call to the `fib` function throws away all of its intermediate calculations to `fib(n - 1I)` and `fib(n - 2I)`, giving it a runtime complexity of about O(2^n). What if we kept all of those intermediate calculations around in a lookup table? Here's the memoized version of the `fib` function:

```
> #time;;

--> Timing now on

> let rec fib =
    let dict = new System.Collections.Generic.Dictionary<_,_>()
    fun n ->
        match dict.TryGetValue(n) with
        | true, v -> v
        | false, _ ->
            let temp =
                if n = 0I then 0I
                elif n = 1I then 1I
```

```
                else fib (n - 1I) + fib(n - 2I)
            dict.Add(n, temp)
            temp;;

val fib : (Math.bigint -> Math.bigint)

> fib 35I;;
Real: 00:00:00.000, CPU: 00:00:00.000, GC gen0: 0, gen1: 0, gen2: 0
val it : Math.bigint = 9227465I
```

Much better! This version of the `fib` function runs almost instaneously. In fact, since we only calculate the value of any `fib(n)` precisely once, and dictionary lookups are an `O(1)` operation, this fib function runs in O(n) time.

Notice all of the memoization logic is contained in the `fib` function. We can write a more general function to memoize any function:

```
let memoize f =
    let dict = new System.Collections.Generic.Dictionary<_,_>()
    fun n ->
        match dict.TryGetValue(n) with
        | (true, v) -> v
        | _ ->
            let temp = f(n)
            dict.Add(n, temp)
            temp

let rec fib = memoize(fun n ->
    if n = 0I then 0I
    elif n = 1I then 1I
    else fib (n - 1I) + fib (n - 2I) )
```

> **Note:** Its very important to remember that the implementation above is not thread-safe -- the dictionary should be locked before adding/retrieving items if it will be accessed by multiple threads.
>
> Additionally, although dictionary lookups occur in constant time, the hash function used by the dictionary can take an arbitrarily long time to execute (this is especially true with strings, where the time it takes to hash a string is proportional to its length). For this reason, it is wholly possible for a memoized function to have less performance than an unmemoized function. Always profile code to determine whether optimization is necessary and whether memoization genuinely improves performance.

Lazy Values

The F# `lazy` data type is an interesting primitive which delays evaluation of a value until the value is actually needed. Once computed, lazy values are cached for reuse later:

```
> let x = lazy(printfn "I'm lazy"; 5 + 5);;

val x : Lazy<int> = <unevaluated>

> x.Force();; (* Should print "I'm lazy" *)
I'm lazy
val it : int = 10

> x.Force();; (* Value already computed, should not print "I'm lazy" again *)
val it : int = 10
```

F# uses some compiler magic to avoid evaluating the expression `(printfn "I'm lazy"; 5 + 5)` on declaration. Lazy values are probably the simplest form of caching, however they can be used to create some interesting and sophisticated data structures. For example, two F# data structures are implemented on top of lazy values, namely the F# Lazy List (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/fsharp.powerpack/microsoft.fsharp.collections.lazylist.html) and `Seq.cache` method.

Lazy lists and cached sequences represent arbitrary sequences of potentially infinite numbers of elements. The elements are computed and cached the first time they are accessed, but will not be recomputed when the sequence is enumerated again. Here's a demonstration in fsi:

```
> let x = seq { for a in 1 .. 10 -> printfn "Got %i" a; a } |> Seq.cache;;

val x : seq<int>

> let y = Seq.take 5 x;;

val y : seq<int>

> Seq.reduce (+) y;;
Got 1
Got 2
Got 3
Got 4
Got 5
val it : int = 15

> Seq.reduce (+) y;; (* Should not recompute values *)
val it : int = 15

> Seq.reduce (+) x;; (* Values 1 through 5 already computed, should only compute 6 through 10 *)
Got 6
Got 7
Got 8
Got 9
Got 10
val it : int = 55
```

Active Patterns

F# : Active Patterns

Active Patterns allow programmers to wrap arbitrary values in a union-like data structure for easy pattern matching. For example, its possible wrap objects with an active pattern, so that you can use objects in pattern matching as easily as any other union type.

Defining Active Patterns

Active patterns look like functions with a funny name:

```
let (|name1|name2|...|) = ...
```

This function defines an ad hoc union data structure, where each union case `name`*`n`* is separated from the next by a `|` and the entire list is enclosed between `(|` and `|)` (humbly called "banana brackets"). In other words, the function does not have a simple name at all, it instead defines a series of union constructors.

A typical active pattern might look like this:

```
let (|Even|Odd|) n =
    if n % 2 = 0 then
        Even
    else
        Odd
```

`Even` and `Odd` are union constructors, so our active pattern either returns an instance of `Even` or an instance of `Odd`. The code above is roughly equivalent to the following:

```
type numKind =
    | Even
    | Odd

let get_choice n =
    if n % 2 = 0 then
        Even
    else
        Odd
```

Active patterns can also define union constructors which take a set of parameters. For example, consider we can wrap a `seq<'a>` with an active pattern as follows:

```
let (|SeqNode|SeqEmpty|) s =
    if Seq.isEmpty s then SeqEmpty
    else SeqNode ((Seq.head s), Seq.skip 1 s)
```

This code is, of course, equivalent to the following:

```
type seqWrapper<'a> =
    | SeqEmpty
    | SeqNode of 'a * seq<'a>

let get_choice s =
    if Seq.isEmpty s then SeqEmpty
    else SeqNode ((Seq.head s), Seq.skip 1 s)
```

You've probably noticed the immediate difference between active patterns and explicitly defined unions:

- Active patterns define an *anonymous* union, where the explicit union has a name (`numKind`, `seqWrapper`, etc.).
- Active patterns determine their constructor parameters using a kind of type-inference, whereas we need to explicitly define the constructor parameters for each case of our explicit union.

Using Active Patterns

The syntax for using active patterns looks a little odd, but once you know what's going on, it's very easy to understand. Active patterns are used in pattern matching expressions, for example:

```
> let (|Even|Odd|) n =
    if n % 2 = 0 then Even
    else Odd

let testNum n =
    match n with
    | Even -> printfn "%i is even" n
    | Odd -> printfn "%i is odd" n;;

val ( |Even|Odd| ) : int -> Choice<unit,unit>
val testNum : int -> unit

> testNum 12;;
12 is even
val it : unit = ()

> testNum 17;;
17 is odd
```

What's going on here? When the pattern matching function encounters `Even`, it calls `(|Even|Odd|)` with parameter in the match clause, it's as if you've written:

```
type numKind =
    | Even
    | Odd

let get_choice n =
    if n % 2 = 0 then
        Even
    else
        Odd

let testNum n =
    match get_choice n with
    | Even -> printfn "%i is even" n
    | Odd -> printfn "%i is odd" n
```

The parameter in the match clause is always passed as the last argument to the active pattern expression. Using our `seq` example from earlier, we can write the following:

```
> let (|SeqNode|SeqEmpty|) s =
    if Seq.isEmpty s then SeqEmpty
    else SeqNode ((Seq.head s), Seq.skip 1 s)

let perfectSquares = seq { for a in 1 .. 10 -> a * a }

let rec printSeq = function
    | SeqEmpty -> printfn "Done."
    | SeqNode(hd, tl) ->
        printf "%A " hd
        printSeq tl;;

val ( |SeqNode|SeqEmpty| ) : seq<'a> -> Choice<('a * seq<'a>),unit>
val perfectSquares : seq<int>
val printSeq : seq<'a> -> unit

> printSeq perfectSquares;;
1 4 9 16 25 36 49 64 81 100 Done.
```

Traditionally, `seq`'s are resistant to pattern matching, but now we can operate on them just as easily as lists.

Parameterizing Active Patterns

Its possible to pass arguments to active patterns, for example:

```
> let (|Contains|) needle (haystack : string) =
    haystack.Contains(needle)

let testString = function
    | Contains "kitty" true -> printfn "Text contains 'kitty'"
    | Contains "doggy" true -> printfn "Text contains 'doggy'"
    | _ -> printfn "Text neither contains 'kitty' nor 'doggy'";;

val ( |Contains| ) : string -> string -> bool
val testString : string -> unit

> testString "I have a pet kitty and she's super adorable!";;
Text contains 'kitty'
val it : unit = ()

> testString "She's fat and purrs a lot :)";;
Text neither contains 'kitty' nor 'doggy'
```

The single-case active pattern `(|Contains|)` wraps the `String.Contains` function. When we call `Contains "kitty" true`, F# passes `"kitty"` and the argument we're matching against to the `(|Contains|)` active pattern and tests the return value against the value `true`. The code above is equivalent to:

```
type choice =
    | Contains of bool

let get_choice needle (haystack : string) = Contains(haystack.Contains(needle))

let testString n =
    match get_choice "kitty" n with
    | Contains(true) -> printfn "Text contains 'kitty'"
    | _ ->
        match get_choice "doggy" n with
        | Contains(true) -> printfn "Text contains 'doggy'"
        | printfn "Text neither contains 'kitty' nor 'doggy'"
```

As you can see, the code using the active patterns is much cleaner and easier to read than the equivalent code using the explicitly defined union.

> **Note:** Single-case active patterns might not look terribly useful at first, but they can really help to clean up messy code. For example, the active pattern above wraps up the `String.Contains` method and allows us to invoke it in a pattern matching expression. Without the active pattern, pattern matching quickly becomes messy:
>
> ```
> let testString = function
>     | (n : string) when n.Contains("kitty") -> printfn "Text contains 'kitty'"
>     | n when n.Contains("doggy") -> printfn "Text contains 'doggy'"
>     | _ -> printfn "Text neither contains 'kitty' nor 'doggy'"
> ```

Partial Active Patterns

A partial active pattern is a special class of single-case active patterns: it either returns `Some` or `None`. For example, a very handy active pattern for working with regex can be defined as follows:

```
> let (|RegexContains|_|) pattern input =
    let matches = System.Text.RegularExpressions.Regex.Matches(input, pattern)
    if matches.Count > 0 then Some [ for m in matches -> m.Value ]
    else None

let testString = function
    | RegexContains "http://\S+" urls -> printfn "Got urls: %A" urls
    | RegexContains "[^@]@[^.]+\.\W+" emails -> printfn "Got email address: %A" emails
    | RegexContains "\d+" numbers -> printfn "Got numbers: %A" numbers
    | _ -> printfn "Didn't find anything.";;

val ( |RegexContains|_| ) : string -> string -> string list option
val testString : string -> unit

> testString "867-5309, Jenny are you there?";;
Got numbers: ["867"; "5309"]
```

This is equivalent to writing:

```
type choice =
    | RegexContains of string list

let get_choice pattern input =
    let matches = System.Text.RegularExpressions.Regex.Matches(input, pattern)
    if matches.Count > 0 then Some (RegexContains [ for m in matches -> m.Value ])
    else None

let testString n =
    match get_choice "http://\S+" n with
    | Some(RegexContains(urls)) -> printfn "Got urls: %A" urls
    | None ->
        match get_choice "[^@]@[^.]+\.\W+" n with
        | Some(RegexContains emails) -> printfn "Got email address: %A" emails
        | None ->
            match get_choice "\d+" n with
            | Some(RegexContains numbers) -> printfn "Got numbers: %A" numbers
            | _ -> printfn "Didn't find anything."
```

Using partial active patterns, we can test an input against any number of active patterns:

```
let (|StartsWith|_|) needle (haystack : string) = if haystack.StartsWith(needle) then Some() else None
let (|EndsWith|_|) needle (haystack : string) = if haystack.EndsWith(needle) then Some() else None
let (|Equals|_|) x y = if x = y then Some() else None
let (|EqualsMonkey|_|) = function (* "Higher-order" active pattern *)
    | Equals "monkey" () -> Some()
    | _ -> None
let (|RegexContains|_|) pattern input =
    let matches = System.Text.RegularExpressions.Regex.Matches(input, pattern)
    if matches.Count > 0 then Some [ for m in matches -> m.Value ]
    else None

let testString n =
    match n with
    | StartsWith "kitty" () -> printfn "starts with 'kitty'"
    | StartsWith "bunny" () -> printfn "starts with 'bunny'"
    | EndsWith "doggy" () -> printfn "ends with 'doggy'"
    | Equals "monkey" () -> printfn "equals 'monkey'"
    | EqualsMonkey -> printfn "EqualsMonkey!" (* Note: EqualsMonkey and EqualsMonkey() are equivalent *)
    | RegexContains "http://\S+" urls -> printfn "Got urls: %A" urls
    | RegexContains "[^@]@[^.]+\.\W+" emails -> printfn "Got email address: %A" emails
    | RegexContains "\d+" numbers -> printfn "Got numbers: %A" numbers
    | _ -> printfn "Didn't find anything."
```

Partial active patterns don't constrain us to a finite set of cases like traditional unions do, we can use as many partial active patterns in match statement as we need.

Additional Resources

- Introduction to Active Patterns (Internet Archive Wayback Machine) (https://web.archive.org/web/20190127200824/https://blogs.msdn.microsoft.com/chrsmith/2008/02/21/introduction-to-f-active-patterns/) by Chris Smith
- Extensible Pattern Matching via Lightweight Language Extension (http://research.microsoft.com/pubs/79947/p29-syme.pdf) by Don Syme

Advanced Data Structures

F# : Advanced Data Structures

F# comes with its own set of data structures, however its very important to know how to implement data structures from scratch.

Incidentally, hundreds of authors have written thousands of lengthy volumes on this single topic alone, so its unreasonable to provide a comprehensive picture of data structures in the short amount of space available for this book. Instead, this chapter is intended as a cursory introduction to the development of immutable data structures using F#. Readers are encouraged to use the resources listed at the bottom of this page for a more comprehensive treatment of algorithms and data structures.

Stacks

F#'s built-in list data structure is essentially an immutable stack. While its certainly usable, for the purposes of writing exploratory code, we're going to implement a stack from scratch. We can represent each node in a stack using a simple union:

```
type 'a stack =
    | EmptyStack
    | StackNode of 'a * 'a stack
```

It's easy enough to create an instance of a stack using:

```
let stack = StackNode(1, StackNode(2, StackNode(3, StackNode(4, StackNode(5, EmptyStack)))))
```

Each StackNode contains a value and a pointer to the next stack in the list. The resulting data structure can be diagrammed as follows:

```
 ___  ___  ___  ___  ___
|_1_|->|_2_|->|_3_|->|_4_|->|_5_|->Empty
```

We can create a boilerplate stack module as follows:

```
module Stack =
    type 'a stack =
        | EmptyStack
        | StackNode of 'a * 'a stack

    let hd = function
        | EmptyStack -> failwith "Empty stack"
        | StackNode(hd, tl) -> hd

    let tl = function
        | EmptyStack -> failwith "Empty stack"
        | StackNode(hd, tl) -> tl

    let cons hd tl = StackNode(hd, tl)

    let empty = EmptyStack
```

Let's say we wanted to add a few methods to our stack, such as a method which updates an item at a certain index. Since our nodes are immutable, we can't update our list in place; we need to copy all of the nodes up to the node we want to update.

```
Setting item at index 2 to the value 9.

         0      1      2      3      4

         ___    ___    ___    ___    ___
let x =  |_1_|->|_2_|->|_3_|->|_4_|->|_5_|->Empty
                          ^
                         /
          ___    ___    ___ /
let y =  |_1_|->|_2_|->|_9_|
```

So, we copy all of the nodes up to index 2 and reuse the remaining nodes. A function like this is very easy to write:

```
let rec update index value s =
    match index, s with
    | index, EmptyStack -> failwith "Index out of range"
    | 0, StackNode(hd, tl) -> StackNode(value, tl)
    | n, StackNode(hd, tl) -> StackNode(hd, update (index - 1) value tl)
```

Appending items from one stack to the rear of another uses a similar technique. Since we can't modify stacks in place, we append two stacks by copying all of the nodes from the "front" stack and pointing the last copied node to the our "rear" stack, resulting in the following:

```
Append x and y

          ___    ___    ___    ___    ___
let x =  |_1_|->|_2_|->|_3_|->|_4_|->|_5_|->Empty

                                    ___    ___    ___
let y =                            |_6_|->|_7_|->|_8_|->Empty
                                   ^
                                  /
           ___    ___    ___    ___ /
let z =  |_1_|->|_2_|->|_3_|->|_4_|->|_5_|
```

We can implement this function with minimal effort using the following:

```
let rec append x y =
    match x with
    | EmptyStack -> y
    | StackNode(hd, tl) -> StackNode(hd, append tl y)
```

Stacks are very easy to work with and implement. The principles behind copying nodes to "modify" stacks is fundamentally the same for all persistent data structures.

Complete Stack Module

```
module Stack =
    type 'a stack =
        | EmptyStack
        | StackNode of 'a * 'a stack

    let hd = function
        | EmptyStack -> failwith "Empty stack"
```

```
        | StackNode(hd, tl) -> hd

    let tl = function
        | EmptyStack -> failwith "Emtpy stack"
        | StackNode(hd, tl) -> tl

    let cons hd tl = StackNode(hd, tl)

    let empty = EmptyStack

    let rec update index value s =
        match index, s with
        | index, EmptyStack -> failwith "Index out of range"
        | 0, StackNode(hd, tl) -> StackNode(value, tl)
        | n, StackNode(hd, tl) -> StackNode(hd, update (index - 1) value tl)

    let rec append x y =
        match x with
        | EmptyStack -> y
        | StackNode(hd, tl) -> StackNode(hd, append tl y)

    let rec map f = function
        | EmptyStack -> EmptyStack
        | StackNode(hd, tl) -> StackNode(f hd, map f tl)

    let rec rev s =
        let rec loop acc = function
            | EmptyStack -> acc
            | StackNode(hd, tl) -> loop (StackNode(hd, acc)) tl
        loop EmptyStack s

    let rec contains x = function
        | EmptyStack -> false
        | StackNode(hd, tl) -> hd = x || contains x tl

    let rec fold f seed = function
        | EmptyStack -> seed
        | StackNode(hd, tl) -> fold f (f seed hd) tl
```

Queues

Naive Queue

Queues aren't quite as straightforward as stacks. A naive queue can be implemented using a stack, with the caveat that:

- Items are always appended to the end of the list, and dequeued from the head of the stack.
- -OR- Items are prepended to the front of the stack, and dequeued by reversing the stack and getting its head.

```
(* AwesomeCollections.fsi *)

type 'a stack =
  | EmptyStack
  | StackNode of 'a * 'a stack

module Stack = begin
  val hd : 'a stack -> 'a
  val tl : 'a stack -> 'a stack
  val cons : 'a -> 'a stack -> 'a stack
  val empty : 'a stack
  val rev : 'a stack -> 'a stack
end

[<Class>]
type 'a Queue =
    member hd : 'a
    member tl : 'a Queue
    member enqueue : 'a -> 'a Queue
    static member empty : 'a Queue
```

```
(* AwesomeCollections.fs *)

type 'a stack =
    | EmptyStack
    | StackNode of 'a * 'a stack

module Stack =
    let hd = function
        | EmptyStack -> failwith "Empty stack"
        | StackNode(hd, tl) -> hd

    let tl = function
        | EmptyStack -> failwith "Emtpy stack"
        | StackNode(hd, tl) -> tl

    let cons hd tl = StackNode(hd, tl)

    let empty = EmptyStack

    let rec rev s =
        let rec loop acc = function
            | EmptyStack -> acc
            | StackNode(hd, tl) -> loop (StackNode(hd, acc)) tl
        loop EmptyStack s

type Queue<'a>(item : stack<'a>) =
    member this.hd
        with get() = Stack.hd (Stack.rev item)

    member this.tl
        with get() = Queue(item |> Stack.rev |> Stack.tl |> Stack.rev)

    member this.enqueue(x) = Queue(StackNode(x, item))

    override this.ToString() = sprintf "%A" item
```

```
    static member empty = Queue<'a>(Stack.empty)
```

We use an interface file to hide the `Queue` class's constructor. Although this technically satisfies the function of a queue, every dequeue is an O(*n*) operation where *n* is the number of items in the queue. There are lots of variations on the same approach, but these are often not very practical in practice. We can certainly improve on the implementation of immutable queues.

Queue From Two Stacks

The implementation above isn't very efficient because it requires reversing our underlying data representation several times. Why not keep those reversed stacks around for future use? Rather than using one stack, we can have two stacks: a front stack `f` and a rear stack `r`.

Stack `f` holds items in the correct order, while stack `r` holds items in reverse order; this allows the first element in `f` to be the head of the queue, and the first element in `r` to be the last item in queue. So, a queue of the numbers 1 .. 6 might be represented with `f = [1;2;3]` and `r = [6;5;4]`.

To enqueue a new item, prepend it to the front of `r`; to dequeue an item, pop it off `f`. Both enqueues and dequeues are O(1) operations. Of course, at some point, `f` will be empty and there will be no more items to dequeue; in this case, simply move all items from `r` to `f` and reverse the list. While the queue certainly has O(n) worst-case behavior, it has acceptable O(1) amortized (average case) bounds.

The code for this implementation is straight forward:

```
type Queue<'a>(f : stack<'a>, r : stack<'a>) =
    let check = function
        | EmptyStack, r -> Queue(Stack.rev r, EmptyStack)
        | f, r -> Queue(f, r)

    member this.hd =
        match f with
        | EmptyStack -> failwith "empty"
        | StackNode(hd, tl) -> hd

    member this.tl =
        match f, r with
        | EmptyStack, _ -> failwith "empty"
        | StackNode(x, f), r -> check(f, r)

    member this.enqueue(x) = check(f, StackNode(x, r))

    static member empty = Queue<'a>(Stack.empty, Stack.empty)
```

This is a simple, common, and useful implementation of an immutable queue. The magic is in the `check` function which maintains that `f` always contains items if they are available.

> **Note:** The queue's periodic O(n) worst case behavior can give it unpredictable response times, especially in applications which rely heavily on persistence since its possible to hit the pathological case each time the queue is accessed. However, this particular implementation of queues is perfectly adequate for the vast majority of applications which do not require persistence or uniform response times.

As shown above, we often want to wrap our underlying data structure in class for two reasons:

1. To simplify the interface to the data structure. For example, clients neither know nor care that our queue uses two stacks; they only know that items in the queue obey the principle of first-in, first-out.
2. To prevent clients from putting the underlying data in the data structure in an invalid state.

Beyond stacks, virtually all data structures are complex enough to require wrapping up class to hide away complex details from clients.

Binary Search Trees

Binary search trees are similar to stacks, but each node points to two other nodes called the left and right child nodes:

```
type 'a tree =
    | EmptyTree
    | TreeNode of 'a * 'a tree * 'a tree
```

Additionally, nodes in the tree are ordered in a particular way: each item in a tree is greater than all items in its left child node and less than all items in its right child node.

Since our tree is immutable, we "insert" into the tree by returning a brand new tree with the node inserted. This process is more efficient than it sounds: we copy nodes as we traverse down the tree, so we only copy nodes which are in the path of our node being inserted. Writing a binary search tree is relatively straightforward:

```
(* AwesomeCollections.fsi *)

[<Class>]
type 'a BinaryTree =
    member hd : 'a
    member exists : 'a -> bool
    member insert : 'a -> 'a BinaryTree
```

```
(* AwesomeCollections.fs *)

type 'a tree =
    | EmptyTree
    | TreeNode of 'a * 'a tree * 'a tree

module Tree =
    let hd = function
        | EmptyTree -> failwith "empty"
```

```
        | TreeNode(hd, l, r) -> hd

    let rec exists item = function
        | EmptyTree -> false
        | TreeNode(hd, l, r) ->
            if hd = item then true
            elif item < hd then exists item l
            else exists item r

    let rec insert item = function
        | EmptyTree -> TreeNode(item, EmptyTree, EmptyTree)
        | TreeNode(hd, l, r) as node ->
            if hd = item then node
            elif item < hd then TreeNode(hd, insert item l, r)
            else TreeNode(hd, l, insert item r)

type 'a BinaryTree(inner : 'a tree) =
    member this.hd = Tree.hd inner

    member this.exists item = Tree.exists item inner

    member this.insert item = BinaryTree(Tree.insert item inner)

    member this.empty = BinaryTree<'a>(EmptyTree)
```

We're using an interface and a wrapper class to hide the implementation details of the tree from the user, otherwise the user could construct a tree which invalidates the specific ordering rules used in the binary tree.

This implementation is simple and it allows us to add and lookup any item in the tree in O(log n) best case time. However, it suffers from a pathological case: if we add items in sorted order, or mostly sorted order, then the tree can become heavily unbalanced. For example, the following code:

```
[1 .. 7] |> Seq.fold (fun (t : BinaryTree<_>) x -> t.insert(x)) BinaryTree.empty
```

Results in this tree:

```
    1
   / \
  E   2
     / \
    E   3
       / \
      E   4
         / \
        E   5
           / \
          E   6
             / \
            E   7
               / \
              E   E
```

A tree like this isn't much better than our inefficient queue implementation above! Trees are most efficient when they have a minimum height and are as full as possible. Ideally, we'd like to represent the tree above as follows:

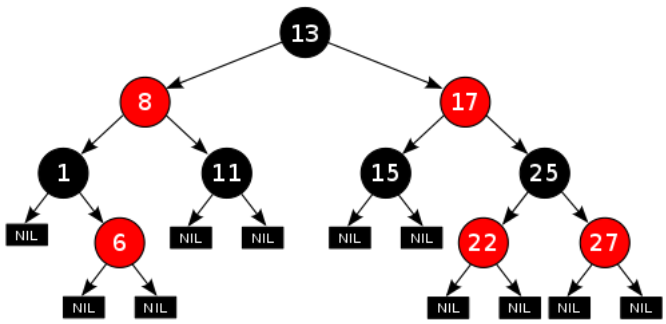
```
        _ 4 _
       /     \
      2       6
     / \     / \
    1   3   5   7
   / \ / \ / \ / \
   E E E E E E E E
```

The minimum height of the tree is ceiling(log n + 1), where n is the number of items in the list. When we insert items into the tree, we want the tree to balance itself to maintain the minimum height. There are a variety of self-balancing tree implementations, many of which are easy to implement as immutable data structures.

Red Black Trees

Red-black trees are self-balancing trees which attach a "color" attribute to each node in the tree. In addition to the rules defining a binary search tree, red-black trees must maintain the following set of rules:

1. A node is either red or black.
2. The root node is always black.
3. No red node has a red child.
4. Every simple path from a given node to any of its descendant leaves contains the same number of black nodes.

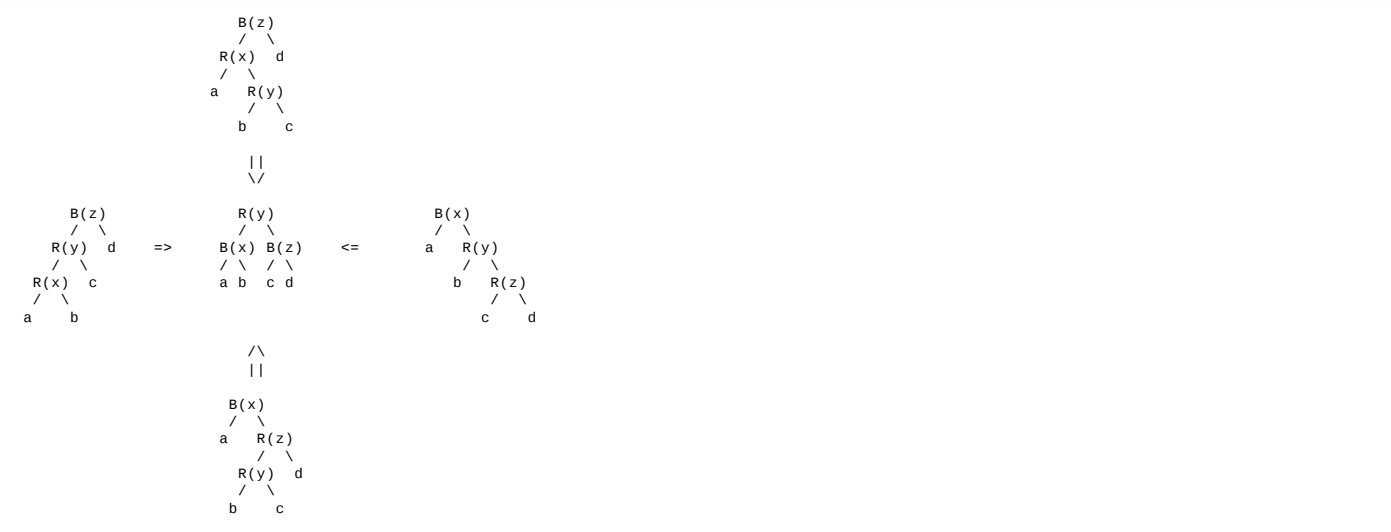


An example of a red-black tree

We can augment our binary tree with a color field as follows:

```
type color = R | B
type 'a tree =
    | E
    | T of color * 'a * 'a tree * 'a tree
```

When we insert into the tree, we need to rebalance the tree to restore the rules. In particular, we need to remove nodes with a red child. There are four cases where a red node may have a red child. They are depicted in the diagram below by the top, right, bottom, and left trees. The center tree is the balanced version.



We can modify our binary tree class as follows:

```
(* AwesomeCollections.fsi *)

[<Class>]
type 'a BinaryTree =
    member hd : 'a
    member left : 'a BinaryTree
    member right : 'a BinaryTree
    member exists : 'a -> bool
    member insert : 'a -> 'a BinaryTree
    member print : unit -> unit
    static member empty : 'a BinaryTree
```

```
(* AwesomeCollections.fs *)

type color = R | B
type 'a tree =
    | E
    | T of color * 'a tree * 'a * 'a tree

module Tree =
    let hd = function
        | E -> failwith "empty"
        | T(c, l, x, r) -> x

    let left = function
        | E -> failwith "empty"
        | T(c, l, x, r) -> l

    let right = function
        | E -> failwith "empty"
        | T(c, l, x, r) -> r

    let rec exists item = function
        | E -> false
        | T(c, l, x, r) ->
            if item = x then true
            elif item < x then exists item l
            else exists item r

    let balance = function                                  (* Red nodes in relation to black root *)
        | B, T(R, T(R, a, x, b), y, c), z, d                (* Left, left *)
        | B, T(R, a, x, T(R, b, y, c)), z, d                (* Left, right *)
        | B, a, x, T(R, T(R, b, y, c), z, d)                (* Right, left *)
        | B, a, x, T(R, b, y, T(R, c, z, d))                (* Right, right *)
            -> T(R, T(B, a, x, b), y, T(B, c, z, d))
        | c, l, x, r -> T(c, l, x, r)

    let insert item tree =
        let rec ins = function
            | E -> T(R, E, item, E)
            | T(c, a, y, b) as node ->
                if item = y then node
                elif item < y then balance(c, ins a, y, b)
                else balance(c, a, y, ins b)

        (* Forcing root node to be black *)
        match ins tree with
            | E -> failwith "Should never return empty from an insert"
            | T(_, l, x, r) -> T(B, l, x, r)

    let rec print (spaces : int) = function
        | E -> ()
        | T(c, l, x, r) ->
            print (spaces + 4) r
```

```
            printfn "%s %A%A" (new System.String(' ', spaces)) c x
            print (spaces + 4) l

type 'a BinaryTree(inner : 'a tree) =
    member this.hd = Tree.hd inner
    member this.left = BinaryTree(Tree.left inner)
    member this.right = BinaryTree(Tree.right inner)
    member this.exists item = Tree.exists item inner
    member this.insert item = BinaryTree(Tree.insert item inner)
    member this.print() = Tree.print 0 inner
    static member empty = BinaryTree<'a>(E)
```

All of the magic that makes this tree work happens in the `balance` function. We're not performing any terribly complicated transformations to the tree, yet it comes out relatively balanced (in fact, the maximum depth of this tree is 2 * ceiling(log n + 1)).

AVL Trees

AVL trees are named after its two inventors, G.M. Adelson-Velskii and E.M. Landis. These trees are self-balancing because the heights of the two child subtrees of any node will only differ 0 or 1; therefore, these trees are said to be height-balanced.

An empty node in a tree has a height of 0; non-empty nodes have a height >= 1. We can store the height of each node in our tree definition:

```
type 'a tree =
    | Node of int * 'a tree * 'a * 'a tree  (* height, left child, value, right child *)
    | Nil
```

The height of any node is equal to `max(left height, right height) + 1`. For convenience, we'll use the following constructor to create a tree node and initialize its height:

```
let height = function
    | Node(h, _, _, _) -> h
    | Nil -> 0

let make l x r =
    let h = 1 + max (height l) (height r)
    Node(h, l, x ,r)
```

Inserting into an AVL tree is very similar to inserting into an unbalanced binary tree with one exception: after we insert a node, we use a series of tree rotations to re-balance the tree. Each node has an implicit property, its *balance factor*, which refers to the left-child's height minus the right-child's height; a positive balance factor indicates the tree is weighted on the left, negative indicates the tree is weighted on the right, otherwise the tree is balanced.

We only need to rebalance the tree when balance factor for a node is +/-2. There are four scenarios which can cause our tree to become unbalanced:

Left-left case: root balance factor = +2, left-childs balance factor = +1. Balanced by right-rotating the root node:

```
         5                           3
       /   \    Root                /   \
      3     D   Right rotation     2     5
     / \            ----->        / \   / \
    2   C                        A   B C   D
   / \
  A   B
```

Left-right case: root balance factor = +2, right-child's balance factor = -1. Balanced by left-rotating the left child, then right-rotating the root (this operation is called a double right rotation):

```
       5                             5                               4
     /   \      Left child          /   \        Root               /   \
    3     D     Left rotation      4     D       Right rotation     3     5
   / \              ----->        / \              ----->         / \   / \
  A   4                           3   C                          A   B C   D
     / \                         / \
    B   C                       A   B
```

Right-right case: root balance factor = -2, right-child's balance factor = -1. Balanced by left-rotating the root node:

```
    3                             5
   /   \          Root            /   \
  A     5         Left rotation   3     7
       / \            ----->     / \   / \
      B   7                     A   B C   D
         / \
        C   D
```

Right-left case: root balance factor = -2, right-child's balance factor = +1. Balanced by right-rotating the right child, then left-rotating the root (this operation is called a double-left rotation):

```
     3                               3                               4
   /   \        Right child         /   \        Root               /   \
  A     5       Right rotation     A     4       Left rotation      3     5
       / \           ----->              / \          ----->      / \   / \
      4   D                             B   5                     A   B C   D
     / \                                   / \
    B   C                                 C   D
```

With this in mind, its very easy to put together the rest of our AVL tree:

```fsharp
(* AwesomeCollections.fsi *)
[<Class>]
type 'a AvlTree =
    member Height : int
    member Left : 'a AvlTree
    member Right : 'a AvlTree
    member Value : 'a
    member Insert : 'a -> 'a AvlTree
    member Contains : 'a -> bool

module AvlTree =
    [<GeneralizableValue>]
    val empty<'a> : AvlTree<'a>
```

```fsharp
(* AwesomeCollections.fs *)

type 'a tree =
    | Node of int * 'a tree * 'a * 'a tree
    | Nil

(*
    Notation:
        h = height
        x = value
        l = left child
        r = right child

        lh = left child's height
        lx = left child's value
        ll = left child's left child
        lr = left child's right child

        rh = right child's height
        rx = right child's value
        rl = right child's left child
        rr = right child's right child
*)

let height = function
    | Node(h, _, _, _) -> h
    | Nil -> 0

let make l x r =
    let h = 1 + max (height l) (height r)
    Node(h, l, x ,r)

let rotRight = function
    | Node(_, Node(_, ll, lx, lr), x, r) ->
        let r' = make lr x r
        make ll lx r'
    | node -> node

let rotLeft = function
    | Node(_, l, x, Node(_, rl, rx, rr)) ->
        let l' = make l x rl
        make l' rx rr
    | node -> node

let doubleRotLeft = function
    | Node(h, l, x, r) ->
        let r' = rotRight r
        let node' = make l x r'
        rotLeft node'
    | node -> node

let doubleRotRight = function
    | Node(h, l, x, r) ->
        let l' = rotLeft l
        let node' = make l' x r
        rotRight node'
    | node -> node

let balanceFactor = function
    | Nil -> 0
    | Node(_, l, _, r) -> (height l) - (height r)

let balance = function
    (* left unbalanced *)
    | Node(h, l, x, r) as node when balanceFactor node >= 2 ->
        if balanceFactor l >= 1 then rotRight node      (* left left case *)
        else doubleRotRight node                        (* left right case *)
    (* right unbalanced *)
    | Node(h, l, x, r) as node when balanceFactor node <= -2 ->
        if balanceFactor r <= -1 then rotLeft node      (* right right case *)
        else doubleRotLeft node                         (* right left case *)
    | node -> node

let rec insert v = function
    | Nil -> Node(1, Nil, v, Nil)
    | Node(_, l, x, r) as node ->
        if v = x then node
        else
            let l', r' = if v < x then insert v l, r else l, insert v r
            let node' = make l' x r'
            balance <| node'

let rec contains v = function
    | Nil -> false
    | Node(_, l, x, r) ->
        if v = x then true
        else
            if v < x then contains v l
            else contains v r

type 'a AvlTree(tree : 'a tree) =
    member this.Height = height tree

    member this.Left =
        match tree with
        | Node(_, l, _, _) -> new AvlTree<'a>(l)
        | Nil -> failwith "Empty tree"
```

```
    member this.Right =
        match tree with
        | Node(_, _, _, r) -> new AvlTree<'a>(r)
        | Nil -> failwith "Empty tree"

    member this.Value =
        match tree with
        | Node(_, _, x, _) -> x
        | Nil -> failwith "Empty tree"

    member this.Insert(x) = new AvlTree<'a>(insert x tree)

    member this.Contains(v) = contains v tree

module AvlTree =
    [<GeneralizableValue>]
    let empty<'a> : AvlTree<'a> = new AvlTree<'a>(Nil)
```

> **Note:** The `[<GeneralizableValue>]` attribute indicates to F# that the construct can give rise to generic code through type inference . Without the attribute, F# will infer the type of `AvlTree.empty` as the undefined type `AvlTree<'_a>`, resulting in a "value restriction" error at compilation.

> **Optimization tip:** The tree supports inserts and lookups in `log(n)` time, where n is the number of nodes in the tree. This is already pretty good, but we can make it faster by eliminating unnecessary comparisons. Notice when we insert a node into the left side of the tree, we can only add weight to the left child; however, the `balance` function checks both sides of the tree for each insert. By re-writing `balance` into a `balance_left` and `balance_right` function to handle, we can handle left- and right-child inserts separately. Similar optimizations are possible on the red-black tree implementation as well.

An AVL trees height is limited to `1.44 * log(n)`, whereas a red-black tree's height is limited to `2 * log(n)`. The AVL trees smaller height and more rigid balancing leads to slower insert/removal but faster retrieval than red-black trees. In practice, the difference will be hardly noticeable: a lookup on a 10,000,000 node AVL tree lookup requires at most 34 comparisons, compared to 47 comparisons on a red-black tree.

Heaps

Binary search trees can efficiently find arbitrary elements in a set, however it can be occasionally useful to access the *minimum* element in set. Heaps are special data structure which satisfy the *heap property*: the value of every node is greater than the value of any of its child nodes. Additionally, we can keep the tree approximately balanced using the *leftist property*, meaning that the height of any left child heap is at least as large as its right sibling. We can hold the height of each tree in each heap node.

Finally, since heaps can be implemented as min- or max-heaps, where the root element will either be the largest or smallest element in the set, we support both types of heaps by passing in an ordering function into heap's constructor as such:

```
type 'a heap =
    | EmptyHeap
    | HeapNode of int * 'a * 'a heap * 'a heap

type 'a BinaryHeap(comparer : 'a -> 'a -> int, inner : 'a heap) =
    static member make(comparer) = BinaryHeap<_>(comparer, EmptyHeap)
```

> **Note:** the functionality we gain by passing the `comparer` function into the `BinaryHeap` constructor approximates OCaml functors, although its not quite as elegant.

An interesting consequence of the leftist property is that elements along any path in a heap are stored in sorted order. This means we can merge any two heaps by merging their right spines and swapping children as necessary to restore the leftist property. Since each right spine contains at least as many nodes as the left spine, the height of each right spine is proportional to the logarithm of the number of elements in the heap, so merging two heaps can be performed in O(log n) time. We can implement all of the properties of our heap as follows:

```
(* AwesomeCollections.fsi *)

[<Class>]
type 'a BinaryHeap =
    member hd : 'a
    member tl : 'a BinaryHeap
    member insert : 'a -> 'a BinaryHeap
    member merge : 'a BinaryHeap -> 'a BinaryHeap
    interface System.Collections.IEnumerable
    interface System.Collections.Generic.IEnumerable<'a>
    static member make : ('b -> 'b -> int) -> 'b BinaryHeap
```

```
(* AwesomeCollections.fs *)

type 'a heap =
    | EmptyHeap
    | HeapNode of int * 'a * 'a heap * 'a heap

module Heap =
    let height = function
        | EmptyHeap -> 0
        | HeapNode(h, _, _, _) -> h

    (* Helper function to restore the leftist property *)
    let makeT (x, a, b) =
        if height a >= height b then HeapNode(height b + 1, x, a, b)
        else HeapNode(height a + 1, x, b, a)

    let rec merge comparer = function
        | x, EmptyHeap -> x
        | EmptyHeap, x -> x
        | (HeapNode(_, x, l1, r1) as h1), (HeapNode(_, y, l2, r2) as h2) ->
            if comparer x y <= 0 then makeT(x, l1, merge comparer (r1, h2))
            else makeT (y, l2, merge comparer (h1, r2))

    let hd = function
        | EmptyHeap -> failwith "empty"
        | HeapNode(h, x, l, r) -> x
```

```
    let tl comparer = function
        | EmptyHeap -> failwith "empty"
        | HeapNode(h, x, l, r) -> merge comparer (l, r)

    let rec to_seq comparer = function
        | EmptyHeap -> Seq.empty
        | HeapNode(h, x, l, r) as node -> seq { yield x; yield! to_seq comparer (tl comparer node) }

type 'a BinaryHeap(comparer : 'a -> 'a -> int, inner : 'a heap) =
    (* private *)
    member this.inner = inner

    (* public *)
    member this.hd = Heap.hd inner
    member this.tl = BinaryHeap(comparer, Heap.tl comparer inner)
    member this.merge (other : BinaryHeap<_>) = BinaryHeap(comparer, Heap.merge comparer (inner, other.inner))
    member this.insert x = BinaryHeap(comparer, Heap.merge comparer (inner,(HeapNode(1, x, EmptyHeap, EmptyHeap))))

    interface System.Collections.Generic.IEnumerable<'a> with
        member this.GetEnumerator() = (Heap.to_seq comparer inner).GetEnumerator()

    interface System.Collections.IEnumerable with
        member this.GetEnumerator() = (Heap.to_seq comparer inner :> System.Collections.IEnumerable).GetEnumerator()

    static member make(comparer) = BinaryHeap<_>(comparer, EmptyHeap)
```

This heap implements the `IEnumerable<'a>` interface, allowing us to iterate through it like a `seq`. In addition to the leftist heap shown above, its very easy to implement immutable versions of splay heaps, binomial heaps, Fibonacci heaps, pairing heaps, and a variety other tree-like data structures in F#.

Lazy Data Structures

Its worth noting that some purely functional data structures above are not as efficient as their imperative implementations. For example, appending two immutable stacks `x` and `y` together takes O(*n*) time, where *n* is the number of elements in stack `x`. However, we can exploit laziness in ways which make purely functional data structures just as efficient as their imperative counterparts.

For example, its easy to create a stack-like data structure which delays all computation until its really needed:

```
type 'a lazyStack =
    | Node of Lazy<'a * 'a lazyStack>
    | EmptyStack

module LazyStack =
    let (|Cons|Nil|) = function
        | Node(item) ->
            let hd, tl = item.Force()
            Cons(hd, tl)
        | EmptyStack -> Nil

    let hd = function
        | Cons(hd, tl) -> hd
        | Nil -> failwith "empty"

    let tl = function
        | Cons(hd, tl) -> tl
        | Nil -> failwith "empty"

    let cons(hd, tl) = Node(lazy(hd, tl))

    let empty = EmptyStack

    let rec append x y =
        match x with
        | Cons(hd, tl) -> Node(lazy(printfn "appending... got %A" hd; hd, append tl y))
        | Nil -> y

    let rec iter f = function
        | Cons(hd, tl) -> f(hd); iter f tl
        | Nil -> ()
```

In the example above, the `append` operation returns one node delays the rest of the computation, so appending two lists will occur in constant time. A `printfn` statement above has been added to demonstrate that we really don't compute appended values until the first time they're accessed:

```
> open LazyStack;;
> let x = cons(1, cons(2, cons(3, cons(4, EmptyStack))));;

val x : int lazyStack = Node <unevaluated>

> let y = cons(5, cons(6, cons(7, EmptyStack)));;

val y : int lazyStack = Node <unevaluated>

> let z = append x y;;

val z : int lazyStack = Node <unevaluated>

> hd z;;
appending... got 1
val it : int = 1

> hd (tl (tl z) );;
appending... got 2
appending... got 3
val it : int = 3

> iter (fun x -> printfn "%i" x) z;;
1
2
3
appending... got 4
4
5
6
```

```
7
val it : unit = ()
```

Interestingly, the `append` method clearly runs in O(1) time because the actual appending operation is delayed until a user grabs the head of the list. At the same time, grabbing the head of the list may have the side effect of triggering, at most, one call to the append method without causing a monolithic rebuilding the rest of the data structure, so grabbing the head is itself an O(1) operation. This stack implementation supports supports constant-time consing and appending, and linear time lookups.

Similarly, implementations of lazy queues exists which support O(1) worst-case behavior for all operations.

Additional Resources

- Purely Functional Data Structures] by Chris Okasaki ISBN 978-0521663502. Highly recommended. Provides techniques and analysis of immutable data structures using SML.
- The Algorithm Design Manual by Steven S. Skiena ISBN 978-0387948607. Highly recommended. Provides language-agnostic description of a variety of algorithms, data structures, and techniques for solving hard problems in computer science.
- Tutorial on immutable data structures using C#:

1. Kinds of Immutability (http://blogs.msdn.com/ericlippert/archive/2007/11/13/immutability-in-c-part-one-kinds-of-immutability.aspx)
2. A Simple Immutable Stack (http://blogs.msdn.com/ericlippert/archive/2007/12/04/immutability-in-c-part-two-a-simple-immutable-stack.aspx)
3. A Covariant Immutable Stack (http://blogs.msdn.com/ericlippert/archive/2007/12/06/immutability-in-c-part-three-a-covariant-immutable-stack.aspx)
4. An Immutable Queue (http://blogs.msdn.com/ericlippert/archive/2007/12/10/immutability-in-c-part-four-an-immutable-queue.aspx)
5. LOLZ! (http://blogs.msdn.com/ericlippert/archive/2007/12/13/immutability-in-c-part-five-lolz.aspx)
6. A Simple Binary Tree (http://blogs.msdn.com/ericlippert/archive/2007/12/18/immutability-in-c-part-six-a-simple-binary-tree.aspx)
7. More on Binary Trees (http://blogs.msdn.com/ericlippert/archive/2007/12/19/immutability-in-c-part-seven-more-on-binary-trees.aspx)
8. Even More on Binary Trees (http://blogs.msdn.com/ericlippert/archive/2008/01/18/immutability-in-c-part-eight-even-more-on-binary-trees.aspx)
9. AVL Tree Implementation (http://blogs.msdn.com/ericlippert/archive/2008/01/21/immutability-in-c-part-nine-academic-plus-my-avl-tree-implementation.aspx)
10. A Double-ended Queue (http://blogs.msdn.com/ericlippert/archive/2008/01/22/immutability-in-c-part-10-a-double-ended-queue.aspx)
11. A Working Double-ended Queue (http://blogs.msdn.com/ericlippert/archive/2008/02/12/immutability-in-c-part-eleven-a-working-double-ended-queue.aspx)

Reflection

F# : Reflection

Reflection allows programmers to inspect types and invoke methods of objects at runtime without knowing their data type at compile time.

At first glance, reflection seems to go against the spirit of ML as it is inherently not type-safe, so typing errors using reflection are not discovered until runtime. However, .NET's typing philosophy is best stated as *static typing where possible*, *dynamic typing when needed*, where reflection serves to bring in the most desirable behaviors of dynamic typing into the static typing world. In fact, dynamic typing can be a huge time saver, often promotes the design of more expressive APIs, and allows code to be refactored much further than possible with static typing.

This section is intended as a cursory overview of reflection, not a comprehensive tutorial.

Inspecting Types

There are a variety of ways to inspect the type of an object. The most direct way is calling the `.GetType()` method (inherited from `System.Object`) on any non-null object:

```
> "hello world".GetType();;
val it : System.Type =
  System.String
    {Assembly = mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089;
     AssemblyQualifiedName = "System.String, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
     Attributes = AutoLayout, AnsiClass, Class, Public, Sealed, Serializable, BeforeFieldInit;
     BaseType = System.Object;
     ContainsGenericParameters = false;
     DeclaringMethod = ?;
     DeclaringType = null;
     FullName = "System.String";
     GUID = 296afbff-1b0b-3ff5-9d6c-4e7e599f8b57;
     GenericParameterAttributes = ?;
     GenericParameterPosition = ?;
     ...
```

Its also possible to get type information without an actual object using the built-in `typeof` method:

```
> typeof<System.IO.File>;;
val it : System.Type =
  System.IO.File
    {Assembly = mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089;
     AssemblyQualifiedName = "System.IO.File, mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
     Attributes = AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit;
```

```
    BaseType = System.Object;
    ContainsGenericParameters = false;
    DeclaringMethod = ?;
    DeclaringType = null;
    FullName = "System.IO.File";
    ...
```

object.GetType and typeof return an instance of System.Type (http://msdn.microsoft.com/en-us/library/system.type_members.aspx), which has a variety of useful properties such as:

- **val Name : string**

  Returns the name of the type.

- **val GetConstructors : unit -> ConstructorInfo array**

  Returns an array of constructors defined on the type.

- **val GetMembers : unit -> MemberInfo array**

  Returns an array of members defined on the type.

- **val InvokeMember : (name : string, invokeAttr : BindingFlags, binder : Binder, target : obj, args : obj) -> obj**

  Invokes the specified member, using the specified binding constraints and matching the specified argument list

Example: Reading Properties

The following program will print out the properties of any object passed into it:

```
type Car(make : string, model : string, year : int) =
    member this.Make = make
    member this.Model = model
    member this.Year = year
    member this.WheelCount = 4

type Cat() =
    let mutable age = 3
    let mutable name = System.String.Empty

    member this.Purr() = printfn "Purrr"
    member this.Age
        with get() = age
        and set(v) = age <- v
    member this.Name
        with get() = name
        and set(v) = name <- v

let printProperties x =
    let t = x.GetType()
    let properties = t.GetProperties()
    printfn "-----------"
    printfn "%s" t.FullName
    properties |> Array.iter (fun prop ->
        if prop.CanRead then
            let value = prop.GetValue(x, null)
            printfn "%s: %O" prop.Name value
        else
            printfn "%s: ?" prop.Name)

let carInstance = new Car("Ford", "Focus", 2009)
let catInstance =
    let temp = new Cat()
    temp.Name <- "Mittens"
    temp

printProperties carInstance
printProperties catInstance
```

This program outputs the following:

```
-----------
Program+Car
WheelCount: 4
Year: 2009
Model: Focus
Make: Ford
-----------
Program+Cat
Name: Mittens
Age: 3
```

Example: Setting Private Fields

In addition to discovering types, we can dynamically invoke methods and set properties:

```
let dynamicSet x propName propValue =
    let property = x.GetType().GetProperty(propName)
    property.SetValue(x, propValue, null)
```

Reflection is particularly remarkable in that it can read/write private fields, even on objects which appear to be immutable. In particular, we can explore and manipulate the underlying properties of an F# list:

```
> open System.Reflection
let x = [1;2;3;4;5]
let lastNode = x.Tail.Tail.Tail.Tail;;

val x : int list = [1; 2; 3; 4; 5]
val lastNode : int list = [5]

> lastNode.GetType().GetFields(BindingFlags.NonPublic ||| BindingFlags.Instance) |> Array.map (fun field -> field.Name);;
val it : string array = [|"__Head"; "__Tail"|]
> let tailField = lastNode.GetType().GetField("__Tail", BindingFlags.NonPublic ||| BindingFlags.Instance);;

val tailField : FieldInfo =
  Microsoft.FSharp.Collections.FSharpList`1[System.Int32] __Tail

> tailField.SetValue(lastNode, x);; (* circular list *)
val it : unit = ()
> x |> Seq.take 20 |> Seq.to_list;;
val it : int list =
  [1; 2; 3; 4; 5; 1; 2; 3; 4; 5; 1; 2; 3; 4; 5; 1; 2; 3; 4; 5]
```

The example above mutates the list in place and to produce a circularly linked list. In .NET, "immutable" doesn't really mean *immutable* and private members are mostly an illusion.

> **Note:** The power of reflection has definite security implications, but a full discussion of reflection security is far outside of the scope of this section. Readers are encouraged to visit the Security Considerations for Reflection (http://msdn.microsoft.com/en-us/library/stfy7tfc(VS.100).aspx) article on MSDN for more information.

Microsoft.FSharp.Reflection Namespace

While .NET's built-in reflection API is useful, the F# compiler performs a lot of magic which makes built-in types like unions, tuples, functions, and other built-in types appear strange using vanilla reflection. The Microsoft.FSharp.Reflection namespace (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/fsharp.core/microsoft.fsharp.reflection.html) provides a wrapper for exploring F# types.

```
open System.Reflection
open Microsoft.FSharp.Reflection

let explore x =
    let t = x.GetType()
    if FSharpType.IsTuple(t) then
        let fields =
            FSharpValue.GetTupleFields(x)
            |> Array.map string
            |> fun strings -> System.String.Join(", ", strings)

        printfn "Tuple: (%s)" fields
    elif FSharpType.IsUnion(t) then
        let union, fields =  FSharpValue.GetUnionFields(x, t)

        printfn "Union: %s(%A)" union.Name fields
    else
        printfn "Got another type"
```

Using fsi:

```
> explore (Some("Hello world"));;
Union: Some([|"Hello world"|])
val it : unit = ()

> explore (7, "Hello world");;
Tuple: (7, Hello world)
val it : unit = ()

> explore (Some("Hello world"));;
Union: Some([|"Hello world"|])
val it : unit = ()

> explore [1;2;3;4];;
Union: Cons([|1; [2; 3; 4]|])
val it : unit = ()

> explore "Hello world";;
Got another type
```

Working With Attributes

.NET attributes and reflection go hand-in-hand. Attributes allow programmers to decorate classes, methods, members, and other source code with metadata used at runtime. Many .NET classes use attributes to annotate code in a variety of ways; it is only possible to access and interpret attributes through reflection. This section will provide a brief overview of attributes. Readers interested in a more complete overview are encouraged to read MSDN's Extending Metadata With Attributes (http://msdn.microsoft.com/en-us/library/5x6cd29c(VS.100).aspx) series.

Attributes are defined using `[<AttributeName>]`, a notation already seen in a variety of places in previous chapters of this book. The .NET framework includes a number of built-in attributes, including:

- System.ObsoleteAttribute (http://msdn.microsoft.com/en-us/library/system.obsoleteattribute.aspx) - used to mark source code intended to be removed in future versions.
- System.FlagsAttribute (http://msdn.microsoft.com/en-us/library/system.flagsattribute.aspx) - indicates that an enumeration can be treated as a bit field.
- System.SerializableAttribute (http://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx) - indicates that class can be serialized.

- System.Diagnostics.DebuggerStepThroughAttribute (http://msdn.microsoft.com/en-us/library/system.diagnostics.debuggerstepthroughattribute.aspx) - indicates that the debugger should not step into a method unless it contains a break point.

We can create custom attributes by defining a new type which inherits from `System.Attribute (http://msdn.microsoft.com/en-us/library/system.attribute.aspx)`:

```
type MyAttribute(text : string) =
    inherit System.Attribute()

    do printfn "MyAttribute created. Text: %s" text

    member this.Text = text

[<MyAttribute("Hello world")>]
type MyClass() =
    member this.SomeProperty = "This is a property"
```

We can access attribute using reflection:

```
> let x = new MyClass();;

val x : MyClass

> x.GetType().GetCustomAttributes(true);;
MyAttribute created. Text: Hello world
val it : obj [] =
  [|System.SerializableAttribute {TypeId = System.SerializableAttribute;};
    FSI_0028+MyAttribute {Text = "Hello world";
                          TypeId = FSI_0028+MyAttribute;};
    Microsoft.FSharp.Core.CompilationMappingAttribute
      {SequenceNumber = 0;
       SourceConstructFlags = ObjectType;
       TypeId = Microsoft.FSharp.Core.CompilationMappingAttribute;
       VariantNumber = 0;}|]
```

The `MyAttribute` class has the side-effect of printing to the console on instantiation, demonstrating that `MyAttribute` does not get constructed when instances of `MyClass` are created.

Example: Encapsulating Singleton Design Pattern

Attributes are often used to decorate classes with any kind of ad-hoc functionality. For example, let's say we wanted to control whether single or multiple instances of classes are created based on an attribute:

```
open System
open System.Collections.Generic

[<AttributeUsage(AttributeTargets.Class)>]
type ConstructionAttribute(singleInstance : bool) =
    inherit Attribute()
    member this.IsSingleton = singleInstance

let singletons = Dictionary<System.Type,obj>()
let make<'a>() : 'a =
    let newInstance() = Activator.CreateInstance<'a>()
    let attributes = typeof<'a>.GetCustomAttributes(typeof<ConstructionAttribute>, true)
    let singleInstance =
        if attributes.Length > 0 then
            let constructionAttribute = attributes.[0] :?> ConstructionAttribute
            constructionAttribute.IsSingleton
        else false

    if singleInstance then
        match singletons.TryGetValue(typeof<'a>) with
        | true, v -> v :?> 'a
        | _ ->
            let instance = newInstance()
            singletons.Add(typeof<'a>, instance)
            instance
    else newInstance()

[<ConstructionAttribute(true)>]
type SingleOnly() =
    do printfn "SingleOnly constructor"

[<ConstructionAttribute(false)>]
type NewAlways() =
    do printfn "NewAlways constructor"

let x = make<SingleOnly>()
let x' = make<SingleOnly>()
let y = make<NewAlways>()
let y' = make<NewAlways>()

printfn "x = x': %b" (x = x')
printfn "y = y': %b" (y = y')
Console.ReadKey(true) |> ignore
```

This program outputs the following:

```
SingleOnly constructor
NewAlways constructor
NewAlways constructor
x = x': true
y = y': false
```

Using the attribute above, we've completely abstracted away the implementation details of the singleton design pattern, reducing it down to a single attribute. Its worth noting that the program above hard-codes a value of `true` or `false` into the attribute constructor; if we wanted to, we could pass a string representing a key from the application's config file and make class construction dependent on the config file.

Computation Expressions

F# : Computation Expressions

Computation expressions are easily the most difficult, yet most powerful language constructs to understand in F#.

Monad Primer

Computation expressions are inspired by Haskell monads, which in turn are inspired by the mathematical concept of monads in category theory. To avoid all of the abstract technical and mathematical theory underlying monads, a "monad" is, in very simple terms, a scary sounding word which means *execute this function and pass its return value to this other function*.

> **Note:** The designers of F# use the term "computation expression" and "workflow" because it's less obscure and daunting than the word "monad", and because *monad* and *computation expression*, while similar, are not precisely the same thing. The author of this book prefers "monad" to emphasize the parallel between the F# and Haskell (and, strictly as an aside, it's just a neat sounding five-dollar word).

Monads in Haskell

Haskell is interesting because it's a functional programming language where all statements are executed lazily, meaning Haskell doesn't compute values until they are actually needed. While this gives Haskell some unique features such as the capacity to define "infinite" data structures (http://www.haskell.org/tutorial/functions.html#tut-infinite), it also makes it hard to reason about the execution of programs since you can't guarantee that lines of code will be executed in any particular order (if at all).

Consequently, it's quite a challenge to do things which need to be executed in a sequence, which includes any form of I/O, acquiring locks objects in multithreaded code, reading/writing to sockets, and any conceivable action which has a side-effect on any memory elsewhere in our application. Haskell manages sequential operations using something called a monad, which can be used to simulate state in an immutable environment.

Visualizing Monads with F#

To visualize monads, let's take some everyday F# code written in imperative style:

```
let read_line() = System.Console.ReadLine()
let print_string(s) = printf "%s" s

print_string "What's your name? "
let name = read_line()
print_string ("Hello, " + name)
```

We can re-write the `read_line` and `print_string` functions to take an extra parameter, namely a function to execute once our computation completes. We'd end up with something that looks more like this:

```
let read_line(f) = f(System.Console.ReadLine())
let print_string(s, f) = f(printf "%s" s)

print_string("What's your name? ", fun () ->
    read_line(fun name ->
        print_string("Hello, " + name, fun () -> () ) ) )
```

If you can understand this much, then you can understand any monad.

Of course, it is perfectly reasonable to say *what masochistic reason would anyone have for writing code like that? All it does it print out "Hello, Steve" to the console!* After all, C#, Java, C++, or other languages we know and love execute code in exactly the order specified—in other words, monads solve a problem in Haskell which simply doesn't exist in imperative languages. Consequently, the monad design pattern is virtually unknown in imperative languages.

However, monads are occasionally useful for modeling computations which are difficult to capture in an imperative style.

The Maybe Monad

A well-known monad, the Maybe monad, represents a short-circuited computation which should "bail out" if any part of the computation fails. Using a simple example, let's say we wanted to write a function which asks the user for 3 integer inputs between 0 and 100 (inclusive) -- if at any point, the user enters an input which is non-numeric or falls out of our range, the entire computation should be aborted. Traditionally, we might represent this kind of program using the following:

```
let addThreeNumbers() =
    let getNum msg =
        printf "%s" msg
        // NOTE: return values from .Net methods that accept 'out' parameters are exposed to F# as tuples.
        match System.Int32.TryParse(System.Console.ReadLine()) with
        | (true, n) when n >= 0 && n <= 100 -> Some(n)
        | _ -> None

    match getNum "#1: " with
    | Some(x) ->
        match getNum "#2: " with
        | Some(y) ->
```

```
            match getNum "#3: " with
            | Some(z) -> Some(x + y + z)
            | None -> None
        | None -> None
    | None -> None
```

> **Note:** Admittedly, the simplicity of this program -- grabbing a few integers -- is ridiculous, and there are many more concise ways to write this code by grabbing all of the values up front. However, it might help to imagine that `getNum` was a relatively expensive operation (maybe it executes a query against a database, sends and receives data over a network, initializes a complex data structure), and the most efficient way to write this program requires us to bail out as soon as we encounter the *first* invalid value.

This code is very ugly and redundant. However, we can simplify this code by converting it to monadic style:

```
let addThreeNumbers() =
    let bind(input, rest) =
        match System.Int32.TryParse(input()) with
        | (true, n) when n >= 0 && n <= 100 -> rest(n)
        | _ -> None

    let createMsg msg = fun () -> printf "%s" msg; System.Console.ReadLine()

    bind(createMsg "#1: ", fun x ->
        bind(createMsg "#2: ", fun y ->
            bind(createMsg "#3: ", fun z -> Some(x + y + z) ) ) )
```

The magic is in the `bind` method. We extract the return value from our function `input` and pass it (or bind it) as the first parameter to `rest`.

Why use monads?

The code above is still quite extravagant and verbose for practical use, however monads are especially useful for modeling calculations which are difficult to capture sequentially. Multithreaded code, for example, is notoriously resistant to efforts to write in an imperative style; however it becomes remarkably concise and easy to write in monadic style. Let's modify our bind method above as follows:

```
open System.Threading
let bind(input, rest) =
    ThreadPool.QueueUserWorkItem(new WaitCallback( fun _ -> rest(input()) )) |> ignore
```

Now our bind method will execute a function in its own thread. Using monads, we can write multithreaded code in a safe, imperative style. Here's an example in fsi demonstrating this technique:

```
> open System.Threading
open System.Text.RegularExpressions

let bind(input, rest) =
    ThreadPool.QueueUserWorkItem(new WaitCallback( fun _ -> rest(input()) )) |> ignore

let downloadAsync (url : string) =
    let printMsg msg = printfn "ThreadID = %i, Url = %s, %s" (Thread.CurrentThread.ManagedThreadId) url msg
    bind( (fun () -> printMsg "Creating webclient..."; new System.Net.WebClient()), fun webclient ->
        bind( (fun () -> printMsg "Downloading url..."; webclient.DownloadString(url)), fun html ->
            bind( (fun () -> printMsg "Extracting urls..."; Regex.Matches(html, @"http://\S+") ), fun matches ->
                printMsg ("Found " + matches.Count.ToString() + " links")
            )
        )
    )

["http://www.google.com/"; "http://microsoft.com/"; "http://www.wordpress.com/"; "http://www.peta.org"] |> Seq.iter downloadAsync;;

val bind : (unit -> 'a) * ('a -> unit) -> unit
val downloadAsync : string -> unit

>
ThreadID = 5, Url = http://www.google.com/, Creating webclient...
ThreadID = 11, Url = http://microsoft.com/, Creating webclient...
ThreadID = 5, Url = http://www.peta.org, Creating webclient...
ThreadID = 11, Url = http://www.wordpress.com/, Creating webclient...
ThreadID = 5, Url = http://microsoft.com/, Downloading url...
ThreadID = 11, Url = http://www.google.com/, Downloading url...
ThreadID = 11, Url = http://www.peta.org, Downloading url...
ThreadID = 13, Url = http://www.wordpress.com/, Downloading url...
ThreadID = 11, Url = http://www.google.com/, Extracting urls...
ThreadID = 11, Url = http://www.google.com/, Found 21 links
ThreadID = 11, Url = http://www.peta.org, Extracting urls...
ThreadID = 11, Url = http://www.peta.org, Found 111 links
ThreadID = 5, Url = http://microsoft.com/, Extracting urls...
ThreadID = 5, Url = http://microsoft.com/, Found 1 links
ThreadID = 13, Url = http://www.wordpress.com/, Extracting urls...
ThreadID = 13, Url = http://www.wordpress.com/, Found 132 links
```

Its interesting to notice that Google starts downloading on thread 5 and finishes on thread 11. Additionally, thread 11 is shared between Microsoft, Peta, and Google at some point. Each time we call `bind`, we pull a thread out of .NET's threadpool, when the function returns the thread is released back to the threadpool where another thread might pick it up again—its wholly possible for async functions to hop between any number of threads throughout their lifetime.

This technique is so powerful that it's baked into F# library in the form of the async workflow.

Defining Computation Expressions

Computation expressions are fundamentally the same concept as seen above, although they hide the complexity of monadic syntax behind a thick layer of syntactic sugar. A monad is a special kind of class which must have the following methods: `Bind`, `Delay`, and `Return`.

We can rewrite our Maybe monad described earlier as follows:

```
type MaybeBuilder() =
    member this.Bind(x, f) =
        match x with
        | Some(x) when x >= 0 && x <= 100 -> f(x)
        | _ -> None
    member this.Delay(f) = f()
    member this.Return(x) = Some x
```

We can test this class in fsi:

```
> type MaybeBuilder() =
    member this.Bind(x, f) =
        printfn "this.Bind: %A" x
        match x with
        | Some(x) when x >= 0 && x <= 100 -> f(x)
        | _ -> None
    member this.Delay(f) = f()
    member this.Return(x) = Some x

let maybe = MaybeBuilder();;

type MaybeBuilder =
  class
    new : unit -> MaybeBuilder
    member Bind : x:int option * f:(int -> 'a0 option) -> 'a0 option
    member Delay : f:(unit -> 'a0) -> 'a0
    member Return : x:'a0 -> 'a0 option
  end
val maybe : MaybeBuilder

> maybe.Delay(fun () ->
    let x = 12
    maybe.Bind(Some 11, fun y ->
        maybe.Bind(Some 30, fun z ->
            maybe.Return(x + y + z)
            )
        )
    );;
this.Bind: Some 11
this.Bind: Some 30
val it : int option = Some 53

> maybe.Delay(fun () ->
    let x = 12
    maybe.Bind(Some -50, fun y ->
        maybe.Bind(Some 30, fun z ->
            maybe.Return(x + y + z)
            )
        )
    );;
this.Bind: Some -50
val it : int option = None
```

Syntax Sugar

Monads are powerful, but beyond two or three variables, the number of nested functions becomes cumbersome to work with. F# provides syntactic sugar which allows us to write the same code in a more readable fashion. Workflows are evaluated using the form `builder { comp-expr }`. For example, the following pieces of code are equivalent:

Sugared syntax	De-sugared syntax

`let maybe = new MaybeBuilder()`
`let sugared =`
`  maybe {`
`  let x = 12`
`  let! y = Some 11`
`  let! z = Some 30`
`  return x + y + z`
`  }`
	`let maybe = new MaybeBuilder()`
	`let desugared =`
	`  maybe.Delay(fun () ->`
	`  let x = 12`
	`  maybe.Bind(Some 11, fun y ->`
	`  maybe.Bind(Some 30, fun z ->`
	`  maybe.Return(x + y + z)`
	`  )`
	`  )`
	`  )`

> **Note:** You probably noticed that the sugared syntax is strikingly similar to the syntax used to declare sequence expressions, `seq { expr }`. This is not a coincidence. In the F# library, sequences are defined as computation expressions and used as such. The async workflow is another computation expression you'll encounter while learning F#.

The sugared form reads like normal F#. The code `let x = 12` behaves as expected, but what is `let!` doing? Notice that we say `let! y = Some 11`, but the value `y` has the type `int` rather than `int option`. The construct `let! y = ...` invokes a function called `maybe.Bind(x, f)`, where the value `y` is bound to parameter passed into the `f` function.

Similarly, `return ...` invokes a function called `maybe.Return(x)`. Several new keywords de-sugar to some other construct, including ones you've already seen in sequence expressions like `yield` and `yield!`, as well as new ones like `use` and `use!`.

This fsi sample shows how easy it is to use our maybe monad with computation expression syntax:

```
> type MaybeBuilder() =
    member this.Bind(x, f) =
        printfn "this.Bind: %A" x
        match x with
        | Some(x) when x >= 0 && x <= 100 -> f(x)
        | _ -> None
    member this.Delay(f) = f()
    member this.Return(x) = Some x

let maybe = MaybeBuilder();;
```

```
type MaybeBuilder =
  class
    new : unit -> MaybeBuilder
    member Bind : x:int option * f:(int -> 'a0 option) -> 'a0 option
    member Delay : f:(unit -> 'a0) -> 'a0
    member Return : x:'a0 -> 'a0 option
  end
val maybe : MaybeBuilder

> maybe {
    let x = 12
    let! y = Some 11
    let! z = Some 30
    return x + y + z
};;
this.Bind: Some 11
this.Bind: Some 30
val it : int option = Some 53

> maybe {
    let x = 12
    let! y = Some -50
    let! z = Some 30
    return x + y + z
};;
this.Bind: Some -50
val it : int option = None
```

This code does the same thing as the desugared code, only its much much easier to read.

Dissecting Syntax Sugar

According the F# spec (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec2.aspx#_Toc207785614), workflows may be defined with the following members:

Member	Description
`member Bind : M<'a> * ('a -> M<'b>) -> M<'b>`	Required member. Used to de-sugar `let!` and `do!` within computation expressions.
`member Return : 'a -> M<'a>`	Required member. Used to de-sugar `return` within computation expressions.
`member Delay : (unit -> M<'a>) -> M<'a>`	Required member. Used to ensure side effects within a computation expression are performed when expected.
`member Yield : 'a -> M<'a>`	Optional member. Used to de-sugar `yield` within computation expressions.
`member For : seq<'a> * ('a -> M<'b>) -> M<'b>`	Optional member. Used to de-sugar `for ... do ...` within computation expressions. `M<'b>` can optionally be `M<unit>`
`member While : (unit -> bool) * M<'a> -> M<'a>`	Optional member. Used to de-sugar `while ... do ...` within computation expressions. `M<'b>` can optionally be `M<unit>`
`member Using : 'a * ('a -> M<'b>) -> M<'b> when 'a :> IDisposable`	Optional member. Used to de-sugar `use` bindings within computation expressions.
`member Combine : M<'a> -> M<'a> -> M<'a>`	Optional member. Used to de-sugar sequencing within computation expressions. The first `M<'a>` can optionally be `M<unit>`
`member Zero : unit -> M<'a>`	Optional member. Used to de-sugar empty `else` branches of `if/then` within computation expressions.
`member TryWith : M<'a> -> M<'a> -> M<'a>`	Optional member. Used to de-sugar empty `try/with` bindings within computation expressions.
`member TryFinally : M<'a> -> M<'a> -> M<'a>`	Optional member. Used to de-sugar `try/finally` bindings within computation expressions.

These sugared constructs are de-sugared as follows:

Construct	De-sugared Form
`let pat = expr in cexpr`	`let pat = expr in cexpr`
`let! pat = expr in cexpr`	`b.Bind(expr, (fun pat -> cexpr))`
`return expr`	`b.Return(expr)`
`return! expr`	`b.ReturnFrom(expr)`
`yield expr`	`b.Yield(expr)`
`yield! expr`	`b.YieldFrom(expr)`
`use pat = expr in cexpr`	`b.Using(expr, (fun pat -> cexpr))`
`use! pat = expr in cexpr`	`b.Bind(expr, (fun x -> b.Using(x, fun pat -> cexpr))`
`do! expr in cexpr`	`b.Bind(expr, (fun () -> cexpr))`
`for pat in expr do cexpr`	`b.For(expr, (fun pat -> cexpr))`
`while expr do cexpr`	`b.While((fun () -> expr), b.Delay( fun () -> cexpr))`
`if expr then cexpr1 else cexpr2`	`if expr then cexpr1 else cexpr2`
`if expr then cexpr`	`if expr then cexpr else b.Zero()`
`cexpr1` `cexpr2`	`b.Combine(cexpr1, b.Delay(fun () -> cexpr2))`
`try cexpr with patn -> cexprn`	`b.TryWith(expr, fun v -> match v with (patn:ext) -> cexprn | _ raise exn)`
`try` cexpr `finally` expr	`b.TryFinally(cexpr, (fun () -> expr))`

What are Computation Expressions Used For?

F# encourages a programming style called *language oriented programming* to solve problems. In contrast to general purpose programming style, language oriented programming centers around the programmers identifying problems they want to solve, then writing domain specific mini-languages to tackle the problem, and finally solve problem in the new mini-language.

Computation expressions are one of several tools F# programmers have at their disposal for designing mini-languages.

Its surprising how often computation expressions and monad-like constructs occur in practice. For example, the Haskell User Group (http://spbhug.folding-maps.org/wiki/MonadsEn) has a collection of common and uncommon monads, including those which compute distributions of integers and parse text. Another significant example, an F# implementation of software transactional memory, is introduced on hubFS (http://hubfs.fpish.net/topic/None/57220).

Additional Resources

- Haskell.org: All About Monads (http://www.haskell.org/haskellwiki/All_About_Monads) - Another collection of monads in Haskell.

Async Workflows

F# : Async Workflows

Async workflows allow programmers to convert single-threaded code into multi-threaded code with minimal code changes.

Defining Async Workflows

Async workflows are defined using computation expression notation:

```
async { comp-exprs }
```

Here's an example using fsi:

```
> let asyncAdd x y = async { return x + y };;
val asyncAdd : int -> int -> Async<int>
```

Notice the return type of `asyncAdd`. It does not actually run a function; instead, it returns an `async<int>`, which is a special kind of wrapper around our function.

The Async Module

The Async Module (http://msdn.microsoft.com/en-us/library/ee370232.aspx) is used for operating on `async<'a>` objects. It contains several useful methods, the most important of which are:

```
member RunSynchronously : computation:Async<'T> * ?timeout:int -> 'T
```

Run the asynchronous computation and await its result. If an exception occurs in the asynchronous computation then an exception is re-raised by this function. Run as part of the default AsyncGroup.

`member Parallel : computationList:seq<Async<'T>> -> Async<'T array>`

Specify an asynchronous computation that, when run, executes all the given asynchronous computations, initially queueing each in the thread pool. If any raise an exception then the overall computation will raise an exception, and attempt to cancel the others. All the sub-computations belong to an AsyncGroup that is a subsidiary of the AsyncGroup of the outer computations.

`member Start : computation:Async<unit> -> unit`

Start the asynchronous computation in the thread pool. Do not await its result. Run as part of the default AsyncGroup

`Async.RunSynchronously` is used to run `async<'a>` blocks and wait for them to return, `Run.Parallel` automatically runs each `async<'a>` on as many processors as the CPU has, and `Async.Start` runs without waiting for the operation to complete. To use the canonical example, downloading a web page, we can write code for downloading a web page asyncronously as follows in fsi:

```
> let extractLinks url =
    async {
        let webClient = new System.Net.WebClient()

        printfn "Downloading %s" url
        let html = webClient.DownloadString(url : string)
        printfn "Got %i bytes" html.Length

        let matches = System.Text.RegularExpressions.Regex.Matches(html, @"http://\S+")
        printfn "Got %i links" matches.Count

        return url, matches.Count
    };;

val extractLinks : string -> Async<string * int>

> Async.RunSynchronously (extractLinks "http://www.msn.com/");;
Downloading http://www.msn.com/
Got 50742 bytes
Got 260 links
val it : string * int = ("http://www.msn.com/", 260)
```

async<'a> Members

`async<'a>` objects are constructed from the AsyncBuilder (http://msdn.microsoft.com/en-us/library/ee340369.aspx), which has the following important members:

`member Bind : p:Async<'a> * f:('a -> Async<'b>) -> Async<'b>` / **`let!`**

Specify an asynchronous computation that, when run, runs 'p', and when 'p' generates a result 'res', runs 'f res'.

`member Return : v:'a -> Async<'a>` / **`return`**

Specify an asynchronous computation that, when run, returns the result 'v'

In other words, `let!` executes an async workflow and binds its return value to an identifier, `return` simply returns a result, and `return!` executes an async workflow and returns its return value as a result.

These primitives allow us to compose async blocks within one another. For example, we can improve on the code above by downloading a web page asyncronously and extracting its urls asyncronously as well:

```
let extractLinksAsync html =
    async {
        return System.Text.RegularExpressions.Regex.Matches(html, @"http://\S+")
    }

let downloadAndExtractLinks url =
    async {
        let webClient = new System.Net.WebClient()
        let html = webClient.DownloadString(url : string)
        let! links = extractLinksAsync html
        return url, links.Count
    }
```

Notice that `let!` takes an `async<'a>` and binds its return value to an identifier of type `'a`. We can test this code in fsi:

```
> let links = downloadAndExtractLinks "http://www.wordpress.com/";;

val links : Async<string * int>

> Async.Run links;;
val it : string * int = ("http://www.wordpress.com/", 132)
```

What does `let!` do?

`let!` runs an `async<'a>` object on its own thread, then it immediately releases the current thread back to the threadpool. When `let!` returns, execution of the workflow will continue on the new thread, which may or may not be the same thread that the workflow started out on. As a result, async workflows tend to "hop" between threads, an interesting effect demonstrated explicitly here, but this is not generally regarded as a bad thing.

Async Extensions Methods

Async Examples

Parallel Map

Consider the function `Seq.map`. This function is synchronous, however there is no real reason why it *needs* to be synchronous, since each element can be mapped in parallel (provided we're not sharing any mutable state). Using a module extension, we can write a parallel version of `Seq.map` with minimal effort:

```
module Seq =
    let pmap f l =
        seq { for a in l -> async { return f a } }
        |> Async.Parallel
        |> Async.Run
```

Parallel mapping can have a dramatic impact on the speed of map operations. We can compare serial and parallel mapping directly using the following:

```
open System.Text.RegularExpressions
open System.Net

let download url =
    let webclient = new System.Net.WebClient()
    webclient.DownloadString(url : string)

let extractLinks html = Regex.Matches(html, @"http://\S+")

let downloadAndExtractLinks url =
    let links = (url |> download |> extractLinks)
    url, links.Count

let urls =
     [@"http://www.craigslist.com/";
     @"http://www.msn.com/";
     @"http://en.wikibooks.org/wiki/Main_Page";
     @"http://www.wordpress.com/";
     @"http://news.google.com/";]

let pmap f l =
    seq { for a in l -> async { return f a } }
    |> Async.Parallel
    |> Async.Run

let testSynchronous() = List.map downloadAndExtractLinks urls
let testAsynchronous() = pmap downloadAndExtractLinks urls

let time msg f =
    let stopwatch = System.Diagnostics.Stopwatch.StartNew()
    let temp = f()
    stopwatch.Stop()
    printfn "(%f ms) %s: %A" stopwatch.Elapsed.TotalMilliseconds msg temp

let main() =
    printfn "Start..."
    time "Synchronous" testSynchronous
    time "Asynchronous" testAsynchronous
    printfn "Done."

main()
```

This program has the following types:

```
val download : string -> string
val extractLinks : string -> MatchCollection
val downloadAndExtractLinks : string -> string * int
val urls : string list
val pmap : ('a -> 'b) -> seq<'a> -> 'b array
val testSynchronous : unit -> (string * int) list
val testAsynchronous : unit -> (string * int) array
val time : string -> (unit -> 'a) -> unit
val main : unit -> unit
```

This program outputs the following:

```
Start...
(4276.190900 ms) Synchronous: [("http://www.craigslist.com/", 185); ("http://www.msn.com/", 262);
 ("http://en.wikibooks.org/wiki/Main_Page", 190);
 ("http://www.wordpress.com/", 132); ("http://news.google.com/", 296)]
(1939.117900 ms) Asynchronous: [|("http://www.craigslist.com/", 185); ("http://www.msn.com/", 261);
  ("http://en.wikibooks.org/wiki/Main_Page", 190);
  ("http://www.wordpress.com/", 132); ("http://news.google.com/", 294)|]
Done.
```

The code using `pmap` ran about 2.2x faster because web pages are downloaded in parallel, rather than serially.

Concurrency with Functional Programming

Why Concurrency Matters

For the first 50 years of software development, programmers could take comfort in the fact that computer hardware roughly doubled in power every 18 months. If a program was slow today, one could just wait a few months and the program would run at double the speed with no change to the source code. This trend continued well into the early 2000s, where commodity desktop machines in 2003 had more processing power than the fastest supercomputers in 1993. However, after the publication of a well-known article, The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software (http://www.gotw.ca/publications/concurrency-ddj.htm) by Herb Sutter, processors have peaked at around 3.7 GHz in 2005. The theoretical cap in in computing speed is limited by the speed of light and the laws of

physics, and we've very nearly reached that limit. Since CPU designers are unable to design faster CPUs, they have turned toward designing processors with multiple cores and better support for multithreading. Programmers no longer have the luxury of their applications running twice as fast with improving hardware—the free lunch is over.

Clockrates are not getting any faster, however the amount of data businesses process each year grows exponentially (usually at a rate of 10-20% per year). To meet the growing processing demands of business, the future of all software development is tending toward the development of highly parallel, multithreaded applications which take advantage of multicores processors, distributed systems, and cloud computing.

Problems with Mutable State

Multithreaded programming has a reputation for being notoriously difficult to get right and having a rather steep learning curve. Why does it have this reputation? To put it simply, mutable shared state makes programs difficult to reason about. When two threads are mutating the same variables, it is very easy to put the variable in an invalid state.

Race Conditions

As a demonstration, here's how to increment a global variable using shared state (non-threaded version):

```
let test() =
    let counter = ref 0m

    let IncrGlobalCounter numberOfTimes =
        for i in 1 .. numberOfTimes do
            counter := !counter + 1m

    IncrGlobalCounter 1000000
    IncrGlobalCounter 1000000

    !counter // returns 2000000M
```

This works, but some programmer might notice that both calls to `IncrGlobalCounter` could be computed in parallel since there's no real reason to wait for one call to finish before the other. Using the .NET threading primitives in the System.Threading (http://msdn.microsoft.com/en-us/library/system.threading.aspx) namespace, a programmer can re-write this as follows:

```
open System.Threading

let testAsync() =
    let counter = ref 0m

    let IncrGlobalCounter numberOfTimes =
        for i in 1 .. numberOfTimes do
            counter := !counter + 1m

    let AsyncIncrGlobalCounter numberOfTimes =
        new Thread(fun () -> IncrGlobalCounter(numberOfTimes))

    let t1 = AsyncIncrGlobalCounter 1000000
    let t2 = AsyncIncrGlobalCounter 1000000
    t1.Start() // runs t1 asyncronously
    t2.Start() // runs t2 asyncronously
    t1.Join()  // waits until t1 finishes
    t2.Join()  // waits until t2 finishes

    !counter
```

This program *should* do the same thing as the previous program, only it should run in ~1/2 the time. Here are the results of 5 test runs in fsi:

```
> [for a in 1 .. 5 -> testAsync()];;
val it : decimal list = [1498017M; 1509820M; 1426922M; 1504574M; 1420401M]
```

The program is computationally sound, but it produces a different result everytime its run. What happened?

It takes several machine instructions increment a decimal value. In particular, the .NET IL for incrementing a decimal looks like this:

```
// pushes static field onto evaluation stack
L_0004: ldsfld valuetype [mscorlib]System.Decimal ConsoleApplication1.Program::i

// executes Decimal.op_Increment method
L_0009: call valuetype [mscorlib]System.Decimal [mscorlib]System.Decimal::op_Increment(valuetype [mscorlib]System.Decimal)

// replaces static field with value from evaluation stack
L_000e: stsfld valuetype [mscorlib]System.Decimal ConsoleApplication1.Program::i
```

Imagine that we have two threads calling this code (calls made by Thread1 and Thread2 are interleaved):

```
Thread1: Loads value "100" onto its evaluation stack.
Thread1: Call add with "100" and "1"
Thread2: Loads value "100" onto its evaluation stack.
Thread1: Writes "101" back out to static variable
Thread2: Call add with "100" and "1"
Thread2: Writes "101" back out to static variable (Oops, we've incremented an old value and wrote it back out)
Thread1: Loads value "101" onto its evaluation stack.
Thread2: Loads value "101" onto its evaluation stack.

(Now we let Thread1 get a little further ahead of Thread2)
Thread1: Call add with "101" and "1"
Thread1: Writes "102" back out to static variable.
Thread1: Loads value "102" to evaluation stack
Thread1: Call add with "102" and "1"
Thread1: Writes "103" back out to static variable.
Thread2: Call add with "101" and "1
Thread2: Writes "102" back out to static variable (Oops, now we've completely overwritten work done by Thread1!)
```

This kind of bug is called a *race condition* and it occurs all the time in multithreaded applications. Unlike normal bugs, race-conditions are often non-deterministic, making them extremely difficult to track down.

Usually, programmers solve race conditions by introducing locks. When an object is "locked", all other threads are forced to wait until the object is "unlocked" before they proceed. We can re-write the code above using a block access to the `counter` variable while each thread writes to it:

```
open System.Threading

let testAsync() =
    let counter = ref 0m
    let IncrGlobalCounter numberOfTimes =
        for i in 1 .. numberOfTimes do
            lock counter (fun () -> counter := !counter + 1m)
            (* lock is a function in F# library. It automatically unlocks "counter" when 'fun () -> ...' completes *)

    let AsyncIncrGlobalCounter numberOfTimes =
        new Thread(fun () -> IncrGlobalCounter(numberOfTimes))

    let t1 = AsyncIncrGlobalCounter 1000000
    let t2 = AsyncIncrGlobalCounter 1000000
    t1.Start() // runs t1 asyncronously
    t2.Start() // runs t2 asyncronously
    t1.Join()  // waits until t1 finishes
    t2.Join()  // waits until t2 finishes

    !counter
```

The lock guarantees each thread exclusive access to shared state and forces each thread to wait on the other while the code `counter := !counter + 1m` runs to completion. This function now produces the expected result.

Deadlocks

Locks force threads to wait until an object is unlocked. However, locks often lead to a new problem: Let's say we have ThreadA and ThreadB which operate on two corresponding pieces of shared state, StateA and StateB. ThreadA locks stateA and stateB, ThreadB locks stateB and stateA. If the timing is right, when ThreadA needs to access stateB, its waits until ThreadB unlocks stateB; when ThreadB needs to access stateA, it can't proceed either since stateA is locked by ThreadA. Both threads mutually block one another, and they are unable to proceed any further. This is called a deadlock.

Here's some simple code which demonstrates a deadlock:

```
open System.Threading

let testDeadlock() =
    let stateA = ref "Shared State A"
    let stateB = ref "Shared State B"

    let threadA = new Thread(fun () ->
        printfn "threadA started"
        lock stateA (fun () ->
            printfn "stateA: %s" !stateA
            Thread.Sleep(100) // pauses thread for 100 ms. Simimulates busy processing
            lock stateB (fun () -> printfn "stateB: %s" !stateB))
        printfn "threadA finished")

    let threadB = new Thread(fun () ->
        printfn "threadB started"
        lock stateB (fun () ->
            printfn "stateB: %s" !stateB
            Thread.Sleep(100) // pauses thread for 100 ms. Simimulates busy processing
            lock stateA (fun () -> printfn "stateA: %s" !stateA))
        printfn "threadB finished")

    printfn "Starting..."
    threadA.Start()
    threadB.Start()
    threadA.Join()
    threadB.Join()
    printfn "Finished..."
```

These kinds of bugs occur all the time in multithreaded code, although they usually aren't quite as explicit as the code shown above.

Why Functional Programming Matters

To put it bluntly, mutable state is enemy of multithreaded code. Functional programming often simplifies multithreading tremendously: since values are immutable by default, programmers don't need to worry about one thread mutating the value of shared state between two threads, so it eliminates a whole class of multithreading bugs related to race conditions. Since there are no race conditions, there's no reason to use locks either, so immutability eliminates another whole class of bugs related to deadlocks as well.

MailboxProcessor

F# : MailboxProcessor Class

F#'s **MailboxProcessor class** is essentially a dedicated message queue running on its own logical thread of control. Any thread can send the MailboxProcessor a message asynchronously or synchronously, allowing threads to communicate between one another through message passing. The "thread" of control is actually a lightweight simulated thread implemented via asynchronous reactions to messages. This style of message-passing concurrency is inspired by the Erlang programming language.

Defining MailboxProcessors

MailboxProcessors are created using the MailboxProcessor.Start (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/FSharp.Core/Microsoft.FSharp.Control.type_MailboxProcessor-1.html) method which has the type `Start : initial:(MailboxProcessor<'msg> -> Async<unit>) * ?asyncGroup:AsyncGroup -> MailboxProcessor<'msg>`:

```
let counter =
    MailboxProcessor.Start(fun inbox ->
        let rec loop n =
            async { do printfn "n = %d, waiting..." n
                    let! msg = inbox.Receive()
                    return! loop(n+msg) }
        loop 0)
```

The value `inbox` has the type `MailboxProcessor<'msg>` and represents the message queue. The method `inbox.Receive()` dequeues the first message from the message queue and binds it to the `msg` identifier. If there are no messages in queue, `Receive` releases the thread back to the threadpool and waits for further messages. No threads are blocked while `Receive` waits for further messages.

We can experiment with `counter` in fsi:

```
> let counter =
    MailboxProcessor.Start(fun inbox ->
        let rec loop n =
            async { do printfn "n = %d, waiting..." n
                    let! msg = inbox.Receive()
                    return! loop(n+msg) }
        loop 0);;

val counter : MailboxProcessor<int>

n = 0, waiting...
> counter.Post(5);;
n = 5, waiting...
val it : unit = ()
> counter.Post(20);;
n = 25, waiting...
val it : unit = ()
> counter.Post(10);;
n = 35, waiting...
val it : unit = ()
```

MailboxProcessor Methods

There are several useful methods in the MailboxProcessor class:

`static member Start : initial:(MailboxProcessor<'msg> -> Async<unit>) * ?asyncGroup:AsyncGroup -> MailboxProcessor<'msg>`

> Create and start an instance of a MailboxProcessor. The asynchronous computation executed by the processor is the one returned by the 'initial' function.

`member Post : message:'msg -> unit`

> Post a message to the message queue of the MailboxProcessor, asynchronously.

`member PostAndReply : buildMessage:(AsyncReplyChannel<'reply> -> 'msg) * ?timeout:int * ?exitContext:bool -> 'reply`

> Post a message to the message queue of the MailboxProcessor and await a reply on the channel. The message is produced by a single call to the first function which must build a message containing the reply channel. The receiving MailboxProcessor must process this message and invoke the Reply method on the reply channel precisely once.

`member Receive : ?timeout:int -> Async<'msg>`

> Return an asynchronous computation which will consume the first message in arrival order. No thread is blocked while waiting for further messages. Raise a TimeoutException if the timeout is exceeded.

Two-way Communication

Just as easily as we can send messages to MailboxProcessors, a MailboxProcessor can send replies back to consumers. For example, we can interrogate the value of a MailboxProcessor using the `PostAndReply` method as follows:

```
type msg =
    | Incr of int
    | Fetch of AsyncReplyChannel<int>

let counter =
    MailboxProcessor.Start(fun inbox ->
        let rec loop n =
            async { let! msg = inbox.Receive()
                    match msg with
                    | Incr(x) -> return! loop(n + x)
                    | Fetch(replyChannel) ->
                       replyChannel.Reply(n)
                       return! loop(n) }
        loop 0)
```

The `msg` union wraps two types of messages: we can tell the MailboxProcessor to increment, or have it send its contents to a reply channel. The type `AsyncReplyChannel<'a> (http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/FSharp.Core/Microsoft.FSharp.Control.type_AsyncReplyChannel-1.html)` exposes a single method, `member Reply : 'reply -> unit`. We can use this class in fsi as follows:

```
> counter.Post(Incr 7);;
val it : unit = ()
> counter.Post(Incr 50);;
val it : unit = ()
> counter.PostAndReply(fun replyChannel -> Fetch replyChannel);;
val it : int = 57
```

Notice that `PostAndReply` is a syncronous method.

Encapsulating MailboxProcessors with Objects

Often, we don't want to expose the implementation details of our classes to consumers. For example, we can re-write the example above as a class which exposes a few select methods:

```
type countMsg =
    | Die
    | Incr of int
    | Fetch of AsyncReplyChannel<int>

type counter() =
    let innerCounter =
        MailboxProcessor.Start(fun inbox ->
            let rec loop n =
                async { let! msg = inbox.Receive()
                        match msg with
                        | Die -> return ()
                        | Incr x -> return! loop(n + x)
                        | Fetch(reply) ->
                            reply.Reply(n);
                            return! loop n }
            loop 0)

    member this.Incr(x) = innerCounter.Post(Incr x)
    member this.Fetch() = innerCounter.PostAndReply((fun reply -> Fetch(reply)), timeout = 2000)
    member this.Die() = innerCounter.Post(Die)
```

MailboxProcessor Examples

Prime Number Sieve

Rob Pike delivered a fascinating presentation (https://www.youtube.com/watch?v=hB05UFqOtFA) at a Google TechTalk on the NewSqueak programming language. NewSqueak's approach to concurrency uses channels, analogous to MailboxProcessors, for inter-thread communication. Toward the end of the presentation, he demonstrates how to implement a prime number sieve using these channels. The following is an implementation of prime number sieve based on Pike's NewSqueak code:

```
type 'a seqMsg =
    | Die
    | Next of AsyncReplyChannel<'a>

type primes() =
    let counter(init) =
        MailboxProcessor.Start(fun inbox ->
            let rec loop n =
                async { let! msg = inbox.Receive()
                        match msg with
                        | Die -> return ()
                        | Next(reply) ->
                            reply.Reply(n)
                            return! loop(n + 1) }
            loop init)

    let filter(c : MailboxProcessor<'a seqMsg>, pred) =
        MailboxProcessor.Start(fun inbox ->
            let rec loop() =
                async {
                    let! msg = inbox.Receive()
                    match msg with
                    | Die ->
                        c.Post(Die)
                        return()
                    | Next(reply) ->
                        let rec filter' n =
                            if pred n then async { return n }
                            else
                                async {let! m = c.PostAndAsyncReply(Next)
                                       return! filter' m }
                        let! testItem = c.PostAndAsyncReply(Next)
                        let! filteredItem = filter' testItem
                        reply.Reply(filteredItem)
                        return! loop()
                }
            loop()
        )

    let processor = MailboxProcessor.Start(fun inbox ->
        let rec loop (oldFilter : MailboxProcessor<int seqMsg>) prime =
            async {
                let! msg = inbox.Receive()
                match msg with
                | Die ->
                    oldFilter.Post(Die)
                    return()
```

```
                | Next(reply) ->
                    reply.Reply(prime)
                    let newFilter = filter(oldFilter, (fun x -> x % prime <> 0))
                    let! newPrime = oldFilter.PostAndAsyncReply(Next)
                    return! loop newFilter newPrime
            }
        loop (counter(3)) 2)

    member this.Next() = processor.PostAndReply( (fun reply -> Next(reply)), timeout = 2000)

    interface System.IDisposable with
        member this.Dispose() = processor.Post(Die)

    static member upto max =
        [ use p = new primes()
          let lastPrime = ref (p.Next())
          while !lastPrime <= max do
            yield !lastPrime
            lastPrime := p.Next() ]
```

`counter` represents an infinite list of numbers from n..infinity.

`filter` is simply a filter for another MailboxProcessor. Its analogous to `Seq.filter`.

`processor` is essentially an iterative filter: we seed our prime list with the first prime, `2` and a infinite list of numbers from 3..infinity. Each time we process a message, we return the prime number, then replace our infinite list with a new list which filters out all numbers divisible by our prime. The head of each new filtered list is the next prime number.

So, the first time we call `Next`, we get back a 2 and replace our infinite list with all numbers not divisible by two. We call next again, we get back the next prime, 3, and filter our list again for all numbers divisible by 3. We call next again, we get back the next prime, 5 (we skip 4 since its divisible by 2), and filter all numbers divisible by 5. This process repeats indefinitely. The end result is a prime number sieve with an identical implementation to the Sieve of Eratosthenes.

We can test this class in fsi:

```
> let p = new primes();;

val p : primes

> p.Next();;
val it : int = 2
> p.Next();;
val it : int = 3
> p.Next();;
val it : int = 5
> p.Next();;
val it : int = 7
> p.Next();;
val it : int = 11
> p.Next();;
val it : int = 13
> p.Next();;
val it : int = 17
> primes.upto 100;;
val it : int list
= [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
   73; 79; 83; 89; 97]
```

Lexing and Parsing

F# : Lexing and Parsing

Lexing and parsing is a very handy way to convert source-code (or other human-readable input which has a well-defined syntax) into an abstract syntax tree (AST) which represents the source-code. F# comes with two tools, FsLex and FsYacc, which are used to convert input into an AST.

FsLex and FsYacc have more or less the same specification as OCamlLex and OCamlYacc (http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html), which in turn are based on the Lex and Yacc (http://dinosaur.compilertools.net/) family of lexer/parser generators. Virtually all material concerned with OCamlLex/OCamlYacc can transfer seamlessly over to FsLex/FsYacc. With that in mind, SooHyoung Oh's OCamlYacc tutorial (http://courses.softlab.ntua.gr/compilers/2015a/ocamlyacc-tutorial.pdf) and companion OCamlLex Tutorial (http://courses.softlab.ntua.gr/compilers/2015a/ocamllex-tutorial.pdf) are the single best online resources to learn how to use the lexing and parsing tools which come with F# (and OCaml for that matter!).

Lexing and Parsing from a High-Level View

Transforming input into a well-defined abstract syntax tree requires (at minimum) two transformations:

1. A lexer uses regular expressions to convert each syntactical element from the input into a token, essentially mapping the input to a stream of tokens.
2. A parser reads in a stream of tokens and attempts to match tokens to a set of rules, where the end result maps the token stream to an abstract syntax tree.

It is certainly possible to write a lexer which generates the abstract syntax tree directly, but this only works for the most simplistic grammars. If a grammar contains balanced parentheses or other recursive constructs, optional tokens, repeating groups of tokens, operator precedence, or anything which can't be captured by regular expressions, then it is easiest to write a parser in addition to a lexer.

With F#, it is possible to write custom file formats, domain specific languages, and even full-blown compilers for your new language.

Extended Example: Parsing SQL

The following code will demonstrate step-by-step how to define a simple lexer/parser for a subset of SQL. If you're using Visual Studio, you should add a reference to `FSharp.PowerPack.dll` to your project. If you're compiling on the commandline, use the `-r` flag to reference the aforemented F# powerpack assembly.

Step 1: Define the Abstract Syntax Tree

We want to parse the following SQL statement:

```
SELECT x, y, z
FROM t1
LEFT JOIN t2
INNER JOIN t3 ON t3.ID = t2.ID
WHERE x = 50 AND y = 20
ORDER BY x ASC, y DESC, z
```

This is a pretty simple query, and while it doesn't demonstrate everything you can do with the language, it's a good start. We can model everything in this query using the following definitions in F#:

```
// Sql.fs

type value =
    | Int of int
    | Float of float
    | String of string

type dir = Asc | Desc
type op = Eq | Gt | Ge | Lt | Le    // =, >, >=, <, <=

type order = string * dir

type where =
    | Cond of (value * op * value)
    | And of where * where
    | Or of where * where

type joinType = Inner | Left | Right

type join = string * joinType * where option    // table name, join, optional "on" clause

type sqlStatement =
    { Table : string;
        Columns : string list;
        Joins : join list;
        Where : where option;
        OrderBy : order list }
```

A record type neatly groups all of our related values into a single object. When we finish our parser, it should be able to convert a string in an object of type `sqlStatement`.

Step 2: Define the parser tokens

A token is any single identifiable element in a grammar. Let's look at the string we're trying to parse:

```
SELECT x, y, z
FROM t1
LEFT JOIN t2
INNER JOIN t3 ON t3.ID = t2.ID
WHERE x = 50 AND y = 20
ORDER BY x ASC, y DESC, z
```

So far, we have several keywords (by convention, all keywords are uppercase): `SELECT`, `FROM`, `LEFT`, `INNER`, `RIGHT`, `JOIN`, `ON`, `WHERE`, `AND`, `OR`, `ORDER`, `BY`, `ASC`, `DESC`, and `COMMA`.

There are also a few comparison operators: `EQ`, `GT`, `GE`, `LT`, `LE`.

We also have non-keyword identifiers composed of strings and numeric literals. which we'll represent using the keyword `ID`, `INT`, `FLOAT`.

Finally, there is one more token, `EOF`, which indicates the end of our input stream.

Now we can create a basic parser file for FsYacc, name the file SqlParser.fsp:

```
%{
open Sql
%}

%token <string> ID
%token <int> INT
%token <float> FLOAT

%token AND OR
%token COMMA
%token EQ LT LE GT GE
%token JOIN INNER LEFT RIGHT ON
%token SELECT FROM WHERE ORDER BY
%token ASC DESC
%token EOF

// start
```

```
%start start
%type <string> start

%%

start:
    |                           { "Nothing to see here" }
%%
```

This is boilerplate code with the section for tokens filled in.

Compile the parser using the following command line: `fsyacc SqlParser.fsp`

Tips:

- If you haven't done so already, you should add the F# bin directory to your PATH environment variable.
- If you're using Visual Studio, you can automatically generate your parser code on each compile. Right-click on your project file and choose "Properties". Navigate to the Build Events tab, and add the following to the 'Pre-build event command line' and use the following: `fsyacc "$(ProjectDir)SqlParser.fsp"`. Also remember to exclude this file from the build process: right-click the file, choose "Properties" and select "None" against "Build Action".

If everything works, FsYacc will generate two files, SqlParser.fsi and SqlParser.fs. You'll need to add these files to your project if they don't already exist. If you open the SqlParser.fsi file, you'll notice the tokens you defined in your .fsl file have been converted into a union type.

Step 3: Defining the lexer rules

Lexers convert text inputs into a stream of tokens. We can start with the following boiler plate code:

```
{
// header: any valid F# can appear here.
open Lexing
}

// regex macros
let char        = ['a'-'z' 'A'-'Z']
let digit       = ['0'-'9']
let int         = '-'?digit+
let float       = '-'?digit+ '.' digit+
let whitespace  = [' ' '\t']
let newline     = "\n\r" | '\n' | '\r'

// rules
rule tokenize = parse
| whitespace    { tokenize lexbuf }
| newline       { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf; }
| eof           { () }
```

This is not "real" F# code, but rather a special language used by FsLex.

The `let` bindings at the top of the file are used to define regular expression macros. `eof` is a special marker used to identify the end of a string buffer input.

`rule ... = parse ...` defines our lexing function, called `tokenize` above. Our lexing function consists of a series of rules, which has two pieces: 1) a regular expression, 2) an expression to evaluate if the regex matches, such as returning a token. Text is read from the token stream one character at a time until it matches a regular expression and returns a token.

We can fill in the remainder of our lexer by adding more matching expressions:

```
{
open Lexing

// opening the SqlParser module to give us access to the tokens it defines
open SqlParser
}

let char        = ['a'-'z' 'A'-'Z']
let digit       = ['0'-'9']
let int         = '-'?digit+
let float       = '-'?digit+ '.' digit+
let identifier  = char(char|digit|['-' '_' '.'])*
let whitespace  = [' ' '\t']
let newline     = "\n\r" | '\n' | '\r'

rule tokenize = parse
| whitespace    { tokenize lexbuf }
| newline       { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf; }
| int           { INT(Int32.Parse(lexeme lexbuf)) }
| float         { FLOAT(Double.Parse(lexeme lexbuf)) }
| ','           { COMMA }
| eof           { EOF }
```

Notice the code between the `{`'s and `}`'s consists of plain old F# code. Also notice we are returning the same tokens (`INT`, `FLOAT`, `COMMA` and `EOF`) that we defined in SqlParser.fsp. As you can probably infer, the code `lexeme lexbuf` returns the string our parser matched. The `tokenize` function will be converted into function which has a return type of `SqlParser.token`.

We can fill in the rest of the lexer rules fairly easily:

```
{
open System
open SqlParser
open Lexing

let keywords =
    [
```

```
        "SELECT", SELECT;
        "FROM", FROM;
        "WHERE", WHERE;
        "ORDER", ORDER;
        "BY", BY;
        "JOIN", JOIN;
        "INNER", INNER;
        "LEFT", LEFT;
        "RIGHT", RIGHT;
        "ASC", ASC;
        "DESC", DESC;
        "AND", AND;
        "OR", OR;
        "ON", ON;
    ] |> Map.ofList

let ops =
    [
        "=",    EQ;
        "<",    LT;
        "<=",   LE;
        ">",    GT;
        ">=",   GE;
    ] |> Map.ofList
}

let char        = ['a'-'z' 'A'-'Z']
let digit       = ['0'-'9']
let int         = '-'?digit+
let float       = '-'?digit+ '.' digit+
let identifier  = char(char|digit|['-' '_' '.'])*
let whitespace  = [' ' '\t']
let newline     = "\n\r" | '\n' | '\r'
let operator    = ">" | ">=" | "<" | "<=" | "="

rule tokenize = parse
| whitespace    { tokenize lexbuf }
| newline       { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf; }
| int           { INT(Int32.Parse(lexeme lexbuf)) }
| float         { FLOAT(Double.Parse(lexeme lexbuf)) }
| operator      { ops.[lexeme lexbuf] }
| identifier    { match keywords.TryFind(lexeme lexbuf) with
                  | Some(token) -> token
                  | None -> ID(lexeme lexbuf) }
| ','           { COMMA }
| eof           { EOF }
```

Notice we've created a few maps, one for keywords and one for operators. While we certainly can define these as rules in our lexer, its generally recommended to have a very small number of rules to avoid a "state explosion".

To compile this lexer, execute the following code on the commandline: `fslex SqlLexer.fsl`. (Try adding this file to your project's Build Events as well.) Then, add the file SqlLexer.fs to the project. We can experiment with the lexer now with some sample input:

```
open System
open Sql

let x = "
    SELECT x, y, z
    FROM t1
    LEFT JOIN t2
    INNER JOIN t3 ON t3.ID = t2.ID
    WHERE x = 50 AND y = 20
    ORDER BY x ASC, y DESC, z
"

let lexbuf = Lexing.from_string x
while not lexbuf.IsPastEndOfStream do
    printfn "%A" (SqlLexer.tokenize lexbuf)

Console.WriteLine("(press any key)")
Console.ReadKey(true) |> ignore
```

This program will print out a list of tokens matched by the string above.

Step 4: Define the parser rules

A parser converts a stream of tokens into an abstract syntax tree. We can modify our boilerplate parser as follows (will not compile):

```
%{
open Sql
%}

%token <string> ID
%token <int> INT
%token <float> FLOAT

%token AND OR
%token COMMA
%token EQ LT LE GT GE
%token JOIN INNER LEFT RIGHT ON
%token SELECT FROM WHERE ORDER BY
%token ASC DESC
%token EOF

// start
%start start
%type <Sql.sqlStatement> start

%%

start:  SELECT columnList
        FROM ID
        joinList
        whereClause
        orderByClause
```

```
        EOF                 {
                                { Table = $4;
                                  Columns = $2;
                                  Joins = $5;
                                  Where = $6;
                                  OrderBy = $7 }
                            }

value:
    | INT                   { Int($1) }
    | FLOAT                 { Float($1) }
    | ID                    { String($1) }
%%
```

Let's examine the `start:` function. You can immediately see that we have a list of tokens which gives a rough outline of a select statement. In addition to that, you can see the F# code contained between `{`'s and `}`'s which will be executed when the code successfully matches—in this case, its returning an instance of the `Sql.sqlStatement` record.

The F# code contains "$1", "$2", :$3", etc. which vaguely resembles regex replace syntax. Each "$#" corresponds to the index (starting at 1) of the token in our matching rule. The indexes become obvious when they're annotated as follows:

```
(*      $1     $2          *)
start:  SELECT columnList

(*      $3   $4            *)
        FROM ID

(*      $5                 *)
        joinList

(*      $6                 *)
        whereClause

(*      $7                 *)
        orderByClause

(*      $8                 *)
        EOF                 {
                                { Table = $4;
                                  Columns = $2;
                                  Joins = $5;
                                  Where = $6;
                                  OrderBy = $7 }
                            }
```

So, the `start` rule breaks our tokens into a basic shape, which we then use to map to our `sqlStatement` record. You're probably wondering where the `columnList`, `joinList`, `whereClause`, and `orderByClause` come from—these are not tokens, but are rather additional parse rules which we'll have to define. Let's start with the first rule:

```
columnList:
    | ID                    { [$1]}
    | ID COMMA columnList    { $1 :: $3 }
```

`columnList` matches text in the style of "`a, b, c, ... z`" and returns the results as a list. Notice this rule is defined recursively (also notice the order of rules is not significant). FsYacc's match algorithm is "greedy", meaning it will try to match as many tokens as possible. When FsYacc receives an `ID` token, it will match the first rule, but it also matches part of the second rule as well. FsYacc then performs a one-token lookahead: it the next token is a `COMMA`, then it will attempt to match additional tokens until the full rule can be satisfied.

> **Note:** The definition of columnList above is not tail recursive, so it may throw a stack overflow exception for exceptionally large inputs. A tail recursive version of this rule can be defined as follows:
>
> ```
> start:  ...
>         EOF                 {
>                                 { Table = $4;
>                                   Columns = List.rev $2;
>                                   Joins = $5;
>                                   Where = $6;
>                                   OrderBy = $7 }
>                             }
>
> columnList:
>     | ID                    { [$1]}
>     | columnList COMMA ID   { $3 :: $1 }
> ```
>
> The tail-recursive version creates the list backwards, so we have to reverse when we return our final output from the parser.

We can treat the JOIN clause in the same way, however its a little more complicated:

```
// join clause
joinList:
    |                               { [] } // empty rule, matches 0 tokens
    | joinClause                    { [$1] }
    | joinClause joinList           { $1 :: $2 }

joinClause:
    | INNER JOIN ID joinOnClause    { $3, Inner, $4 }
    | LEFT JOIN ID joinOnClause     { $3, Left, $4 }
    | RIGHT JOIN ID joinOnClause    { $3, Right, $4 }

joinOnClause:
    |                               { None }
    | ON conditionList              { Some($2) }

conditionList:
    | value op value                    { Cond($1, $2, $3) }
    | value op value AND conditionList  { And(Cond($1, $2, $3), $5) }
    | value op value OR conditionList   { Or(Cond($1, $2, $3), $5) }
```

`joinList` is defined in terms of several functions. This results because there are repeating groups of tokens (such as multiple tables being joined) and optional tokens (the optional "ON" clause). You've already seen that we handle repeating groups of tokens using recursive rules. To handle optional tokens, we simply break the optional syntax into a separate function, and create an empty rule to represent 0 tokens.

With this strategy in mind, we can write the remaining parser rules:

```
%{
open Sql
%}

%token <string> ID
%token <int> INT
%token <float> FLOAT

%token AND OR
%token COMMA
%token EQ LT LE GT GE
%token JOIN INNER LEFT RIGHT ON
%token SELECT FROM WHERE ORDER BY
%token ASC DESC
%token EOF

// start
%start start
%type <Sql.sqlStatement> start

%%

start:  SELECT columnList
        FROM ID
        joinList
        whereClause
        orderByClause
        EOF                 {
                                { Table = $4;
                                  Columns = List.rev $2;
                                  Joins = $5;
                                  Where = $6;
                                  OrderBy = $7 }
                            }
columnList:
    | ID                    { [$1]}
    | columnList COMMA ID   { $3 :: $1 }

// join clause
joinList:
    |                               { [] }
    | joinClause                    { [$1] }
    | joinClause joinList           { $1 :: $2 }

joinClause:
    | INNER JOIN ID joinOnClause    { $3, Inner, $4 }
    | LEFT JOIN ID joinOnClause     { $3, Left, $4 }
    | RIGHT JOIN ID joinOnClause    { $3, Right, $4 }

joinOnClause:
    |                               { None }
    | ON conditionList              { Some($2) }

conditionList:
    | value op value                    { Cond($1, $2, $3) }
    | value op value AND conditionList  { And(Cond($1, $2, $3), $5) }
    | value op value OR conditionList   { Or(Cond($1, $2, $3), $5) }

// where clause
whereClause:
    |                               { None }
    | WHERE conditionList           { Some($2) }

op: EQ { Eq } | LT { Lt } | LE { Le } | GT { Gt } | GE { Ge }

value:
    | INT                   { Int($1) }
    | FLOAT                 { Float($1) }
    | ID                    { String($1) }

// order by clause

orderByClause:
    |                               { [] }
    | ORDER BY orderByList          { $3 }

orderByList:
    | orderBy                       { [$1] }
    | orderBy COMMA orderByList     { $1 :: $3 }

orderBy:
    | ID                            { $1, Asc }
    | ID ASC                        { $1, Asc }
    | ID DESC                       { $1, Desc}

%%
```

Step 5: Piecing Everything Together

Here is the complete code for our lexer/parser:

SqlParser.fsp

```
%{
open Sql
%}

%token <string> ID
```

```
%token <int> INT
%token <float> FLOAT

%token AND OR
%token COMMA
%token EQ LT LE GT GE
%token JOIN INNER LEFT RIGHT ON
%token SELECT FROM WHERE ORDER BY
%token ASC DESC
%token EOF

// start
%start start
%type <Sql.sqlStatement> start

%%

start:  SELECT columnList
        FROM ID
        joinList
        whereClause
        orderByClause
        EOF                 {
                                { Table = $4;
                                  Columns = List.rev $2;
                                  Joins = $5;
                                  Where = $6;
                                  OrderBy = $7 }
                            }
columnList:
    | ID                    { [$1]}
    | columnList COMMA ID   { $3 :: $1 }

// join clause
joinList:
    |                               { [] }
    | joinClause                    { [$1] }
    | joinClause joinList           { $1 :: $2 }

joinClause:
    | INNER JOIN ID joinOnClause    { $3, Inner, $4 }
    | LEFT JOIN ID joinOnClause     { $3, Left, $4 }
    | RIGHT JOIN ID joinOnClause    { $3, Right, $4 }

joinOnClause:
    |                               { None }
    | ON conditionList              { Some($2) }

conditionList:
    | value op value                    { Cond($1, $2, $3) }
    | value op value AND conditionList  { And(Cond($1, $2, $3), $5) }
    | value op value OR conditionList   { Or(Cond($1, $2, $3), $5) }

// where clause
whereClause:
    |                               { None }
    | WHERE conditionList           { Some($2) }

op: EQ { Eq } | LT { Lt } | LE { Le } | GT { Gt } | GE { Ge }

value:
    | INT                   { Int($1) }
    | FLOAT                 { Float($1) }
    | ID                    { String($1) }

// order by clause

orderByClause:
    |                               { [] }
    | ORDER BY orderByList          { $3 }

orderByList:
    | orderBy                       { [$1] }
    | orderBy COMMA orderByList     { $1 :: $3 }

orderBy:
    | ID                            { $1, Asc }
    | ID ASC                        { $1, Asc }
    | ID DESC                       { $1, Desc}

%%
```

SqlLexer.fsl

```
{
open System
open SqlParser
open Lexing

let keywords =
    [
        "SELECT", SELECT;
        "FROM", FROM;
        "WHERE", WHERE;
        "ORDER", ORDER;
        "BY", BY;
        "JOIN", JOIN;
        "INNER", INNER;
        "LEFT", LEFT;
        "RIGHT", RIGHT;
        "ASC", ASC;
        "DESC", DESC;
        "AND", AND;
        "OR", OR;
        "ON", ON;
    ] |> Map.of_list

let ops =
    [
        "=",    EQ;
        "<",    LT;
        "<=",   LE;
```

```
        ">",    GT;
        ">=",   GE;
    ] |> Map.of_list
}

let char        = ['a'-'z' 'A'-'Z']
let digit       = ['0'-'9']
let int         = '-'?digit+
let float       = '-'?digit+ '.' digit+
let identifier  = char(char|digit|['-' '_' '.'])*
let whitespace  = [' ' '\t']
let newline     = "\n\r" | '\n' | '\r'
let operator    = ">" | ">=" | "<" | "<=" | "="

rule tokenize = parse
| whitespace    { tokenize lexbuf }
| newline       { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf; }
| int           { INT(Int32.Parse(lexeme lexbuf)) }
| float         { FLOAT(Double.Parse(lexeme lexbuf)) }
| operator      { ops.[lexeme lexbuf] }
| identifier    { match keywords.TryFind(lexeme lexbuf) with
                  | Some(token) -> token
                  | None -> ID(lexeme lexbuf) }
| ','           { COMMA }
| eof           { EOF }
```

Program.fs

```
open System
open Sql

let x = "
    SELECT x, y, z
    FROM t1
    LEFT JOIN t2
    INNER JOIN t3 ON t3.ID = t2.ID
    WHERE x = 50 AND y = 20
    ORDER BY x ASC, y DESC, z
"

let lexbuf = Lexing.from_string x
let y = SqlParser.start SqlLexer.tokenize lexbuf
printfn "%A" y

Console.WriteLine("(press any key)")
Console.ReadKey(true) |> ignore
```

Altogether, our minimal SQL lexer/parser is about 150 lines of code (including non-trivial lines of code and whitespace). I'll leave it as an exercise for the reader to implement the remainder of the SQL language spec.

2011-03-06: I tried the above instructions with VS2010 and F# 2.0 and PowerPack 2.0. I had to make a few changes:

- Add "module SqlLexer" on the 2nd line of SqlLexer.fsl
- Change Map.of_list to Map.ofList
- Add " --module SqlParser" to the command line of fsyacc
- Add FSharp.PowerPack to get Lexing module

2011-07-06: (Sjuul Janssen) These where the steps I had to take in order to make this work.

If you get the message "Expecting a LexBuffer<char> but given a LexBuffer<byte> The type 'char' does not match the type 'byte'"

- Add "fslex.exe "$(ProjectDir)SqlLexer.fsl" --unicode" to the pre-build
- in program.fs change "let lexbuf = Lexing.from_string x" to "let lexbuf = Lexing.LexBuffer<_>.FromString x"
- in SqlLexer.fsi change "lexeme lexbuf" to "LexBuffer<_>.LexemeString lexbuf"

If you get the message that some module doesn't exist or that some module is declared multiple times. Make sure that in the solution explorer the files come in this order:

- Sql.fs
- SqlParser.fsp
- SqlLexer.fsl
- SqlParser.fsi
- SqlParser.fs
- SqlLexer.fs
- Program.fs

If you get the message "Method not found: 'System.Object Microsoft.FSharp.Text.Parsing.Tables`1.Interpret(Microsoft.FSharp.Core.FSharpFunc`2<Microsoft.FSharp.Text.Lexing.LexBuffer`1<Byte>,!0>, ..." Go to http://www.microsoft.com/download/en/details.aspx?id=15834 and reinstall Visual Studio 2010 F# 2.0 Runtime SP1 (choose for **repair**)

2011-07-06: (mkduffi) Could someone please provide a sample project. I have followed all of your changes but still can not build. Thanks.

Sample

https://github.com/obeleh/FsYacc-Example

2011-07-07 (mkduffi) Thanks for posting the sample. Here is what I did to the Program.fs file:

```
namespace FS
module Parser =

open System
open Sql

let x = "
    SELECT x, y, z
    FROM t1
    LEFT JOIN t2
    INNER JOIN t3 ON t3.ID = t2.ID
    WHERE x = 50 AND y = 20
    ORDER BY x ASC, y DESC, z
"
let ParseSql x =
    let lexbuf = Lexing.LexBuffer<_>.FromString x
    let y = SqlParser.start SqlLexer.tokenize lexbuf
    y

let y = (ParseSql x)
printfn "%A" y

Console.WriteLine("(press any key)")
Console.ReadKey(true) |> ignore
```

I added a C# console project for testing and this is what is in the Program.cs file:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string query =
                @"SELECT x, y, z
                FROM t1
                LEFT JOIN t2
                INNER JOIN t3 ON t3.ID = t2.ID
                WHERE x = 50 AND y = 20
                ORDER BY x ASC, y DESC, z";

            Sql.sqlStatement stmnt = FS.Parser.ParseSql(query);
        }
    };
}
```

I had to add the YaccSample project reference as well as a reference to the FSharp.Core assembly to get this to work.

If anyone could help me figure out how to support table aliases that would be awesome.

Thanks

2011-07-08 (Sjuul Janssen) Contact me through my github account. I'm working on this and some other stuff.

Notes for Contributors

Would-be contributors are asked to observe the following:

- We aim to present functional programming in F# in a simple manner. Many texts go into great detail about the theoretical aspects of FP (*e.g.* lambda calculus) fairly early on which is :
  - distracting to beginners who assume that functional programming is for "boffins" only.
  - unnecessary for learning how to use functional programming techniques.
- Ensure that you only use a language construct or concept after it has been explained. It is the task of any tutorial to split things up into digestable chunks. This may mean explaining or demonstrating things in a clumsier or less general way than you would like. So be it.
- Do not treat the existing text or structure as sacrosanct. If it would be better done another way then by all means do it. Remember that WikiBooks are under version control, so any mistake can always be backed out. (Having said that, major changes should probably be discussed first).

Additionally topics should be organized in a way that makes sense to first-time programmers and those who have no prior experience with functional programming languages. In particular:

- Readers should be able to start at the beginning of this book and understand each section. No page should reference code or language constructs before they are properly introduced.
- Each section should build on prior sections. In other words, this book is meant to be read in serial format: from start to finish. Readers are not expected to understand code or concepts by skipping over material.
- An emphasis should be placed on explaining writing F# code properly, rather than on syntax alone. This is extremely important, but unfortunately overlooked in many programming tutorials on the web. There's no real point learning F# if programmers only intend to transcribe C++ in a different syntax; mastering this language means mastering its idioms and functional programming style.
- Its good for articles to have a high density of short, explanatory code samples. Always accompany new concepts with code that that reader can experiment with.

The text of this page is adapted from the page for Haskell.

Acknowledgments

The initial inspiration for this WikiBook came from reading the Haskell book.

Exercise Solutions

This section holds the solutions to the exercises in this book.

The solutions are on separate pages to ensure that you don't inadvertently see the solution to the next problem - which would spoil all the fun!

Basic F#

License