FREE eBook

LEARNING F#

Free unaffiliated eBook created from
Stack Overflow contributors.

#f#

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fsharp

It is an unofficial and free F# ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official F#.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with F#

Remarks

F# is a "functional-first" language. You can learn about all of the different types of expressions, along with functions.

The F# compiler -- which is open source -- compiles your programs into IL, which means that you can use F# code from any .NET compatible language such as C#; and run it on Mono, .NET Core, or the .NET Framework on Windows.

Versions

Version	Release Date
1.x	2005-05-01
2.0	2010-04-01
3.0	2012-08-01
3.1	2013-10-01
4.0	2015-07-01

Examples

Installation or Setup

Windows

If you have Visual Studio (any version including express and community) installed, F# should already be included. Just choose F# as the language when you create a new project. Or see http://fsharp.org/use/windows/ for more options.

OS X

Xamarin Studio supports F#. Alternately, you could use VS Code for OS X, which is a cross-platform editor by Microsoft.
Once done with installing VS Code, launch `VS Code Quick Open` (`Ctrl`+`P`) then run `ext install Ionide-fsharp`

You may also consider Visual Studio for Mac.

There are other alternatives described here.

Linux

Install the `mono-complete` and `fsharp` packages via your distribution's package manager (Apt, Yum, etc.). For a good editing experience, use either Visual Studio Code and install the `ionide-fsharp` plugin, or use Atom and install the `ionide-installer` plugin. See http://fsharp.org/use/linux/ for more options.

Hello, World!

This is the code for a simple console project, that prints "Hello, World!" to STDOUT, and exits with an exit code of `0`

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

Example breakdown Line-by-line:

- `[<EntryPoint>]` - A .net Attribute that marks "the method that you use to set the entry point" of your program (source).
- `let main argv` - this defines a function called `main` with a single parameter `argv`. Because this is the program entry point, `argv` will be an array of strings. The contents of the array are the arguments that were passed to the program when it was executed.
- `printfn "Hello, World!"` - the `printfn` function outputs the string** passed as its first argument, also appending a newline.
- `0` - F# functions always return a value, and the value returned is the result of the last expression in the function. Putting `0` as the last line means that the function will always return zero (an integer).

** This is actually *not* a string even though it looks like one. It's actually a TextWriterFormat, which optionally allows the usage of statically type checked arguments. But for the purpose of a "hello world" example it can be thought of as being a string.

F# Interactive

F# Interactive, is a REPL environment that lets you execute F# code, one line at a time.

If you have installed Visual Studio with F#, you can run F# Interactive in console by typing `"C:\Program Files (x86)\Microsoft SDKs\F#\4.0\Framework\v4.0\Fsi.exe"`. On Linux or OS X, the command is `fsharpi` instead, which should be either in `/usr/bin` or in `/usr/local/bin` depending on how you installed F# -- either way, the command should be on your `PATH` so you can just type `fsharpi`.

Example of F# interactive usage:

```
> let i = 1 // fsi prompt, declare i
- let j = 2 // declare j
- i+j // compose expression
- ;; // execute commands

val i : int = 1 // fsi output started, this gives the value of i
val j : int = 2 // the value of j
val it : int = 3 // computed expression

> #quit;; //quit fsi
```

Use `#help;;` for help

Please note the use of `;;` to tell the REPL to execute any previously-typed commands.

Read Getting started with F# online: https://riptutorial.com/fsharp/topic/817/getting-started-with-fsharp

Chapter 2: 1 : F# WPF Code Behind Application with FsXaml

Introduction

Most examples found for F# WPF programming seem to deal with the MVVM pattern, and a few with MVC, but there is next to none that shows properly how to get up and running with "good old" code behind.

The code behind pattern is very easy to use for teaching as well as experimentation. It is used in numerous introduction books and learning material on the web. That's why.

These examples will demonstrate how to create a code behind application with windows, controls, images and icons, and more.

Examples

Create a new F# WPF Code Behind Application.

Create an F# console application.

Change the **Output type** of the application to *Windows Application*.

Add the **FsXaml** NuGet package.

Add these four source files, in the order listed here.

MainWindow.xaml

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="First Demo" Height="200" Width="300">
    <Canvas>
        <Button Name="btnTest" Content="Test" Canvas.Left="10" Canvas.Top="10" Height="28"
Width="72"/>
    </Canvas>
</Window>
```

MainWindow.xaml.fs

```
namespace FirstDemo

type MainWindowXaml = FsXaml.XAML<"MainWindow.xaml">

type MainWindow() as this =
    inherit MainWindowXaml()
```

```
    let whenLoaded _ =
        ()

    let whenClosing _ =
        ()

    let whenClosed _ =
        ()

    let btnTestClick _ =
        this.Title <- "Yup, it works!"

    do
        this.Loaded.Add whenLoaded
        this.Closing.Add whenClosing
        this.Closed.Add whenClosed
        this.btnTest.Click.Add btnTestClick
```

App.xaml

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

App.xaml.fs

```
namespace FirstDemo

open System

type App = FsXaml.XAML<"App.xaml">

module Main =

    [<STAThread; EntryPoint>]
    let main _ =
        let app = App()
        let mainWindow = new MainWindow()
        app.Run(mainWindow) // Returns application's exit code.
```

Delete the *Program.fs* file from the project.

Change the **Build Action** to *Resource* for the two xaml files.

Add a reference to the .NET assembly **UIAutomationTypes**.

Compile and run.

You can't use the designer to add event handlers, but that's not a problem at all. Simply add them manually in the code behind, like you see with the three handlers in this example, including the handler for the test button.

UPDATE: An alternative and probably more elegant way to add event handlers has been added to

FsXaml. You can add the event handler in XAML, same as in C# but you have to do it manually, and then override the corresponding member that turns up in your F# type. I recommend this.

3 : Add an icon to a window

It's a good idea to keep all icons and images in one or more folders.

Right click on the project, and use F# Power Tools / New Folder to create a folder named Images.

On disk, place your icon in the new *Images* folder.

Back in Visual Studio, right click on *Images*, and use **Add / Existing Item**, then show *All Files (.)** to see the icon file so that you can select it, and then **Add** it.

Select the icon file, and set its **Build Action** to *Resource*.

In MainWindow.xaml, use the Icon attribute like this. Surrounding lines are shown for context.

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="First Demo" Height="200" Width="300"
    Icon="Images/MainWindow.ico">
<Canvas>
```

Before you run, do a Rebuild, and not just a Build. This is because Visual Studio doesn't always put the icon file into the executable unless you rebuild.

It is the window, and not the application, that now has an icon. You will see the icon in the top left of the window at runtime, and you will see it in the task bar. The Task Manager and Windows File Explorer will not show this icon, because they display the application icon rather than the window icon.

4 : Add icon to application

Create a text file named AppIcon.rc, with the following content.

```
1 ICON "AppIcon.ico"
```

You will need an icon file named AppIcon.ico for this to work, but of course you can adjust the names to your liking.

Run the following command.

```
"C:\Program Files (x86)\Windows Kits\10\bin\x64\rc.exe" /v AppIcon.rc
```

If you can't find rc.exe in this location, then search for it below **C:\Program Files (x86)\Windows Kits**. If you still can't find it, then download Windows SDK from Microsoft.

A file named AppIcon.res will be generated.

In Visual Studio, open the project properties. Select the **Application** page.

In the text box titled **Resource File**, type *AppIcon.res* (or *Images\AppIcon.res* if you put it there), and then close the project properties to save.

An error message will appear, stating "The resource file entered does not exist. Ignore this. The error message will not reappear.

Rebuild. The executable will then have an application icon, and this shows in File Explorer. When running, this icon will also appear in Task Manager.

2 : Add a control

Add these two files in this order above the files for the main window.

MyControl.xaml

```
<UserControl
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d" Height="50" Width="150">
    <Canvas Background="LightGreen">
        <Button Name="btnMyTest" Content="My Test" Canvas.Left="10" Canvas.Top="10"
Height="28" Width="106"/>
    </Canvas>
</UserControl>
```

MyControl.xaml.fs

```
namespace FirstDemo

type MyControlXaml = FsXaml.XAML<"MyControl.xaml">

type MyControl() =
    inherit MyControlXaml()
```

The **Build Action** for *MyControl.xaml* must be set to *Resource*.

You will of course later need to add "as this" in the declaration of MyControl, just as done for the main window.

In file **MainWindow.xaml.fs**, in the class for MainWindow, add this line

```
let myControl = MyControl()
```

and add these two lines in the **do**-section of the main window class.

```
    this.mainCanvas.Children.Add myControl |> ignore
    myControl.btnMyTest.Content <- "We're in business!"
```

There can be more than one **do**-section in a class, and you are likely to need it when writing lots of code-behind code.

The control has been given a light green background color, so that you can easily see where it is.

Be aware that the control will block the button of the main window from view. It is beyond the scope of these examples to teach you WPF in general, so we won't fix that here.

How to add controls from third party libraries

If you add controls from third party libraries in a C# WPF project, the XAML file will normally have lines like this one.

```
xmlns:xctk="http://schemas.xceed.com/wpf/xaml/toolkit"
```

This will perhaps not work with FsXaml.

The designer and the compiler accepts that line, but there will probably be an exception at runtime complaining about the 3rd party type not being found when reading the XAML.

Try something like the following instead.

```
xmlns:xctk="clr-namespace:Xceed.Wpf.Toolkit;assembly=Xceed.Wpf.Toolkit"
```

This then is an example of a control that depends on the above.

```
<xctk:IntegerUpDown Name="tbInput" Increment="1" Maximum="10" Minimum="0" Canvas.Left="13"
Canvas.Top="27" Width="270"/>
```

The library used in this example is the Extended Wpf Toolkit, available free of charge through NuGet or as installer. If you download libraries through NuGet, then the controls are not available in the Toolbox, but they still show in the designer if you add them manually in XAML, and the properties are available in the Properties pane.

Read 1 : F# WPF Code Behind Application with FsXaml online:
https://riptutorial.com/fsharp/topic/9008/1---fsharp-wpf-code-behind-application-with-fsxaml

Chapter 3: Active Patterns

Examples

Simple Active Patterns

Active patterns are a special type of pattern matching where you can specify named categories that your data may fall into, and then use those categories in `match` statements.

To define an active pattern that classifies numbers as positive, negative or zero:

```
let (|Positive|Negative|Zero|) num =
    if num > 0 then Positive
    elif num < 0 then Negative
    else Zero
```

This can then be used in a pattern matching expression:

```
let Sign value =
    match value with
    | Positive -> printf "%d is positive" value
    | Negative -> printf "%d is negative" value
    | Zero -> printf "The value is zero"

Sign -19 // -19 is negative
Sign 2 // 2 is positive
Sign 0 // The value is zero
```

Active Patterns with parameters

Active patterns are just simple functions.

Like functions you can define additional parameters:

```
let (|HasExtension|_|) expected (uri : string) =
    let result = uri.EndsWith (expected, StringComparison.CurrentCultureIgnoreCase)
    match result with
    | true -> Some true
    | _ -> None
```

This can be used in a pattern matching this way:

```
    let isXMLFile uri =
        match uri with
        | HasExtension ".xml" _ -> true
        | _ -> false
```

Active Patterns can be used to validate and transform function arguments

An interesting but rather unknown usage of Active Patterns in `F#` is that they can be used to validate and transform function arguments.

Consider the classic way to do argument validation:

```
// val f : string option -> string option -> string
let f v u =
  let v = defaultArg v "Hello"
  let u = defaultArg u "There"
  v + " " + u

// val g : 'T -> 'T (requires 'T null)
let g v =
  match v with
  | null  -> raise (System.NullReferenceException ())
  | _     -> v.ToString ()
```

Typically we add code in the method to verify that arguments are correct. Using Active Patterns in `F#` we can generalize this and declare the intent in the argument declaration.

The following code is equivalent to the code above:

```
let inline (|DefaultArg|) dv ov = defaultArg ov dv

let inline (|NotNull|) v =
  match v with
  | null  -> raise (System.NullReferenceException ())
  | _     -> v

// val f : string option -> string option -> string
let f (DefaultArg "Hello" v) (DefaultArg "There" u) = v + " " + u

// val g : 'T -> string (requires 'T null)
let g (NotNull v) = v.ToString ()
```

For the user of function `f` and `g` there's no difference between the two different versions.

```
printfn "%A" <| f (Some "Test") None  // Prints "Test There"
printfn "%A" <| g "Test"              // Prints "Test"
printfn "%A" <| g null                // Will throw
```

A concern is if Active Patterns adds performance overhead. Let's use `ILSpy` to decompile `f` and `g` to see if that is the case.

```
public static string f(FSharpOption<string> _arg2, FSharpOption<string> _arg1)
{
  return Operators.DefaultArg<string>(_arg2, "Hello") + " " +
Operators.DefaultArg<string>(_arg1, "There");
}

public static string g<a>(a _arg1) where a : class
{
  if (_arg1 != null)
  {
    a a = _arg1;
```

```
    return a.ToString();
  }
  throw new NullReferenceException();
}
```

Thanks to `inline` the Active Patterns adds no extra overhead compared to the classic way of the doing argument validation.

Active Patterns as .NET API wrappers

Active Patterns can be used to make calling some .NET API's feel more natural, particularly those that use an output parameter to return more than just the function return value.

For example, you'd normally call the `System.Int32.TryParse` method as follows:

```
let couldParse, parsedInt = System.Int32.TryParse("1")
if couldParse then printfn "Successfully parsed int: %i" parsedInt
else printfn "Could not parse int"
```

You can improve this a bit using pattern matching:

```
match System.Int32.TryParse("1") with
| (true, parsedInt) -> printfn "Successfully parsed int: %i" parsedInt
| (false, _) -> printfn "Could not parse int"
```

However, we can also define the following Active Pattern that wraps the `System.Int32.TryParse` function:

```
let (|Int|_|) str =
    match System.Int32.TryParse(str) with
    | (true, parsedInt) -> Some parsedInt
    | _ -> None
```

Now we can do the following:

```
match "1" with
| Int parsedInt -> printfn "Successfully parsed int: %i" parsedInt
| _ -> printfn "Could not parse int"
```

Another good candidate for being wrapped in an Active Patterns are the regular expressions API's:

```
let (|MatchRegex|_|) pattern input =
    let m = System.Text.RegularExpressions.Regex.Match(input, pattern)
    if m.Success then Some m.Groups.[1].Value
    else None

match "bad" with
| MatchRegex "(good|great)" mood ->
    printfn "Wow, it's a %s day!" mood
| MatchRegex "(bad|terrible)" mood ->
    printfn "Unfortunately, it's a %s day." mood
```

```
| _ ->
    printfn "Just a normal day"
```

Complete and Partial Active Patterns

There are two types of Active Patterns that somewhat differ in usage - Complete and Partial.

Complete Active Patterns can be used when you are able to enumerate all the outcomes, like "is a number odd or even?"

```
let (|Odd|Even|) number =
  if number % 2 = 0
  then Even
  else Odd
```

Notice the Active Pattern definition lists both possible cases and nothing else, and the body returns one of the listed cases. When you use it in a match expression, this is a complete match:

```
let n = 13
match n with
| Odd -> printf "%i is odd" n
| Even -> printf "%i is even" n
```

This is handy when you want to break down the input space into known categories that cover it completely.

Partial Active Patterns on the other hand let you explicitly ignore some possible results by returning an `option`. Their definition uses a special case of _ for the unmatched case.

```
let (|Integer|_|) str =
  match Int32.TryParse(str) with
  | (true, i) -> Some i
  | _ -> None
```

This way we can match even when some cases cannot be handled by our parsing function.

```
let s = "13"
match s with
| Integer i -> "%i was successfully parsed!" i
| _ -> "%s is not an int" s
```

Partial Active Patterns can be used as a form of testing, whether the input falls into a specific category in the input space while ignoring other options.

Read Active Patterns online: https://riptutorial.com/fsharp/topic/962/active-patterns

Chapter 4: Classes

Examples

Declaring a class

```
// class declaration with a list of parameters for a primary constructor
type Car (model: string, plates: string, miles: int) =

    // secondary constructor (it must call primary constructor)
    new (model, plates) =
        let miles = 0
        new Car(model, plates, miles)

    // fields
    member this.model = model
    member this.plates = plates
    member this.miles = miles
```

Creating an instance

```
let myCar = Car ("Honda Civic", "blue", 23)
// or more explicitly
let (myCar : Car) = new Car ("Honda Civic", "blue", 23)
```

Read Classes online: https://riptutorial.com/fsharp/topic/3003/classes

Chapter 5: Design pattern implementation in F#

Examples

Data-driven programming in F#

Thanks to type-inference and partial application in F# data-driven programming is succinct and readable.

Let's imagine we are selling car insurance. Before we try selling it to a customer, we try to determine if the customer is a valid potential customer for our company by checking the customer's sex and age.

A simple customer model:

```
type Sex =
  | Male
  | Female

type Customer =
  {
    Name    : string
    Born    : System.DateTime
    Sex     : Sex
  }
```

Next we want to define an exclusion list (table) so that if a customer matches any row in the exclusion list the customer is cannot buy our car insurance.

```
//  If any row in this list matches the Customer, the customer isn't eligible for the car
insurance.
let exclusionList =
  let __          _   = true
  let olderThan   x y = x < y
  let youngerThan x y = x > y
  [|
//  Description                         Age               Sex
    "Not allowed for senior citizens" , olderThan   65  , __
    "Not allowed for children"        , youngerThan 16  , __
    "Not allowed for young males"     , youngerThan 25  , (=) Male
  |]
```

Because of type-inference and partial application, the exclusion list is flexible yet easy to understand.

Finally, we define a function that uses the exclusion list (a table) to split the customers into two buckets: potential and denied customers.

```
// Splits customers into two buckets: potential customers and denied customers.
// The denied customer bucket also includes the reason for denying them the car insurance
let splitCustomers (today : System.DateTime) (cs : Customer []) : Customer
[]*(string*Customer) [] =
  let potential = ResizeArray<_> 16 // ResizeArray is an alias for
System.Collections.Generic.List
  let denied    = ResizeArray<_> 16

  for c in cs do
    let age = today.Year - c.Born.Year
    let sex = c.Sex
    match exclusionList |> Array.tryFind (fun (_, testAge, testSex) -> testAge age && testSex
sex) with
    | Some (description, _, _)  -> denied.Add (description, c)
    | None                      -> potential.Add c

  potential.ToArray (), denied.ToArray ()
```

To wrap up, let's define some customers and see if they are any potential customers for our car insurance amongst them:

```
let customers =
  let c n s y m d: Customer = { Name = n; Born = System.DateTime (y, m, d); Sex = s }
  [|
//    Name                      Sex     Born
    c "Clint Eastwood Jr."      Male    1930 05 31
    c "Bill Gates"              Male    1955 10 28
    c "Melina Gates"            Female  1964 08 15
    c "Justin Drew Bieber"      Male    1994 03 01
    c "Sophie Turner"           Female  1996 02 21
    c "Isaac Hempstead Wright"  Male    1999 04 09
  |]

[<EntryPoint>]
let main argv =
  let potential, denied = splitCustomers (System.DateTime (2014, 06, 01)) customers
  printfn "Potential Customers (%d)\n%A" potential.Length potential
  printfn "Denied Customers (%d)\n%A"    denied.Length    denied
  0
```

This prints:

```
Potential Customers (3)
[|{Name = "Bill Gates";
   Born = 1955-10-28 00:00:00;
   Sex = Male;}; {Name = "Melina Gates";
                  Born = 1964-08-15 00:00:00;
                  Sex = Female;}; {Name = "Sophie Turner";
                                   Born = 1996-02-21 00:00:00;
                                   Sex = Female;}|]
Denied Customers (3)
[|("Not allowed for senior citizens", {Name = "Clint Eastwood Jr.";
                                       Born = 1930-05-31 00:00:00;
                                       Sex = Male;});
  ("Not allowed for young males", {Name = "Justin Drew Bieber";
                                   Born = 1994-03-01 00:00:00;
                                   Sex = Male;});
  ("Not allowed for children", {Name = "Isaac Hempstead Wright";
```

```
                                Born = 1999-04-09 00:00:00;
                                Sex = Male;})|]
```

Read Design pattern implementation in F# online: https://riptutorial.com/fsharp/topic/3925/design-pattern-implementation-in-fsharp

Chapter 6: Discriminated Unions

Examples

Naming elements of tuples within discriminated unions

When defining discriminated unions you can name elements of tuple types and use these names during pattern matching.

```
type Shape =
    | Circle of diameter:int
    | Rectangle of width:int * height:int

let shapeIsTenWide = function
    | Circle(diameter=10)
    | Rectangle(width=10) -> true
    | _ -> false
```

Additionally naming the elements of discriminated unions improves readability of the code and interoperability with C# - provided names will be used for properties' names and constructors' parameters. Default generated names in interop code are "Item", "Item1", "Item2"...

Basic Discriminated Union Usage

Discriminated unions in F# offer a a way to define types which may hold any number of different data types. Their functionality is similar to C++ unions or VB variants, but with the additional benefit of being type safe.

```
// define a discriminated union that can hold either a float or a string
type numOrString =
    | F of float
    | S of string

let str = S "hi" // use the S constructor to create a string
let fl = F 3.5 // use the F constructor to create a float

// you can use pattern matching to deconstruct each type
let whatType x =
    match x with
        | F f -> printfn "%f is a float" f
        | S s -> printfn "%s is a string" s

whatType str // hi is a string
whatType fl // 3.500000 is a float
```

Enum-style unions

Type information does not need to be included in the cases of a discriminated union. By omitting type information you can create a union that simply represents a set of choices, similar to an enum.

```
// This union can represent any one day of the week but none of
// them are tied to a specific underlying F# type
type DayOfWeek = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Converting to and from strings with Reflection

Sometimes it's necessary to convert a Discriminated Union to and from a string:

```
module UnionConversion
    open Microsoft.FSharp.Reflection

    let toString (x: 'a) =
        match FSharpValue.GetUnionFields(x, typeof<'a>) with
        | case, _ -> case.Name

    let fromString<'a> (s : string) =
        match FSharpType.GetUnionCases typeof<'a> |> Array.filter (fun case -> case.Name = s)
with
        | [|case|] -> Some(FSharpValue.MakeUnion(case, [||])) :?> 'a)
        | _ -> None
```

Single case discriminated union

A single case discriminated union is like any other discriminated union except that it only has one case.

```
// Define single-case discriminated union type.
type OrderId = OrderId of int
// Construct OrderId type.
let order = OrderId 123
// Deconstruct using pattern matching.
// Parentheses used so compiler doesn't think it is a function definition.
let (OrderId id) = order
```

It is useful for enforcing type safety and commonly used in F# as opposed to C# and Java where creating new types comes with more overhead.

The following two alternative type definitions result in the same single-case discriminated union being declared:

```
type OrderId = | OrderId of int

type OrderId =
      | OrderId of int
```

Using Single-case Discriminated Unions as Records

Sometimes it is useful to create union types with only one case to implement record-like types:

```
type Point = Point of float * float

let point1 = Point(0.0, 3.0)
```

```
let point2 = Point(-2.5, -4.0)
```

These become very useful because they can be decomposed via pattern matching in the same way as tuple arguments can:

```
let (Point(x1, y1)) = point1
// val x1 : float = 0.0
// val y1 : float = 3.0

let distance (Point(x1,y1)) (Point(x2,y2)) =
    pown (x2-x1) 2 + pown (y2-y1) 2 |> sqrt
// val distance : Point -> Point -> float

distance point1 point2
// val it : float = 7.433034374
```

RequireQualifiedAccess

With the `RequireQualifiedAccess` attribute, union cases must be referred to as `MyUnion.MyCase` instead of just `MyCase`. This prevents name collisions in the enclosing namespace or module:

```
type [<RequireQualifiedAccess>] Requirements =
    None | Single | All

// Uses the DU with qualified access
let noRequirements = Requirements.None

// Here, None still refers to the standard F# option case
let getNothing () = None

// Compiler error unless All has been defined elsewhere
let invalid = All
```

If, for example, `System` has been opened, `Single` refers to `System.Single`. There is no collision with the union case `Requirements.Single`.

Recursive discriminated unions

Recursive type

Discriminated unions can be recursive, that is they can refer to themselves in their definition. The prime example here is a tree:

```
type Tree =
    | Branch of int * Tree list
    | Leaf of int
```

As an example, let's define the following tree:

```
    1
  2   5
3   4
```

We can define this tree using our recursive discriminated union as follows:

```
let leaf1 = Leaf 3
let leaf2 = Leaf 4
let leaf3 = Leaf 5

let branch1 = Branch (2, [leaf1; leaf2])
let tip = Branch (1, [branch1; leaf3])
```

Iterating over the tree is then just a matter of pattern matching:

```
let rec toList tree =
    match tree with
    | Leaf x -> [x]
    | Branch (x, xs) -> x :: (List.collect toList xs)

let treeAsList = toList tip // [1; 2; 3; 4; 5]
```

Mutually dependent recursive types

One way to achieve recursion is to have nested mutually dependent types.

```
// BAD
type Arithmetic = {left: Expression; op:string; right: Expression}
// illegal because until this point, Expression is undefined
type Expression =
| LiteralExpr of obj
| ArithmeticExpr of Arithmetic
```

Defining a record type directly inside a discriminated union is deprecated:

```
// BAD
type Expression =
| LiteralExpr of obj
| ArithmeticExpr of {left: Expression; op:string; right: Expression}
// illegal in recent F# versions
```

You can use the `and` keyword to chain mutually dependent definitions:

```
// GOOD
type Arithmetic = {left: Expression; op:string; right: Expression}
and Expression =
| LiteralExpr of obj
| ArithmeticExpr of Arithmetic
```

Read Discriminated Unions online: https://riptutorial.com/fsharp/topic/1025/discriminated-unions

Chapter 7: F# on .NET Core

Examples

Creating a new project via dotnet CLI

Once you've installed the .NET CLI tools, you can create a new project with the following command:

```
dotnet new --lang f#
```

This creates a command line program.

Initial project workflow

Create a new project

```
dotnet new -l f#
```

Restore any packages listed in project.json

```
dotnet restore
```

A project.lock.json file should be written out.

Execute the program

```
dotnet run
```

The above will compile the code if required.

The output of the default project created by `dotnet new -l f#` contains the following:

```
Hello World!
[|||]
```

Read F# on .NET Core online: https://riptutorial.com/fsharp/topic/4404/fsharp-on--net-core

Chapter 8: F# Performance Tips and Tricks

Examples

Using tail-recursion for efficient iteration

Coming from imperative languages many developers wonder how to write a `for-loop` that exits early as `F#` doesn't support `break`, `continue` or `return`. The answer in `F#` is to use tail-recursion which is a flexible and idiomatic way to iterate while still providing excellent performance.

Say we want to implement `tryFind` for `List`. If `F#` supported `return` we would write `tryFind` a bit like this:

```
let tryFind predicate vs =
  for v in vs do
    if predicate v then
      return Some v
  None
```

This doesn't work in `F#`. Instead we write the function using tail-recursion:

```
let tryFind predicate vs =
  let rec loop = function
    | v::vs -> if predicate v then
                   Some v
               else
                   loop vs
    | _ -> None
  loop vs
```

Tail-recursion is performant in `F#` because when the `F#` compiler detects that a function is tail-recursive it rewrites the recursion into an efficient `while-loop`. Using `ILSpy` we can see that this is true for our function `loop`:

```
internal static FSharpOption<a> loop@3-10<a>(FSharpFunc<a, bool> predicate, FSharpList<a>
_arg1)
{
  while (_arg1.TailOrNull != null)
  {
    FSharpList<a> fSharpList = _arg1;
    FSharpList<a> vs = fSharpList.TailOrNull;
    a v = fSharpList.HeadOrDefault;
    if (predicate.Invoke(v))
    {
      return FSharpOption<a>.Some(v);
    }
    FSharpFunc<a, bool> arg_2D_0 = predicate;
    _arg1 = vs;
    predicate = arg_2D_0;
  }
  return null;
}
```

Apart from some unnecessary assignments (which hopefully the JIT-er eliminates) this is essentially an efficient loop.

In addition, tail-recursion is idiomatic for `F#` as it allows us to avoid mutable state. Consider a `sum` function that sums all elements in a `List`. An obvious first try would be this:

```
let sum vs =
  let mutable s = LanguagePrimitives.GenericZero
  for v in vs do
    s <- s + v
  s
```

If we rewrite the loop into tail-recursion we can avoid the mutable state:

```
let sum vs =
  let rec loop s = function
    | v::vs -> loop (s + v) vs
    | _ -> s
  loop LanguagePrimitives.GenericZero vs
```

For efficiency the `F#` compiler transforms this into a `while-loop` that uses mutable state.

Measure and Verify your performance assumptions

This example is written with `F#` in mind but the ideas are applicable in all environments

The first rule when optimizing for performance is to not to rely assumption; always Measure and Verify your assumptions.

As we are not writing machine code directly it is hard to predict how the compiler and JIT:er transform your program to machine code. That's why we need to Measure the execution time to see that we get the performance improvement we expect and Verify that the actual program doesn't contain any hidden overhead.

Verification is the quite simple process that involves reverse engineering the executable using for example tools like ILSpy. The JIT:er complicates Verification in that seeing the actual machine code is tricky but doable. However, usually examining the `IL-code` gives the big gains.

The harder problem is Measuring; harder because it's tricky to setup realistic situations that allows to measure improvements in code. Still Measuring is invaluable.

Analyzing simple F# functions

Let's examine some simple `F#` functions that accumulates all integers in `1..n` written in various different ways. As the range is a simple Arithmetic Series the result can be computed directly but for the purpose of this example we iterate over the range.

First we define some useful functions for measuring the time a function takes:

```
// now () returns current time in milliseconds since start
```

```
let now : unit -> int64 =
  let sw = System.Diagnostics.Stopwatch ()
  sw.Start ()
  fun () -> sw.ElapsedMilliseconds

// time estimates the time 'action' repeated a number of times
let time repeat action : int64*'T =
  let v = action ()  // Warm-up and compute value

  let b = now ()
  for i = 1 to repeat do
    action () |> ignore
  let e = now ()

  e - b, v
```

`time` runs an action repeatedly we need to run the tests for a few hundred milliseconds to reduce variance.

Then we define a few functions that accumulates all integers in `1..n` in different ways.

```
// Accumulates all integers 1..n using 'List'
let accumulateUsingList n =
  List.init (n + 1) id
  |> List.sum

// Accumulates all integers 1..n using 'Seq'
let accumulateUsingSeq n =
  Seq.init (n + 1) id
  |> Seq.sum

// Accumulates all integers 1..n using 'for-expression'
let accumulateUsingFor n =
  let mutable sum = 0
  for i = 1 to n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over list range
let accumulateUsingForEachOverList n =
  let mutable sum = 0
  for i in [1..n] do
    sum <- sum + i
  sum

// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum
```

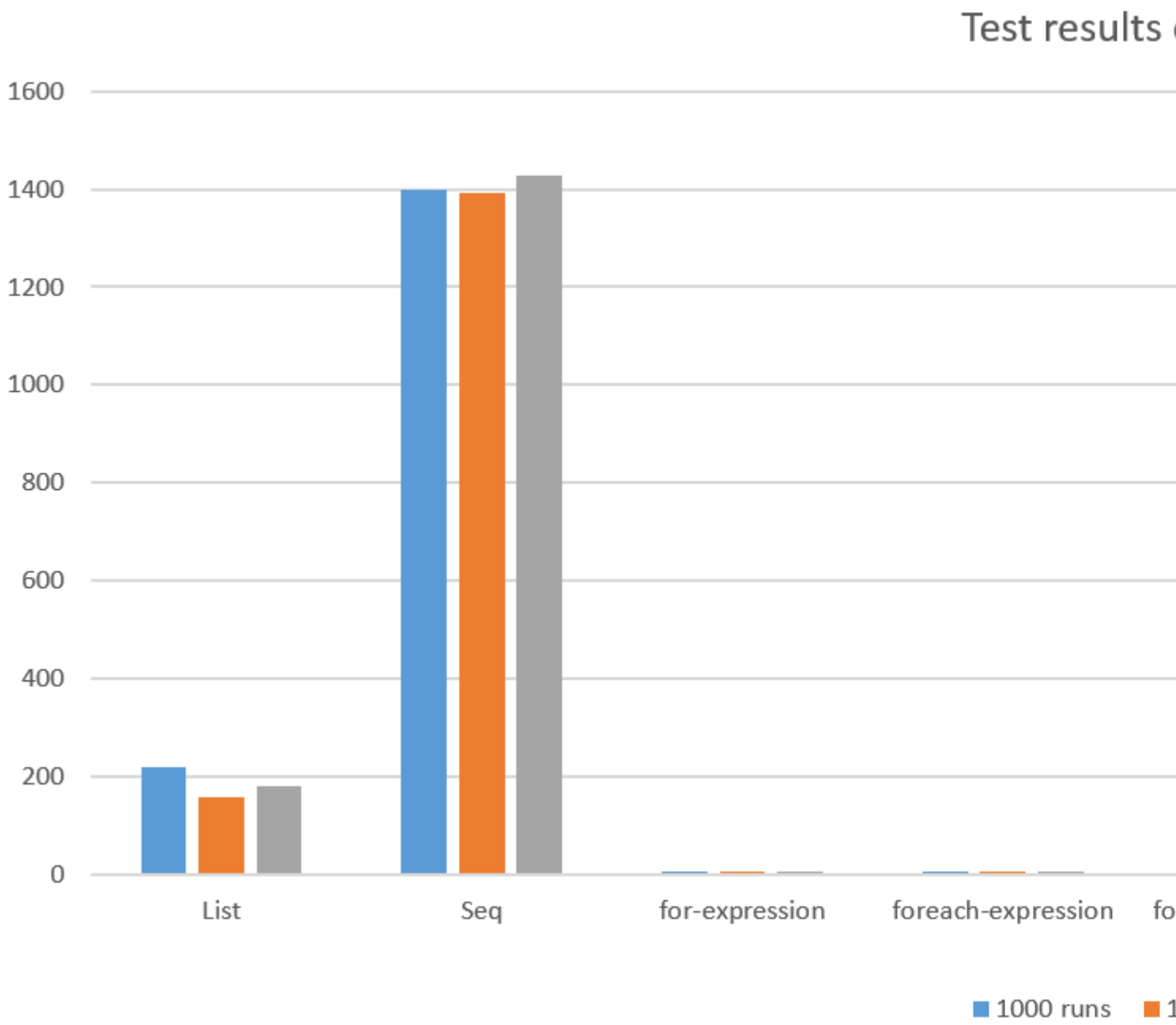
```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int

// Accumulates all integers n..1 using 'for-expression' in reverse order
let accumulateUsingReverseFor n =
  let mutable sum = 0
  for i = n downto 1 do
    sum <- sum + i
  sum

// Accumulates all 64 integers n..1 using 'tail-recursion' in reverse order
let accumulateUsingReverseTailRecursion n =
  let rec loop sum i =
    if i > 0 then
      loop (sum + i) (i - 1)
    else
      sum
  loop 0 n
```

We assume the result to be the same (except for one of the functions that uses increment of 2) but is there difference in performance. To Measure this the following function is defined:

```
let testRun (path : string) =
  use testResult = new System.IO.StreamWriter (path)
  let write   (l : string)  = testResult.WriteLine l
  let writef  fmt           = FSharp.Core.Printf.kprintf write fmt

  write "Name\tTotal\tOuter\tInner\tCC0\tCC1\tCC2\tTime\tResult"

  // total is the total number of iterations being executed
  let total   = 10000000
  // outers let us variate the relation between the inner and outer loop
  //  this is often useful when the algorithm allocates different amount of memory
  //  depending on the input size. This can affect cache locality
  let outers  = [| 1000; 10000; 100000 |]
  for outer in outers do
    let inner = total / outer

    // multiplier is used to increase resolution of certain tests that are significantly
    //  faster than the slower ones

    let testCases =
      [|
    //   Name of test                          multiplier    action
        "List"                               , 1           , accumulateUsingList
        "Seq"                                , 1           , accumulateUsingSeq
        "for-expression"                     , 100         , accumulateUsingFor
        "foreach-expression"                 , 100         , accumulateUsingForEach
        "foreach-expression over List"       , 1           , accumulateUsingForEachOverList
        "foreach-expression increment of 2" , 1           , accumulateUsingForEachStep2
        "foreach-expression over 64 bit"     , 1           , accumulateUsingForEach64
        "reverse for-expression"             , 100         , accumulateUsingReverseFor
        "reverse tail-recursion"             , 100         ,
accumulateUsingReverseTailRecursion
      |]
    for name, multiplier, a in testCases do
```

```
      System.GC.Collect (2, System.GCCollectionMode.Forced, true)
      let cc g = System.GC.CollectionCount g

      printfn "Accumulate using %s with outer=%d and inner=%d ..." name outer inner

      // Collect collection counters before test run
      let pcc0, pcc1, pcc2 = cc 0, cc 1, cc 2

      let ms, result       = time (outer*multiplier) (fun () -> a inner)
      let ms               = (float ms / float multiplier)

      // Collect collection counters after test run
      let acc0, acc1, acc2 = cc 0, cc 1, cc 2
      let cc0, cc1, cc2    = acc0 - pcc0, acc1 - pcc1, acc1 - pcc1
      printfn "  ... took: %f ms, GC collection count %d,%d,%d and produced %A" ms cc0 cc1 cc2
result

      writef "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%f\t%d" name total outer inner cc0 cc1 cc2 ms result
```

The test result while running on .NET 4.5.2 x64:



We see dramatic difference and some of the results are unexpectedly bad.

Let's look at the bad cases:

List

```
// Accumulates all integers 1..n using 'List'
let accumulateUsingList n =
  List.init (n + 1) id
  |> List.sum
```

What happens here is a full list containing all integers `1..n` is created and reduced using a sum. This should be more expensive than just iterating and accumulating over the range, it seems about ~42x slower than the for loop.

In addition, we can see that the GC ran about 100x during the test run because the code allocated

a lot of objects. This also costs CPU.

Seq

```
// Accumulates all integers 1..n using 'Seq'
let accumulateUsingSeq n =
  Seq.init (n + 1) id
  |> Seq.sum
```

The `Seq` version doesn't allocate a full `List` so it's a bit suprising that this ~270x slower than the for loop. In addition, we see that the GC has executed 661x.

`Seq` is inefficient when the amount of work per item is very small (in this case aggregating two integers).

The point is not to never use `Seq`. The point is to Measure.

(**manofstick edit:** `Seq.init` is the culprit of this severe performance issue. It is much more efficent to use the expression `{ 0 .. n }` instead of `Seq.init (n+1) id`. This will become much more efficient still when this PR is merged and released. Even after the release though, the original `Seq.init ... |> Seq.sum` will still be slow, but somewhat counter-intuitively, `Seq.init ... |> Seq.map id |> Seq.sum` will be quite fast. This was to maintain backward compatibility with `Seq.init`s implementation, which doesn't calculate `Current` initially, but rather wraps them in a `Lazy` object - although this too should perform a little better due to this PR. Note to editor: sorry this is kind of rambling notes, but I don't want people to be put off Seq when improvement is just around the corner... *When that times does come it would be good to update the charts that are on this page.*)

foreach-expression over List

```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates all integers 1..n using 'foreach-expression' over list range
let accumulateUsingForEachOverList n =
  let mutable sum = 0
  for i in [1..n] do
    sum <- sum + i
  sum
```

The difference between these two function is very subtle but the performance difference is not, roughly ~76x. Why? Let's reverse engineer the bad code:

```
public static int accumulateUsingForEach(int n)
{
  int sum = 0;
  int i = 1;
  if (n >= i)
  {
    do
```

```
    {
      sum += i;
      i++;
    }
    while (i != n + 1);
  }
  return sum;
}

public static int accumulateUsingForEachOverList(int n)
{
  int sum = 0;
  FSharpList<int> fSharpList =
SeqModule.ToList<int>(Operators.CreateSequence<int>(Operators.OperatorIntrinsics.RangeInt32(1,
1, n)));
  for (FSharpList<int> tailOrNull = fSharpList.TailOrNull; tailOrNull != null; tailOrNull =
fSharpList.TailOrNull)
  {
    int i = fSharpList.HeadOrDefault;
    sum += i;
    fSharpList = tailOrNull;
  }
  return sum;
}
```

`accumulateUsingForEach` is implemented as an efficient `while` loop but `for i in [1..n]` is converted into:

```
FSharpList<int> fSharpList =
  SeqModule.ToList<int>(
    Operators.CreateSequence<int>(
      Operators.OperatorIntrinsics.RangeInt32(1, 1, n)));
```

This means first we create a `Seq` over `1..n` and finally calls `ToList`.

Expensive.

foreach-expression increment of 2

```
// Accumulates all integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach n =
  let mutable sum = 0
  for i in 1..n do
    sum <- sum + i
  sum

// Accumulates every second integer 1..n using 'foreach-expression' over range
let accumulateUsingForEachStep2 n =
  let mutable sum = 0
  for i in 1..2..n do
    sum <- sum + i
  sum
```

Once again the difference between these two functions are subtle but the performance difference is brutal: ~25x

Once again let's run `ILSpy`:

```
public static int accumulateUsingForEachStep2(int n)
{
  int sum = 0;
  IEnumerable<int> enumerable = Operators.OperatorIntrinsics.RangeInt32(1, 2, n);
  foreach (int i in enumerable)
  {
    sum += i;
  }
  return sum;
}
```

A `Seq` is created over `1..2..n` and then we iterate over `Seq` using the enumerator.

We were expecting `F#` to create something like this:

```
public static int accumulateUsingForEachStep2(int n)
{
  int sum = 0;
  for (int i = 1; i < n; i += 2)
  {
    sum += i;
  }
  return sum;
}
```

However, `F#` compiler only supports efficient for loops over int32 ranges that increment by one. For all other cases it falls back on `Operators.OperatorIntrinsics.RangeInt32`. Which will explain the next suprising result

foreach-expression over 64 bit

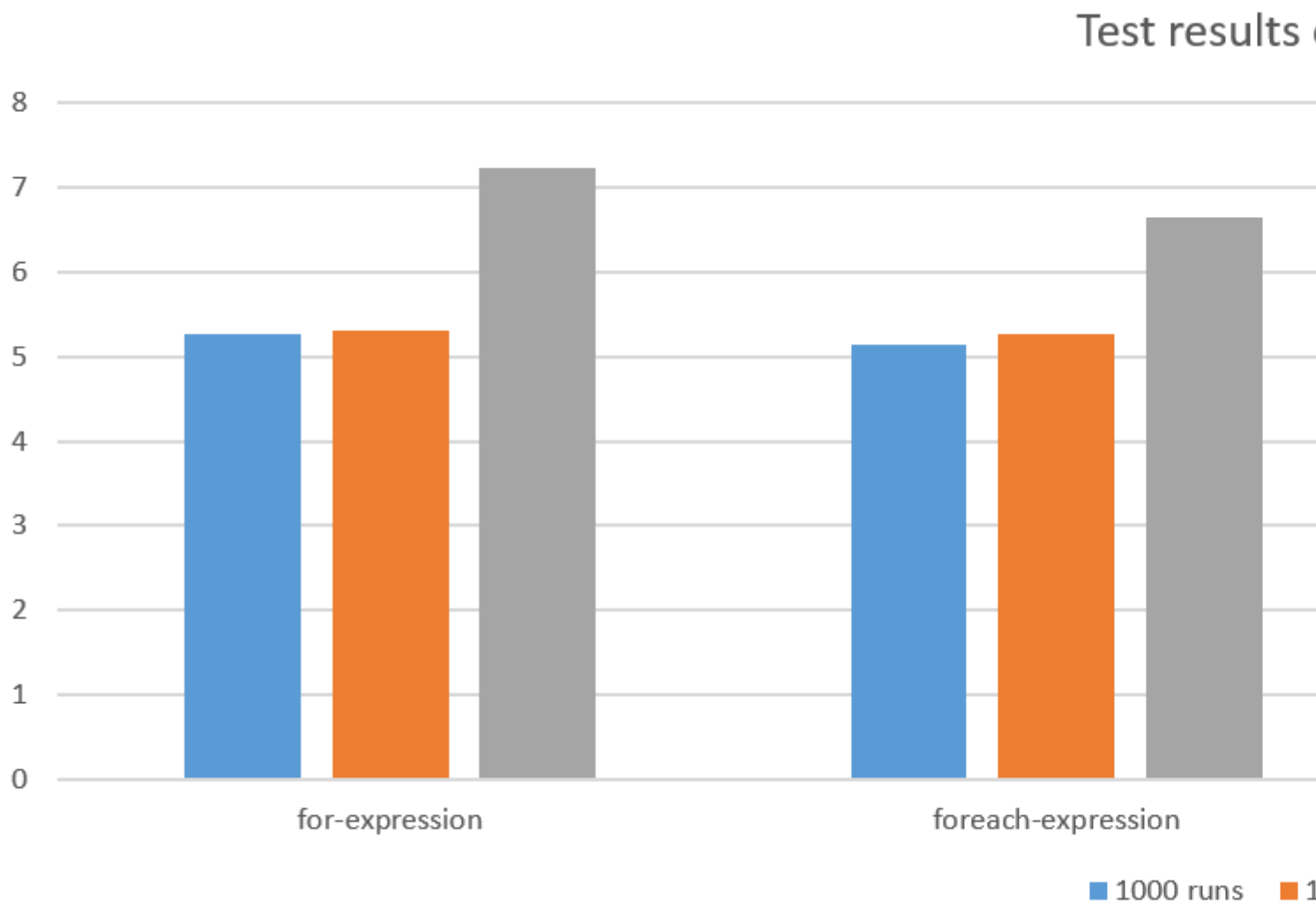
```
// Accumulates all 64 bit integers 1..n using 'foreach-expression' over range
let accumulateUsingForEach64 n =
  let mutable sum = 0L
  for i in 1L..int64 n do
    sum <- sum + i
  sum |> int
```

This performs ~47x slower than the for loop, the only difference is that we iterate over 64 bit integers. `ILSpy` shows us why:

```
public static int accumulateUsingForEach64(int n)
{
  long sum = 0L;
  IEnumerable<long> enumerable = Operators.OperatorIntrinsics.RangeInt64(1L, 1L, (long)n);
  foreach (long i in enumerable)
  {
    sum += i;
  }
  return (int)sum;
}
```

`F#` only supports efficient for loops for `int32` numbers it has to use the fallback `Operators.OperatorIntrinsics.RangeInt64`.

The other cases performs roughly similar:



The reason the performance degrades for larger test runs is that the overhead of invoking the `action` is growing as we doing less and less work in `action`.

Looping towards `0` can sometimes give performance benefits as it might save a CPU register but in this case the CPU has registers to spare so it doesn't seem to make a difference.

Conclusion

Measuring is important because otherwise we might think all these alternatives are equivalent but some alternatives are ~270x slower than others.

The Verification step involves reverse engineering the executable helps us explain *why* we did or did not get performance we expected. In addition, Verification can help us predict performance in the cases it's too difficult to do a proper Measurement.

It's hard to predict performance there always Measure, always Verify your performance assumptions.

Comparison of different F# data pipelines

In `F#` there are many options for creating data pipelines, for example: `List`, `Seq` and `Array`.

What data pipeline is preferable from memory usage and performance perspective?

In order to answer this we'll compare performance and memory usage using different pipelines.

Data Pipeline

In order to measure the overhead, we will use a data pipeline with low cpu-cost per items processed:

```
let seqTest n =
  Seq.init (n + 1) id
  |> Seq.map    int64
  |> Seq.filter (fun v -> v % 2L = 0L)
  |> Seq.map    ((+) 1L)
  |> Seq.sum
```

We will create equivalent pipelines for all alternatives and compare them.

We will variate the size of `n` but let the total number of work be the same.

Data Pipeline Alternatives

We will compare the following alternatives:

1. Imperative code
2. Array (Non-lazy)
3. List (Non-lazy)
4. LINQ (Lazy pull stream)
5. Seq (Lazy pull stream)
6. Nessos (Lazy pull/push stream)
7. PullStream (Simplistic pull stream)
8. PushStream (Simplistic push stream)

Although not a data pipeline we will compare against `Imperative` code since that most closely match how the CPU executes code. That should be that fastest possible way to compute the result allowing us to measure the performance overhead of data pipelines.

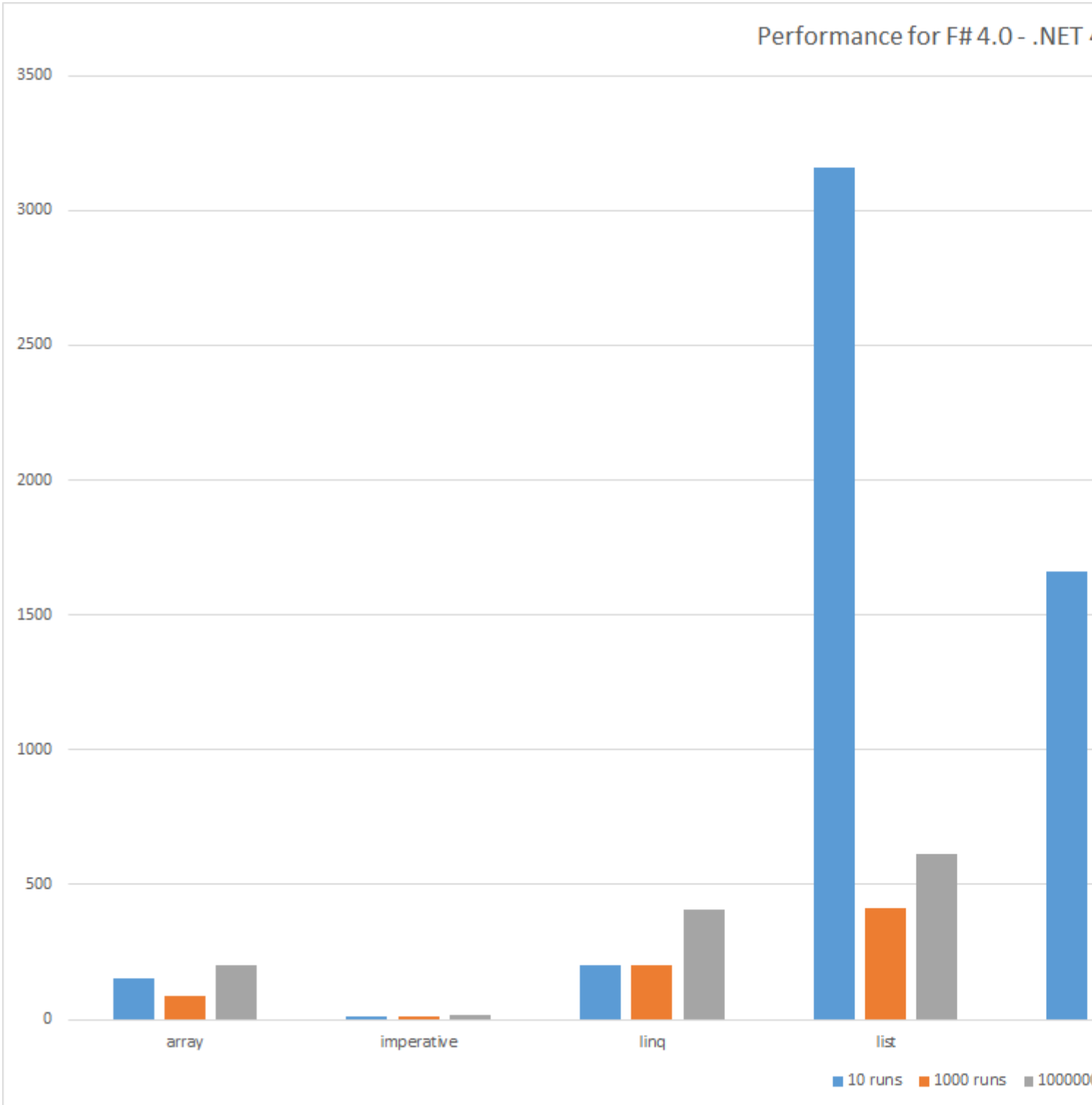
`Array` and `List` compute a full `Array`/`List` in each step so we expect memory overhead.

`LINQ` and `Seq` are both based around `IEnumerable<'T>` which is lazy pull stream (pull means that the consumer stream is pulling data out of the producer stream). We therefore expect the performance and memory usage to be identical.

`Nessos` is a high-performance stream library that supports both push & pull (like Java `Stream`).

PullStream and PushStream are simplistic implementations of `Pull` & `Push` streams.

Performance Results from running on: F# 4.0 - .NET 4.6.1 - x64



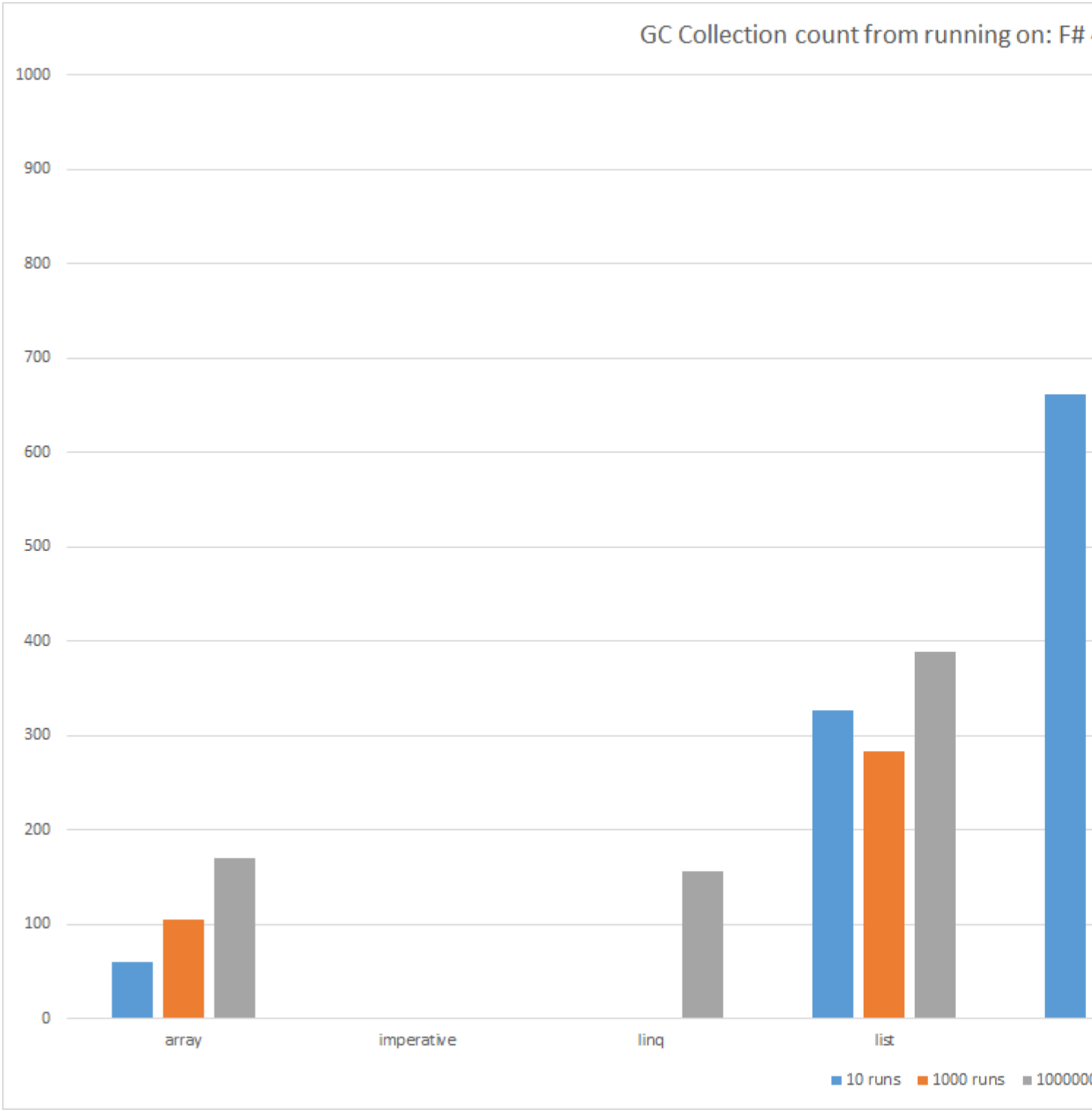
The bars show the elapsed time, lower is better. The total amount of useful work is the same for all tests so the results are comparable. This also means that few runs implies larger datasets.

As usual when Measuring one see interesting results.

1. `List` performance poor is compared to other alternatives for large data sets. This can be because of `GC` or poor cache locality.
2. `Array` performance better than expected.
3. `LINQ` performs better than `Seq`, this is unexpected because both are based around

`IEnumerable<'T>`. However, `Seq` internally is based around a generic impementation for all algorithms while `LINQ` uses specialized algorithms.
4. `Push` performs better than `Pull`. This is expected since the push data pipeline has fewer checks
5. The simplistic `Push` data pipelines performs comparable to `Nessos`. However, `Nessos` supports pull and parallelism.
6. For small data pipelines the performance of `Nessos` degrades possible because pipelines setup overhead.
7. As expected the `Imperative` code performed the best

GC Collection count from running on: F# 4.0 - .NET 4.6.1 - x64



The bars shows the total number of `GC` collection counts during the test, lower is better. This is a measurement of how many objects are created by the data pipeline.

As usual when Measuring one see interesting results.

1. `List` is expectedly creating more objects than `Array` because a `List` is essentially a single linked list of nodes. An array is a continous memory area.
2. Looking at the underlying numbers both `List` & `Array` forces 2 generation collections. These kind of collection are expensive.
3. `Seq` is triggering a surprising amount of collections. It's surprisingly even worse than `List` in

this regard.
4. `LINQ`, `Nessos`, `Push` and `Pull` triggers no collections for few runs. However, objects are allocated so the `GC` eventually will have to run.
5. As expected since the `Imperative` code allocate no objects no `GC` collections were triggered.

Conclusion

All data pipelines do the same amount of useful work in all test cases but we see significant differences in performance and memory usage between the different pipelines.

In addition, we notice that the overhead of data pipelines differ depending on the size of data processed. For example, for small sizes `Array` is performing quite well.

One should keep in mind the amount of work performed in each step in the pipeline is very small in order to measure the overhead. In "real" situations the overhead of `Seq` might not matter because the actual work is more time consuming.

Of more concern is the memory usage differences. `GC` isn't free and it is beneficial for long running applications to keep `GC` pressure down.

For `F#` developers concerned about performance and memory usage it's recommended to check out Nessos Streams.

If you need top-notch performance strategically placed `Imperative` code is worth considering.

Finally, when it comes to performance don't make assumptions. Measure and Verify.

Full source code:

```
module PushStream =
  type Receiver<'T> = 'T -> bool
  type Stream<'T>   = Receiver<'T> -> unit

  let inline filter (f : 'T -> bool) (s : Stream<'T>) : Stream<'T> =
    fun r -> s (fun v -> if f v then r v else true)

  let inline map (m : 'T -> 'U) (s : Stream<'T>) : Stream<'U> =
    fun r -> s (fun v -> r (m v))

  let inline range b e : Stream<int> =
    fun r ->
      let rec loop i = if i <= e && r i then loop (i + 1)
      loop b

  let inline sum (s : Stream<'T>) : 'T =
    let mutable state = LanguagePrimitives.GenericZero<'T>
    s (fun v -> state <- state + v; true)
    state

module PullStream =

  [<Struct>]
  [<NoComparison>]
  [<NoEqualityAttribute>]
  type Maybe<'T>(v : 'T, hasValue : bool) =
```

```
    member    x.Value        = v
    member    x.HasValue     = hasValue
    override  x.ToString ()  =
      if hasValue then
        sprintf "Just %A" v
      else
        "Nothing"

  let Nothing<'T>     = Maybe<'T> (Unchecked.defaultof<'T>, false)
  let inline Just v   = Maybe<'T> (v, true)

  type Iterator<'T> = unit -> Maybe<'T>
  type Stream<'T>   = unit -> Iterator<'T>

  let filter (f : 'T -> bool) (s : Stream<'T>) : Stream<'T> =
    fun () ->
      let i = s ()
      let rec pop () =
        let mv = i ()
        if mv.HasValue then
          let v = mv.Value
          if f v then Just v else pop ()
        else
          Nothing
      pop

  let map (m : 'T -> 'U) (s : Stream<'T>) : Stream<'U> =
    fun () ->
      let i = s ()
      let pop () =
        let mv = i ()
        if mv.HasValue then
          Just (m mv.Value)
        else
          Nothing
      pop

  let range b e : Stream<int> =
    fun () ->
      let mutable i = b
      fun () ->
        if i <= e then
          let p = i
          i <- i + 1
          Just p
        else
          Nothing

  let inline sum (s : Stream<'T>) : 'T =
    let i = s ()
    let rec loop state =
      let mv = i ()
      if mv.HasValue then
        loop (state + mv.Value)
      else
        state
    loop LanguagePrimitives.GenericZero<'T>

module PerfTest =

  open System.Linq
```

```
#if USE_NESSOS
  open Nessos.Streams
#endif

  let now =
    let sw = System.Diagnostics.Stopwatch ()
    sw.Start ()
    fun () -> sw.ElapsedMilliseconds

  let time n a =
    let inline cc i       = System.GC.CollectionCount i

    let v                 = a ()

    System.GC.Collect (2, System.GCCollectionMode.Forced, true)

    let bcc0, bcc1, bcc2  = cc 0, cc 1, cc 2
    let b                 = now ()

    for i in 1..n do
      a () |> ignore

    let e = now ()
    let ecc0, ecc1, ecc2  = cc 0, cc 1, cc 2

    v, (e - b), ecc0 - bcc0, ecc1 - bcc1, ecc2 - bcc2

  let arrayTest n =
    Array.init (n + 1) id
    |> Array.map    int64
    |> Array.filter (fun v -> v % 2L = 0L)
    |> Array.map    ((+) 1L)
    |> Array.sum

  let imperativeTest n =
    let rec loop s i =
      if i >= 0L then
        if i % 2L = 0L then
          loop (s + i + 1L) (i - 1L)
        else
          loop s (i - 1L)
      else
        s
    loop 0L (int64 n)

  let linqTest n =
    (((Enumerable.Range(0, n + 1)).Select int64).Where(fun v -> v % 2L = 0L)).Select((+)
1L).Sum()

  let listTest n =
    List.init (n + 1) id
    |> List.map     int64
    |> List.filter  (fun v -> v % 2L = 0L)
    |> List.map     ((+) 1L)
    |> List.sum

#if USE_NESSOS
  let nessosTest n =
    Stream.initInfinite id
    |> Stream.take    (n + 1)
    |> Stream.map     int64
```

```
    |> Stream.filter  (fun v -> v % 2L = 0L)
    |> Stream.map     ((+) 1L)
    |> Stream.sum
#endif

  let pullTest n =
    PullStream.range 0 n
    |> PullStream.map     int64
    |> PullStream.filter  (fun v -> v % 2L = 0L)
    |> PullStream.map     ((+) 1L)
    |> PullStream.sum

  let pushTest n =
    PushStream.range 0 n
    |> PushStream.map     int64
    |> PushStream.filter  (fun v -> v % 2L = 0L)
    |> PushStream.map     ((+) 1L)
    |> PushStream.sum

  let seqTest n =
    Seq.init (n + 1) id
    |> Seq.map    int64
    |> Seq.filter (fun v -> v % 2L = 0L)
    |> Seq.map    ((+) 1L)
    |> Seq.sum

  let perfTest (path : string) =
    let testCases =
      [|
        "array"       , arrayTest
        "imperative"  , imperativeTest
        "linq"        , linqTest
        "list"        , listTest
        "seq"         , seqTest
#if USE_NESSOS
        "nessos"      , nessosTest
#endif
        "pull"        , pullTest
        "push"        , pushTest
      |]
    use out                   = new System.IO.StreamWriter (path)
    let write (msg : string)  = out.WriteLine msg
    let writef fmt            = FSharp.Core.Printf.kprintf write fmt

    write "Name\tTotal\tOuter\tInner\tElapsed\tCC0\tCC1\tCC2\tResult"

    let total   = 10000000
    let outers = [| 10; 1000; 1000000 |]
    for outer in outers do
      let inner = total / outer
      for name, a in testCases do
        printfn "Running %s with total=%d, outer=%d, inner=%d ..." name total outer inner
        let v, ms, cc0, cc1, cc2 = time outer (fun () -> a inner)
        printfn "  ... %d ms, cc0=%d, cc1=%d, cc2=%d, result=%A" ms cc0 cc1 cc2 v
        writef "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d" name total outer inner ms cc0 cc1 cc2 v

[<EntryPoint>]
let main argv =
  System.Environment.CurrentDirectory <- System.AppDomain.CurrentDomain.BaseDirectory
  PerfTest.perfTest "perf.tsv"
  0
```

Read F# Performance Tips and Tricks online: https://riptutorial.com/fsharp/topic/3562/fsharp-performance-tips-and-tricks

Chapter 9: Folds

Examples

Intro to folds, with a handful of examples

Folds are (higher-order) functions used with sequences of elements. They collapse `seq<'a>` into `'b` where `'b` is any type (possibly still `'a`). This is a bit abstract so lets get into concrete practical examples.

Calculating the sum of all numbers

In this example, `'a` is an `int`. We have a list of numbers and we want to calculate sum all the numbers of it. To sum the numbers of the list `[1; 2; 3]` we write

```
List.fold (fun x y -> x + y) 0 [1; 2; 3] // val it : int = 6
```

Let me explain, because we are dealing with lists, we use `fold` in the `List` module, hence `List.fold` . the first argument `fold` takes is a binary function, the **folder**. The second argument is the **initial value**. `fold` starts folding the list by consecutively applying the folder function to elements in the list starting with the initial value and the first element. If the list is empty, the inital value is returned!

Schematic overview of an execution example looks like this:

```
let add x y = x + y // this is our folder (a binary function, takes two arguments)
List.fold add 0 [1; 2; 3;]
=> List.fold add (add 0 1) [2; 3]
// the binary function is passed again as the folder in the first argument
// the initial value is updated to add 0 1 = 1 in the second argument
// the tail of the list (all elements except the first one) is passed in the third argument
// it becomes this:
List.fold add 1 [2; 3]
// Repeat untill the list is empty -> then return the "inital" value
List.fold add (add 1 2) [3]
List.fold add 3 [3] // add 1 2 = 3
List.fold add (add 3 3) []
List.fold add 6 [] // the list is empty now -> return 6
6
```

The function `List.sum` is roughly `List.fold add LanguagePrimitives.GenericZero` where the generic zero makes it compatible with integers, floats, big integers etc.

Counting elemets in a list (implementing `count`)

This is done almost the same as above, but by ignoring the actual value of the element in the list and instead adding 1.

```
List.fold (fun x y -> x + 1) 0 [1; 2; 3] // val it : int = 3
```

This can also be done like this:

```
[1; 2; 3]
|> List.map (fun x -> 1) // turn every elemet into 1, [1; 2; 3] becomes [1; 1; 1]
|> List.sum // sum [1; 1; 1] is 3
```

So you could define `count` as follows:

```
let count xs =
    xs
    |> List.map (fun x -> 1)
    |> List.fold (+) 0 // or List.sum
```

Finding the maximum of list

This time we will use `List.reduce` which is like `List.fold` but without an initial value as in this case where we don't know what the type is of the values we are compairing:

```
let max x y = if x > y then x else y
// val max : x:'a -> y: 'a -> 'a when 'a : comparison, so only for types that we can compare
List.reduce max [1; 2; 3; 4; 5] // 5
List.reduce max ["a"; "b"; "c"] // "c", because "c" > "b" > "a"
List.reduce max [true; false] // true, because true > false
```

Finding the minimum of a list

Just like when finding the max, the folder is different

```
let min x y = if x < y then x else y
List.reduce min [1; 2; 3; 4; 5] // 1
List.reduce min ["a"; "b"; "c"] // "a"
List.reduce min [true; false] // false
```

Concatenating lists

Here we are taking list of lists The folder function is the `@` operator

```
// [1;2] @ [3; 4] = [1; 2; 3; 4]
let merge xs ys = xs @ ys
List.fold merge [] [[1;2;3]; [4;5;6]; [7;8;9]] // [1;2;3;4;5;6;7;8;9]
```

Or you could use binary operators as your folder function:

```
List.fold (@) [] [[1;2;3]; [4;5;6]; [7;8;9]] // [1;2;3;4;5;6;7;8;9]
```

```
List.fold (+) 0 [1; 2; 3] // 6
List.fold (||) false [true; false] // true, more on this below
List.fold (&&) true [true; false] // false, more on this below
// etc...
```

Calculating the factorial of a number

Same idea as when summing the numbers, but now we multiply them. if we want the factorial of `n` we multiplty all elements in the list `[1 .. n]`. Code becomes:

```
// the folder
let times x y = x * y
let factorial n = List.fold times 1 [1 .. n]
// Or maybe for big integers
let factorial n = List.fold times 1I [1I .. n]
```

Implementing `forall`, `exists` and `contains`

the function `forall` checks if all elements in a sequence satisfy a condition. `exists` checks if atleast one element in the list satisfy the condition. First we need to know how to collapse a list of `bool` values. Well, we use folds ofcourse! boolean operators will be our folder functions.

To check if all elements in a list are `true` we collapse them with the `&&` function with `true` as initial value.

```
List.fold (&&) true [true; true; true] // true
List.fold (&&) true [] // true, empty list -> return inital value
List.fold (&&) true [false; true] // false
```

Likewise, to check if one element is `true` in a list booleans we collapse it with the `||` operator with `false` as initial value:

```
List.fold (||) false [true; false] // true
List.fold (||) false [false; false] // false, all are false, no element is true
List.fold (||) false [] // false, empty list -> return inital value
```

Back to `forall` and `exists`. Here we take a list of any type, use the condition to transform all elements to boolean values and then we collapse it down:

```
let forall condition elements =
    elements
    |> List.map condition // condition : 'a -> bool
    |> List.fold (&&) true

let exists condition elements =
    elements
    |> List.map condition
    |> List.fold (||) false
```

To check if all elements in [1; 2; 3; 4] are smaller than 5:

```
forall (fun n -> n < 5) [1 .. 4] // true
```

define the `contains` method with `exists`:

```
let contains x xs = exists (fun y -> y = x) xs
```

Or even

```
let contains x xs = exists ((=) x) xs
```

Now check if the list [1 .. 5] contains the value 2:

```
contains 2 [1..5] // true
```

Implementing `reverse`:

```
let reverse xs = List.fold (fun acc x -> x :: acc) [] xs
reverse [1 .. 5] // [5; 4; 3; 2; 1]
```

Implementing `map` and `filter`

```
let map f = List.fold (fun acc x -> List.append acc [f x]) List.empty
// map (fun x -> x * x) [1..5] -> [1; 4; 9; 16; 25]

let filter p = Seq.fold (fun acc x -> seq { yield! acc; if p(x) then yield x }) Seq.empty
// filter (fun x -> x % 2 = 0) [1..10] -> [2; 4; 6; 8; 10]
```

Is there anything `fold` can't do? I dont' really know

Calculating the sum of all elements of a list

To calculate the sum of terms (of type float, int or big integer) of a number list, it is preferable to use List.sum In other cases, List.fold is the function that is best suited to calculate such a sum.

1. Sum of complex numbers

In this example, we declare a list of complex numbers and we calculate the sum of all terms in the list.

At the beginning of the program, add a reference to System.Numerics

open System.Numerics

To calculate the sum, we initialize the accumulator to the complex number 0.

```
let clist = [new Complex(1.0, 52.0); new Complex(2.0, -2.0); new Complex(0.0, 1.0)]

let sum = List.fold (+) (new Complex(0.0, 0.0)) clist
```

Result:

```
(3, 51)
```

2. Sum of numbers of union type

Suppose that a list be composed of numbers of union (float or int) type and want to calculate the sum of these numbers.

Declare before the following number type:

```
type number =
| Float of float
| Int of int
```

Calculate the sum of numbers of type number of a list:

```
let list = [Float(1.3); Int(2); Float(10.2)]

let sum = List.fold (
                        fun acc elem ->
                                      match elem with
                                      | Float(elem) -> acc + elem
                                      | Int(elem) -> acc + float(elem)
                       ) 0.0 list
```

Result:

```
13.5
```

The first parameter of the function, which represents the accumulator is of type float and the second parameter, which represents an item in the list is of type number. But before we add, we need to use a pattern matching and cast to type float when elem is of type Int.

Read Folds online: https://riptutorial.com/fsharp/topic/2250/folds

Chapter 10: Functions

Examples

Functions of more than one parameter

In F#, **all functions take exactly one parameter**. This seems an odd statement, since it's trivially easy to declare more than one parameter in a function declaration:

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

But if you type that function declaration into the F# interactive interpreter, you'll see that its type signature is:

```
val add : x:int -> y:int -> int
```

Without the parameter names, that signature is `int -> int -> int`. The `->` operator is right-associative, which means that this signature is equivalent to `int -> (int -> int)`. In other words, `add` is a function that takes one `int` parameter, and returns **a function that takes one `int` and returns `int`**. Try it:

```
let addTwo = add 2
// val addTwo : (int -> int)
let five = addTwo 3
// val five : int = 5
```

However, you can also call a function like `add` in a more "conventional" manner, directly passing it two parameters, and it will work like you'd expect:

```
let three = add 1 2
// val three : int = 3
```

This applies to functions with as many parameters as you want:

```
let addFiveNumbers a b c d e = a + b + c + d + e
// val addFiveNumbers : a:int -> b:int -> c:int -> d:int -> e:int -> int
let addFourNumbers = addFiveNumbers 0
// val addFourNumbers : (int -> int -> int -> int -> int)
let addThreeNumbers = addFourNumbers 0
// val addThreeNumbers : (int -> int -> int -> int)
let addTwoNumbers = addThreeNumbers 0
// val addTwoNumbers : (int -> int -> int)
let six = addThreeNumbers 1 2 3  // This will calculate 0 + 0 + 1 + 2 + 3
// val six : int = 6
```

This method of thinking about multi-parameter functions as being functions that take one parameter and return new functions (which may in turn take one parameter and return new functions, until you reach the final function that takes the final parameter and finally returns a

result) is called **currying**, in honor of mathematician Haskell Curry, who is famous for developing the concept. (It was invented by someone else, but Curry deservedly gets most of the credit for it.)

This concept is used throughout F#, and you'll want to be familiar with it.

Basics of functions

Most functions in F# are created with the `let` syntax:

```
let timesTwo x = x * 2
```

This defines a function named `timesTwo` that takes a single parameter `x`. If you run an interactive F# session (`fsharpi` on OS X and Linux, `fsi.exe` on Windows) and paste that function in (and add the `;;` that tells `fsharpi` to evaluate the code you just typed), you'll see that it replies with:

```
val timesTwo : x:int -> int
```

This means that `timesTwo` is a function that takes a single parameter `x` of type `int`, and returns an `int`. Function signatures are often written without the parameter names, so you'll often see this function type written as `int -> int`.

But wait! How did F# know that `x` was an integer parameter, since you never specified its type? That's due to **type inference**. Because in the function body, you multiplied `x` by `2`, the types of `x` and `2` must be the same. (As a general rule, F# will not implicitly cast values to different types; you must explicitly specify any typecasts you want).

If you want to create a function that doesn't take any parameters, this is the **wrong** way to do it:

```
let hello =  // This is a value, not a function
    printfn "Hello world"
```

The **right** way to do it is:

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

This `hello` function has the type `unit -> unit`, which is explained in The "unit" type.

Curried vs Tupled Functions

There are two ways to define functions with multiple parameters in F#, Curried functions and Tupled functions.

```
let curriedAdd x y = x + y // Signature: x:int -> y:int -> int
let tupledAdd (x, y) = x + y // Signature:  x:int * y:int -> int
```

All functions defined from outside F# (such as the .NET framework) are used in F# with the Tupled form. Most functions in F# core modules are used with Curried form.

The Curried form is considered idiomatic F#, because it allows for partial application. Neither of the following two examples are possible with the Tupled form.

```
let addTen = curriedAdd 10 // Signature: int -> int

// Or with the Piping operator
3 |> curriedAdd 7 // Evaluates to 10
```

The reason behind that is that the Curried function, when called with one parameter, returns a function. Welcome to functional programming !!

```
let explicitCurriedAdd x = (fun y -> x + y) // Signature: x:int -> y:int -> int
let veryExplicitCurriedAdd = (fun x -> (fun y -> x + y)) // Same signature
```

You can see it is exactly the same signature.

However, when interfacing with other .NET code, as in when writing libraries, it is important to define functions using the Tupled form.

Inlining

Inlining allows you to replace a call to a function with the body of the function.

This is sometimes useful for performance reason on critical part of the code. But the counterpart is that your assembly will takes much space since the body of the function is duplicated everywhere a call occurred. You have to be careful when deciding whether to inline a function or not.

A function can be inlined with the `inline` keyword:

```
// inline indicate that we want to replace each call to sayHello with the body
// of the function
let inline sayHello s1 =
    sprintf "Hello %s" s1

// Call to sayHello will be replaced with the body of the function
let s = sayHello "Foo"
// After inlining ->
// let s = sprintf "Hello %s" "Foo"

printfn "%s" s
// Output
// "Hello Foo"
```

Another example with local value:

```
let inline addAndMulti num1 num2 =
    let add = num1 + num2
    add * 2

let i = addAndMulti 2 2
// After inlining ->
// let add = 2 + 2
// let i = add * 2
```

```
printfn "%i" i
// Output
// 8
```

Pipe Forward and Backward

Pipe operators are used to pass parameters to a function in a simple and elegant way. It allows to eliminate intermediate values and make function calls easier to read.

In F#, there are two pipe operators:

- **Forward** (`|>`): Passing parameters from left to right

```
let print message =
    printf "%s" message

// "Hello World" will be passed as a parameter to the print function
"Hello World" |> print
```

- **Backward** (`<|`): Passing parameters from right to left

```
let print message =
    printf "%s" message

// "Hello World" will be passed as a parameter to the print function
print <| "Hello World"
```

Here is an example without pipe operators:

```
// We want to create a sequence from 0 to 10 then:
// 1 Keep only even numbers
// 2 Multiply them by 2
// 3 Print the number

let mySeq = seq { 0..10 }
let filteredSeq = Seq.filter (fun c -> (c % 2) = 0) mySeq
let mappedSeq = Seq.map ((*) 2) filteredSeq
let finalSeq = Seq.map (sprintf "The value is %i.") mappedSeq

printfn "%A" finalSeq
```

We can shorten the previous example and make it cleaner with the forward pipe operator:

```
// Using forward pipe, we can eliminate intermediate let binding
let finalSeq =
    seq { 0..10 }
    |> Seq.filter (fun c -> (c % 2) = 0)
    |> Seq.map ((*) 2)
    |> Seq.map (sprintf "The value is %i.")

printfn "%A" finalSeq
```

Each function result is passed as a parameter to the next function.

If you want to pass multiple parameters to the pipe operator, you have to add a `|` for each additional parameter and create a Tuple with the parameters. Native F# pipe operator supports up to three parameters (|||> or <|||).

```
let printPerson name age =
    printf "My name is %s, I'm %i years old" name age

("Foo", 20) ||> printPerson
```

Read Functions online: https://riptutorial.com/fsharp/topic/2525/functions

Chapter 11: Generics

Examples

Reversal of a list of any type

To reverse a list, it isn't important what type the list elements are, only what order they're in. This is a perfect candidate for a generic function, so the same reverseal function can be used no matter what list is passed.

```
let rev list =
    let rec loop acc = function
        | []           -> acc
        | head :: tail -> loop (head :: acc) tail
    loop [] list
```

The code makes no assumptions about the types of the elements. The compiler (or F# interactive) will tell you that the type signature of this function is `'T list -> 'T list`. The `'T` tells you that it's a generic type with no constraints. You may also see `'a` instead of `'T` - the letter is unimportant because it's only a *generic* placeholder. We can pass an `int list` or a `string list`, and both will work successfully, returning an `int list` or a `string list` respectively.

For example, in F# interactive:

```
> let rev list = ...
val it : 'T list -> 'T list
> rev [1; 2; 3; 4];;
val it : int list = [4; 3; 2; 1]
> rev ["one", "two", "three"];;
val it : string list = ["three", "two", "one"]
```

Mapping a list into a different type

```
let map f list =
    let rec loop acc = function
        | []           -> List.rev acc
        | head :: tail -> loop (f head :: acc) tail
    loop [] list
```

The signature of this function is `('a -> 'b) -> 'a list -> 'b list`, which is the most generic it can be. This does not prevent `'a` from being the same type as being `'b`, but it also allows them to be different. Here you can see that the `'a` type that is the parameter to the function `f` must match the type of the `list` parameter. This function is still generic, but there are some slight constraints on the inputs - if the types don't match, there will be a compile error.

Examples:

```
> let map f list = ...
```

```
val it : ('a -> 'b) -> 'a list -> 'b list
> map (fun x -> float x * 1.5) [1; 2; 3; 4];;
val it : float list = [1.5; 3.0; 4.5; 6.0]
> map (sprintf "abc%.1f") [1.5; 3.0; 4.5; 6.0];;
val it : string list = ["abc1.5"; "abc3.0"; "abc4.5"; "abc6.0"]
> map (fun x -> x + 1) [1.0; 2.0; 3.0];;
error FS0001: The type 'float' does not match the type 'int'
```

Read Generics online: https://riptutorial.com/fsharp/topic/7731/generics

Chapter 12: Introduction to WPF in F#

Introduction

This topic illustrates how to exploit **Functional Programming** in a **WPF application**. The first example comes from a post by Māris Krivtežs (ref *Remarks* section at the bottom). The reason for revisiting this project is twofold:

1\ The design supports separation of concerns, while the model is kept pure and changes are propagated in a functional way.

2\ The resemblance will make for an easy transition to the Gjallarhorn implementation.

Remarks

Library demo projects @GitHub

- FSharp.ViewModule (under FsXaml)
- Gjallarhorn (ref Samples)

Māris Krivtežs wrote two great posts on this topic:

- F# XAML application - MVVM vs MVC where the pros and cons of both approaches are highlighted.

  I feel that none of these XAML application styles benefit much from functional programming. I imagine that the ideal application would consist of the view which produces events and events hold current view state. All application logic should be handled by filtering and manipulating events and view model, and in the output it should produce a new view model which is bound back to the view.

- F# XAML - event driven MVVM as revisited in the topic above.

Examples

FSharp.ViewModule

Our demo app consists of a scoreboard. The score model is an immutable record. The scoreboard events are contained in a Union Type.

```
namespace Score.Model

type Score = { ScoreA: int ; ScoreB: int }
type ScoringEvent = IncA | DecA | IncB | DecB | NewGame
```

Changes are propagated by listening for events and updating the view model accordingly. Instead

of adding members to the model type, as in OOP, we declare a separate module to host the allowed operations.

```
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Score =
    let zero = {ScoreA = 0; ScoreB = 0}
    let update score event =
        match event with
        | IncA -> {score with ScoreA = score.ScoreA + 1}
        | DecA -> {score with ScoreA = max (score.ScoreA - 1) 0}
        | IncB -> {score with ScoreB = score.ScoreB + 1}
        | DecB -> {score with ScoreB = max (score.ScoreB - 1) 0}
        | NewGame -> zero
```

Our view model derives from `EventViewModelBase<'a>`, which has a property `EventStream` of type `IObservable<'a>`. In this case the events we want to subscribe to are of type `ScoringEvent`.

The controller handles the events in a functional manner. Its signature `Score -> ScoringEvent -> Score` shows us that, whenever an event occurs, the current value of the model is transformed into a new value. This allows for our model to remain pure, although our view model is not.

An `eventHandler` is in charge of mutating the state of the view. Inheriting from `EventViewModelBase<'a>` we can use `EventValueCommand` and `EventValueCommandChecked` to hook up the events to the commands.

```
namespace Score.ViewModel

open Score.Model
open FSharp.ViewModule

type MainViewModel(controller : Score -> ScoringEvent -> Score) as self =
    inherit EventViewModelBase<ScoringEvent>()

    let score = self.Factory.Backing(<@ self.Score @>, Score.zero)

    let eventHandler ev = score.Value <- controller score.Value ev

    do
        self.EventStream
        |> Observable.add eventHandler

    member this.IncA = this.Factory.EventValueCommand(IncA)
    member this.DecA = this.Factory.EventValueCommandChecked(DecA, (fun _ -> this.Score.ScoreA
> 0), [ <@@ this.Score @@> ])
    member this.IncB = this.Factory.EventValueCommand(IncB)
    member this.DecB = this.Factory.EventValueCommandChecked(DecB, (fun _ -> this.Score.ScoreB
> 0), [ <@@ this.Score @@> ])
    member this.NewGame = this.Factory.EventValueCommand(NewGame)

    member __.Score = score.Value
```

The code behind file (*.xaml.fs) is where everything is put together, i.e. the update function (`controller`) is injected in the `MainViewModel`.

```
namespace Score.Views
```

```
open FsXaml

type MainView = XAML<"MainWindow.xaml">

type CompositionRoot() =
    member __.ViewModel = Score.ViewModel.MainViewModel(Score.Model.Score.update)
```

The type `CompositionRoot` serves as a wrapper that's referenced in the XAML file.

```
<Window.Resources>
    <ResourceDictionary>
        <local:CompositionRoot x:Key="CompositionRoot"/>
    </ResourceDictionary>
</Window.Resources>
<Window.DataContext>
    <Binding Source="{StaticResource CompositionRoot}" Path="ViewModel" />
</Window.DataContext>
```

I won't dive any deeper into the XAML file as it's basic WPF stuff, the entire project can be found on GitHub.

Gjallarhorn

The core types in the Gjallarhorn library implement `IObservable<'a>`, which will make the implementation look familiar (remember the `EventStream` property from the FSharp.ViewModule example). The only real change to our model is the order of the arguments of the update function. Also, we now use the term *Message* instead of *Event*.

```
namespace ScoreLogic.Model

type Score = { ScoreA: int ; ScoreB: int }
type ScoreMessage = IncA | DecA | IncB | DecB | NewGame

// Module showing allowed operations
[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]
module Score =
    let zero = {ScoreA = 0; ScoreB = 0}
    let update msg score =
        match msg with
        | IncA -> {score with ScoreA = score.ScoreA + 1}
        | DecA -> {score with ScoreA = max (score.ScoreA - 1) 0}
        | IncB -> {score with ScoreB = score.ScoreB + 1}
        | DecB -> {score with ScoreB = max (score.ScoreB - 1) 0}
        | NewGame -> zero
```

In order to build a UI with Gjallarhorn, instead of making classes to support data binding, we create simple functions referred to as a `Component`. In their constructor the first argument `source` is of type `BindingSource` (defined in Gjallarhorn.Bindable), and used to map the model to the view, and events from the view back into messages.

```
namespace ScoreLogic.Model

open Gjallarhorn
```

```
open Gjallarhorn.Bindable

module Program =

    // Create binding for entire application.
    let scoreComponent source (model : ISignal<Score>) =
        // Bind the score to the view
        model |> Binding.toView source "Score"

        [
            // Create commands that turn into ScoreMessages
            source |> Binding.createMessage "NewGame" NewGame
            source |> Binding.createMessage "IncA" IncA
            source |> Binding.createMessage "DecA" DecA
            source |> Binding.createMessage "IncB" IncB
            source |> Binding.createMessage "DecB" DecB
        ]
```

The current implementation differs from the FSharp.ViewModule version in that two commands do not have CanExecute properly implemented yet. Also listing the application's plumbing.

```
namespace ScoreLogic.Model

open Gjallarhorn
open Gjallarhorn.Bindable

module Program =

    // Create binding for entire application.
    let scoreComponent source (model : ISignal<Score>) =
        let aScored = Mutable.create false
        let bScored = Mutable.create false

        // Bind the score itself to the view
        model |> Binding.toView source "Score"

        // Subscribe to changes of the score
        model |> Signal.Subscription.create
            (fun currentValue ->
                aScored.Value <- currentValue.ScoreA > 0
                bScored.Value <- currentValue.ScoreB > 0)
            |> ignore

        [
            // Create commands that turn into ScoreMessages
            source |> Binding.createMessage "NewGame" NewGame
            source |> Binding.createMessage "IncA" IncA
            source |> Binding.createMessageChecked "DecA" aScored DecA
            source |> Binding.createMessage "IncB" IncB
            source |> Binding.createMessageChecked "DecB" bScored DecB
        ]

    let application =
        // Create our score, wrapped in a mutable with an atomic update function
        let score = new AsyncMutable<_>(Score.zero)

        // Create our 3 functions for the application framework

        // Start with the function to create our model (as an ISignal<'a>)
        let createModel () : ISignal<_> = score :> _
```

```
        // Create a function that updates our state given a message
        // Note that we're just taking the message, passing it directly to our model's update
function,
        // then using that to update our core "Mutable" type.
        let update (msg : ScoreMessage) : unit = Score.update msg |> score.Update |> ignore

        // An init function that occurs once everything's created, but before it starts
        let init () : unit = ()

        // Start our application
        Framework.application createModel init update scoreComponent
```

Left with setting up the decoupled view, combining the `MainWindow` type and the logical application.

```
namespace ScoreBoard.Views

open System

open FsXaml
open ScoreLogic.Model

// Create our Window
type MainWindow = XAML<"MainWindow.xaml">

module Main =
    [<STAThread>]
    [<EntryPoint>]
    let main _ =
        // Run using the WPF wrappers around the basic application framework
        Gjallarhorn.Wpf.Framework.runApplication System.Windows.Application MainWindow
Program.application
```

This sums up the core concepts, for additional information and a more elaborate example please refer to Reed Copsey's post. The *Christmas Trees* project highlights a couple of benefits to this approach:

- Effectively redeeming us from the need to (manually) copy the model into a custom collection of view models, managing them, and manually constructing the model back from the results.
- Updates within collections are done in a transparent manner, while maintaining a pure model.
- The logic and view are hosted by two different projects, emphasizing the separation of concerns.

Read Introduction to WPF in F# online: https://riptutorial.com/fsharp/topic/8758/introduction-to-wpf-in-fsharp

Chapter 13: Lazy Evaluation

Examples

Lazy Evaluation Introduction

Most programming languages, including F#, evaluate computations immediately in accord with a model called Strict Evaluation. However, in Lazy Evaluation, computations are not evaluated until they are needed. F# allows us to use lazy evaluation through both the `lazy` keyword and `sequences`.

```
// define a lazy computation
let comp = lazy(10 + 20)

// we need to force the result
let ans = comp.Force()
```

In addition, when using Lazy Evaluation, the results of the computation are cached so if we force the result after our first instance of forcing it, the expression itself won't be evaluated again

```
let rec factorial n =
  if n = 0 then
    1
  else
    (factorial (n - 1)) * n


let computation = lazy(printfn "Hello World\n"; factorial 10)

// Hello World will be printed
let ans = computation.Force()

// Hello World will not be printed here
let ansAgain = computation.Force()
```

Introduction to Lazy Evaluation in F#

F#, like most programming languages, uses Strict Evaluation by default. In Strict Evaluation, computations are executed immediately. In contrast, Lazy Evaluation, defers execution of computations until their results are needed. Moreover, the results of a computation under Lazy Evaluation are cached, thereby obviating the need for the re-evaluation of an expression.

We can use Lazy evaluation in F# through both the `lazy` keyword and `Sequences`

```
// 23 * 23 is not evaluated here
// lazy keyword creates lazy computation whose evaluation is deferred
let x = lazy(23 * 23)

// we need to force the result
let y = x.Force()
```

```
// Hello World not printed here
let z = lazy(printfn "Hello World\n"; 23424)

// Hello World printed and 23424 returned
let ans1 = z.Force()

// Hello World not printed here as z as already been evaluated, but 23424 is
// returned
let ans2 = z.Force()
```

Read Lazy Evaluation online: https://riptutorial.com/fsharp/topic/3682/lazy-evaluation

Chapter 14: Lists

Syntax

- [] // an empty list.

  head::tail // a construction cell holding an element, head, and a list, tail. :: is called the Cons operator.

  let list1 = [1; 2; 3] // Note the usage of a semicolon.

  let list2 = 0 :: list1 // result is [0; 1; 2; 3]

  let list3 = list1 @ list2 // result is [1; 2; 3; 0; 1; 2; 3]. @ is the append operator.

  let list4 = [1..3] // result is [1; 2; 3]

  let list5 = [1..2..10] // result is [1; 3; 5; 7; 9]

  let list6 = [for i in 1..10 do if i % 2 = 1 then yield i] // result is [1; 3; 5; 7; 9]

Examples

Basic List Usage

```
let list1 = [ 1; 2 ]
let list2 = [ 1 .. 100 ]

// Accessing an element
printfn "%A" list1.[0]

// Pattern matching
let rec patternMatch aList =
    match aList with
    | [] -> printfn "This is an empty list"
    | head::tail -> printfn "This list consists of a head element %A and a tail list %A" head
tail
                    patternMatch tail

patternMatch list1

// Mapping elements
let square x = x*x
let list2squared = list2
                   |> List.map square
printfn "%A" list2squared
```

Calculating the total sum of numbers in a list

By recursion

```
let rec sumTotal list =
    match list with
    | [] -> 0 // empty list -> return 0
    | head :: tail -> head + sumTotal tail
```

The above example says: "Look at the `list`, is it empty? return 0. Otherwise it is a non-empty list. So it could be `[1]`, `[1; 2]`, [1; 2; 3] etc. If `list`is [1] then bind the variable `head` to `1` and `tail` to `[]` then execute `head + sumTotal tail`.

Example execution:

```
sumTotal [1; 2; 3]
// head -> 1, tail -> [2; 3]
1 + sumTotal [2; 3]
1 + (2 + sumTotal [3])
1 + (2 + (3 + sumTotal [])) // sumTotal [] is defined to be 0, recursion stops here
1 + (2 + (3 + 0)))
1 + (2 + 3)
1 + 5
6
```

A more general way to encapsulate the above pattern is by using functional folds! `sumTotal` becomes this:

```
let sumTotal list = List.fold (+) 0 list
```

Creating lists

A way to create a list is to place elements in two square brackets, separated by semicolons. The elements must have the same type.

Example:

```
> let integers = [1; 2; 45; -1];;
val integers : int list = [1; 2; 45; -1]


> let floats = [10.7; 2.0; 45.3; -1.05];;
val floats : float list = [10.7; 2.0; 45.3; -1.05]
```

When a list has no element, it is empty. An empty list can be declared as follows:

```
> let emptyList = [];;
val emptyList : 'a list
```

Other example

To create a list of byte, simply to cast the integers:

```
> let bytes = [byte(55); byte(10); byte(100)];;
val bytes : byte list = [55uy; 10uy; 100uy]
```

It is also possible to define lists of functions, of elements of a type defined previously, of objects of a class, etc.

Example

```
> type number = | Real of float | Integer of int;;

type number =
  | Real of float
  | Integer of int

> let numbers = [Integer(45); Real(0.0); Integer(127)];;
val numbers : number list = [Integer 45; Real 0.0; Integer 127]
```

Ranges

For certain types of elements (int, float, char,...), it is possible to define a list by the start element and the end element, using the following template:

```
[start..end]
```

Examples:

```
> let c=['a' .. 'f'];;
val c : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f']

let f=[45 .. 60];;
val f : int list =
  [45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60]
```

You can also specify a step for certain types, with the following model:

```
[start..step..end]
```

Examples:

```
> let i=[4 .. 2 .. 11];;
val i : int list = [4; 6; 8; 10]

> let r=[0.2 .. 0.05 .. 0.28];;
val r : float list = [0.2; 0.25]
```

Generator

Another way to create a list is to generate it automatically by using generator.

We can use one of the following models:

```
[for <identifier> in range -> expr]
```

or

```
[for <identifier> in range do ... yield expr]
```

Examples

```
> let oddNumbers = [for i in 0..10 -> 2 * i + 1];; // odd numbers from 1 to 21
val oddNumbers : int list = [1; 3; 5; 7; 9; 11; 13; 15; 17; 19; 21]

> let multiples3Sqrt = [for i in 1..27 do if i % 3 = 0 then yield sqrt(float(i))];; //sqrt of
multiples of 3 from 3 to 27
val multiples3Sqrt : float list =
  [1.732050808; 2.449489743; 3.0; 3.464101615; 3.872983346; 4.242640687;    4.582575695;
4.898979486; 5.196152423]
```

Operators

Some operators may be used to construct lists:

Cons operator ::

This operator :: is used to add a head element to a list:

```
> let l=12::[] ;;
val l : int list = [12]

> let l1=7::[14; 78; 0] ;;
val l1 : int list = [7; 14; 78; 0]

> let l2 = 2::3::5::7::11::[13;17] ;;
val l2 : int list = [2; 3; 5; 7; 11; 13; 17]
```

Concatenation

The concatenation of lists is carried out with the operator @.

```
> let l1 = [12.5;89.2];;
val l1 : float list = [12.5; 89.2]

> let l2 = [1.8;7.2] @ l1;;
val l2 : float list = [1.8; 7.2; 12.5; 89.2]
```

Read Lists online: https://riptutorial.com/fsharp/topic/1268/lists

Chapter 15: Mailbox Processor

Remarks

`MailboxProcessor` maintains an internal message queue, where multiple producers can post messages using various `Post` method variants. These messages are then retrieved and processed by a single consumer (unless you implement it otherwise) using `Retrieve` and `Scan` variants. By default both producing and consuming the messages is thread-safe.

By default there is no provided error handling. If an uncaught exception is thrown inside the processor's body, the body function will end, all messages in the queue will be lost, no more messages can be posted and the reply channel (if available) will get an exception instead of a response. You have to provide all error handling yourself in case this behavior does not suit your use case.

Examples

Basic Hello World

Let's first create a simple "Hello world!" `MailboxProcessor` which processes one type of message and prints greetings.

You'll need the message type. It can be anything, but Discriminated Unions are a natural choice here as they list all the possible cases on one place and you can easily use pattern matching when processing them.

```
// In this example there is only a single case, it could technically be just a string
type GreeterMessage = SayHelloTo of string
```

Now define the processor itself. This can be done with `MailboxProcessor<'message>.Start` static method which returns a started processor ready to do its job. You can also use the constructor, but then you need to make sure to start the processor later.

```
let processor = MailboxProcessor<GreeterMessage>.Start(fun inbox ->
    let rec innerLoop () = async {
        // This way you retrieve message from the mailbox queue
        // or await them in case the queue empty.
        // You can think of the `inbox` parameter as a reference to self.
        let! message = inbox.Receive()
        // Now you can process the retrieved message.
        match message with
        | SayHelloTo name ->
            printfn "Hi, %s! This is mailbox processor's inner loop!" name
        // After that's done, don't forget to recurse so you can process the next messages!
        innerLoop()
    }
    innerLoop ())
```

The parameter to `Start` is a function which takes a reference to the `MailboxProcessor` itself (which doesn't exist yet as you are just creating it, but will be available once the function executes). That gives you access to its various `Receive` and `Scan` methods to access the messages from the mailbox. Inside this function, you can do whatever processing you need, but a usual approach is an infinite loop that reads the messages one by one and calls itself after each one.

Now the processor is ready, but it doesn't to anything! Why? You need to send it a message to process. This is done with the `Post` method variants - let's use the most basic, fire-and-forget one.

```
processor.Post(SayHelloTo "Alice")
```

This puts a message to `processor`'s internal queue, the mailbox, and immediately returns so that the calling code can continue. Once the processor retrieves the message, it will process it, but that will be done asynchronously to posting it, and it will be most likely done on a separate thread.

Very soon afterwards you should see the message `"Hi, Alice! This is mailbox processor's inner loop!"` printed to the output and you're ready for more complicated samples.

Mutable State Management

Mailbox processors can be used to manage mutable state in a transparent and thread-safe way. Let's build a simple counter.

```
// Increment or decrement by one.
type CounterMessage =
    | Increment
    | Decrement

let createProcessor initialState =
    MailboxProcessor<CounterMessage>.Start(fun inbox ->
        // You can represent the processor's internal mutable state
        // as an immutable parameter to the inner loop function
        let rec innerLoop state = async {
            printfn "Waiting for message, the current state is: %i" state
            let! message = inbox.Receive()
            // In each call you use the current state to produce a new
            // value, which will be passed to the next call, so that
            // next message sees only the new value as its local state
            match message with
            | Increment ->
                let state' = state + 1
                printfn "Counter incremented, the new state is: %i" state'
                innerLoop state'
            | Decrement ->
                let state' = state - 1
                printfn "Counter decremented, the new state is: %i" state'
                innerLoop state'
        }
        // We pass the initialState to the first call to innerLoop
        innerLoop initialState)

// Let's pick an initial value and create the processor
let processor = createProcessor 10
```

Now let's generate some operations

```
processor.Post(Increment)
processor.Post(Increment)
processor.Post(Decrement)
processor.Post(Increment)
```

And you will see the following log

```
Waiting for message, the current state is: 10
Counter incremented, the new state is: 11
Waiting for message, the current state is: 11
Counter incremented, the new state is: 12
Waiting for message, the current state is: 12
Counter decremented, the new state is: 11
Waiting for message, the current state is: 11
Counter incremented, the new state is: 12
Waiting for message, the current state is: 12
```

Concurrency

Since mailbox processor processes the messages one by one and there is no interleaving, you can also produce the messages from multiple threads and you will not see the the typical problems of lost or duplicated operations. There is no way for a message to use the old state of other messages, unless you specifically implement the processor so.

```
let processor = createProcessor 0

[ async { processor.Post(Increment) }
  async { processor.Post(Increment) }
  async { processor.Post(Decrement) }
  async { processor.Post(Decrement) } ]
|> Async.Parallel
|> Async.RunSynchronously
```

All messages are posted from different threads. The order in which messages are posted to the mailbox is not deterministic, so the order of processing them is not deterministic, but since the overall number of increments and decrements is balanced, you will see the final state being 0, no matter in what order and from which threads the messages were sent.

True mutable state

In the previous example we've only simulated mutable state by passing the recursive loop parameter, but mailbox processor has all these properties even for a truly mutable state. This is important when you maintain large state and immutability is impractical for performance reasons.

We can rewrite our counter to the following implementation

```
let createProcessor initialState =
    MailboxProcessor<CounterMessage>.Start(fun inbox ->
```

```
        // In this case we represent the state as a mutable binding
        // local to this function. innerLoop will close over it and
        // change its value in each iteration instead of passing it around
        let mutable state = initialState

        let rec innerLoop () = async {
            printfn "Waiting for message, the current state is: %i" state
            let! message = inbox.Receive()
            match message with
            | Increment ->
                let state <- state + 1
                printfn "Counter incremented, the new state is: %i" state'
                innerLoop ()
            | Decrement ->
                let state <- state - 1
                printfn "Counter decremented, the new state is: %i" state'
                innerLoop ()
        }
        innerLoop ())
```

Even though this would definitely not be thread safe if the counter state was modified directly from multiple threads, you can see by using the parallel message Posts from previous section that mailbox processor processes the messages one after another with no interleaving, so each message uses the most current value.

Return Values

You can asynchronously return a value for each processed message if you send an `AsyncReplyChannel<'a>` as part of the message.

```
type MessageWithResponse = Message of InputData * AsyncReplyChannel<OutputData>
```

Then the mailbox processor can use this channel when processing the message to send a value back to the caller.

```
let! message = inbox.Receive()
match message with
| MessageWithResponse (data, r) ->
    // ...process the data
    let output = ...
    r.Reply(output)
```

Now to create a message, you need the `AsyncReplyChannel<'a>` - what is is and how do you create a working instance? The best way is to let MailboxProcessor provide it for you and extract the response to a more common `Async<'a>`. This can be done by using for example the `PostAndAsynReply` method, where you don't post the complete message, but instead a function of type (in our case) `AsyncReplyChannel<OutputData> -> MessageWithResponse`:

```
let! output = processor.PostAndAsyncReply(r -> MessageWithResponse(input, r))
```

This will post the message in a queue and await the reply, which will arrive once the processor gets to this message and replies using the channel.

There is also a synchronous variant `PostAndReply` which blocks the calling thread until the processor replies.

Out-of-Order Message Processing

You can use `Scan` or `TryScan` methods to look for specific messages in the queue and process them regardless of how many messages are before them. Both methods look at the messages in the queue in the order they arrived and will look for a specified message (up until optional timeout). In case there is no such message, `TryScan` will return None, while `Scan` will keep waiting until such message arrives or the operation times out.

Let's see it in practice. We want the processor to process `RegularOperations` when it can, but whenever there is a `PriorityOperation`, it should be processed as soon as possible, no matter how many other `RegularOperations` are in the queue.

```
type Message =
    | RegularOperation of string
    | PriorityOperation of string

let processor = MailboxProcessor<Message>.Start(fun inbox ->
    let rec innerLoop () = async {
        let! priorityData = inbox.TryScan(fun msg ->
            // If there is a PriorityOperation, retrieve its data.
            match msg with
            | PriorityOperation data -> Some data
            | _ -> None)

        match priorityData with
        | Some data ->
            // Process the data from PriorityOperation.
        | None ->
            // No PriorityOperation was in the queue at the time, so
            // let's fall back to processing all possible messages
            let! message = inbox.Receive()
            match message with
            | RegularOperation data ->
                // We have free time, let's process the RegularOperation.
            | PriorityOperation data ->
                // We did scan the queue, but it might have been empty
                // so it is possible that in the meantime a producer
                // posted a new message and it is a PriorityOperation.
        // And never forget to process next messages.
        innerLoop ()
    }
    innerLoop())
```

Read Mailbox Processor online: https://riptutorial.com/fsharp/topic/9409/mailbox-processor

Chapter 16: Memoization

Examples

Simple memoization

Memoization consists of caching function results to avoid computing the same result multiple times. This is useful when working with functions that perform costly computations.

We can use a simple factorial function as an example:

```
let factorial index =
    let rec innerLoop i acc =
        match i with
        | 0 | 1 -> acc
        | _ -> innerLoop (i - 1) (acc * i)

    innerLoop index 1
```

Calling this function multiple times with the same parameter results in the same computation again and again.

Memoization will help us to cache the factorial result and return it if the same parameter is passed again.

Here is a simple memoization implementation :

```
// memoization takes a function as a parameter
// This function will be called every time a value is not in the cache
let memoization f =
    // The dictionary is used to store values for every parameter that has been seen
    let cache = Dictionary<_,_>()
    fun c ->
        let exist, value = cache.TryGetValue (c)
        match exist with
        | true ->
            // Return the cached result directly, no method call
            printfn "%O -> In cache" c
            value
        | _ ->
            // Function call is required first followed by caching the result for next call
with the same parameters
            printfn "%O -> Not in cache, calling function..." c
            let value = f c
            cache.Add (c, value)
            value
```

The `memoization` function simply takes a function as a parameter and returns a function with the same signature. You could see this in the method signature `f:('a -> 'b) -> ('a -> 'b)`. This way you can use memoization the same way as if you were calling the factorial method.

The `printfn` calls are to show what happens when we call the function multiple times; they can be removed safely.

Using memoization is easy:

```
// Pass the function we want to cache as a parameter via partial application
let factorialMem = memoization factorial

// Then call the memoized function
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 10)
printfn "%i" (factorialMem 4)
printfn "%i" (factorialMem 4)

// Prints
// 10 -> Not in cache, calling function...
// 3628800
// 10 -> In cache
// 3628800
// 10 -> In cache
// 3628800
// 4 -> Not in cache, calling function...
// 24
// 4 -> In cache
// 24
```

Memoization in a recursive function

Using the previous example of calculating the factorial of an integer, put in the hash table all values of factorial calculated inside the recursion, that do not appear in the table.

As in the article about memoization, we declare a function `f` that accepts a function parameter `fact` and a integer parameter. This function `f`, includes instructions to calculate the factorial of `n` from `fact(n-1)`.

This allows to handle recursion by the function returned by `memorec` and not by `fact` itself and possibly stop the calculation if the `fact(n-1)` value has been already calculated and is located in the hash table.

```
let memorec f =
   let cache = Dictionary<_,_>()
   let rec frec n =
       let value = ref 0
       let exist = cache.TryGetValue(n, value)
       match exist with
       | true ->
           printfn "%O -> In cache" n
       | false ->
           printfn "%O -> Not in cache, calling function..." n
           value := f frec n
           cache.Add(n, !value)
       !value
   in frec
```

```
let f = fun fact n -> if n<2 then 1 else n*fact(n-1)

[<EntryPoint>]
let main argv =
    let g = memorec(f)
    printfn "%A" (g 10)
    printfn "%A" "----------------------"
    printfn "%A" (g 5)
    Console.ReadLine()
    0
```

Result:

```
10 -> Not in cache, calling function...
9 -> Not in cache, calling function...
8 -> Not in cache, calling function...
7 -> Not in cache, calling function...
6 -> Not in cache, calling function...
5 -> Not in cache, calling function...
4 -> Not in cache, calling function...
3 -> Not in cache, calling function...
2 -> Not in cache, calling function...
1 -> Not in cache, calling function...
3628800
"----------------------"
5 -> In cache
120
```

Read Memoization online: https://riptutorial.com/fsharp/topic/2698/memoization

Chapter 17: Monads

Examples

Understanding Monads comes from practice

This topic is intended for intermediate to advanced F# developers

"What are Monads?" is a common question. This is easy to answer but like in Hitchhikers guide to galaxy we realize we don't understand the answer because we didn't know what we were asking after.

Many believe the way to understanding Monads is by practicing them. As programmers we typically don't care for the mathematical foundation for what Liskov's Substitution Principle, sub-types or sub-classes are. By using these ideas we have acquired an intuition for what they represent. The same is true for Monads.

In order to help you get started with Monads this example demonstrates how to build a Monadic Parser Combinator library. This might help you get started but understanding will come from writing your own Monadic library.

Enough prose, time for code

The Parser type:

```
// A Parser<'T> is a function that takes a string and position
//  and returns an optionally parsed value and a position
//  A parsed value means the position points to the character following the parsed value
//  No parsed value indicates a parse failure at the position
type Parser<'T> = Parser of (string*int -> 'T option*int)
```

Using this definition of a Parser we define some fundamental parser functions

```
// Runs a parser 't' on the input string 's'
let run t s =
  let (Parser tps) = t
  tps (s, 0)

// Different ways to create parser result
let succeedWith v p = Some v, p
let failAt        p = None  , p

// The 'satisfy' parser succeeds if the character at the current position
//  passes the 'sat' function
let satisfy sat : Parser<char> = Parser <| fun (s, p) ->
  if p < s.Length && sat s.[p] then succeedWith s.[p] (p + 1)
  else failAt p

// 'eos' succeeds if the position is beyond the last character.
//  Useful when testing if the parser have consumed the full string
let eos : Parser<unit> = Parser <| fun (s, p) ->
```

```
  if p < s.Length then failAt p
  else succeedWith () p

let anyChar       = satisfy (fun _ -> true)
let char ch       = satisfy ((=) ch)
let digit         = satisfy System.Char.IsDigit
let letter        = satisfy System.Char.IsLetter
```

`satisfy` is a function that given a `sat` function produces a parser that succeeds if we haven't passed `EOS` and the character at the current position passes the `sat` function. Using `satisfy` we create a number of useful character parsers.

Running this in FSI:

```
> run digit "";;
val it : char option * int = (null, 0)
> run digit "123";;
val it : char option * int = (Some '1', 1)
> run digit "hello";;
val it : char option * int = (null, 0)
```

We have some fundamental parsers into place. We will combine them into more powerful parsers using parser combinator functions

```
// 'fail' is a parser that always fails
let fail<'T>      = Parser <| fun (s, p) -> failAt p
// 'return_' is a parser that always succeed with value 'v'
let return_ v     = Parser <| fun (s, p) -> succeedWith v p

// 'bind' let us combine two parser into a more complex parser
let bind t uf     = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    -> None, tp
  | Some tv ->
    let u = uf tv
    let (Parser ups) = u
    ups (s, tp)
```

The names and signatures are not arbitrarily chosen but we will not delve on this, instead let's see how we use `bind` to combine parser into more complex ones:

```
> run (bind digit (fun v -> digit)) "123";;
val it : char option * int = (Some '2', 2)
> run (bind digit (fun v -> bind digit (fun u -> return_ (v,u)))) "123";;
val it : (char * char) option * int = (Some ('1', '2'), 2)
> run (bind digit (fun v -> bind digit (fun u -> return_ (v,u)))) "1";;
val it : (char * char) option * int = (null, 1)
```

What this shows us is that `bind` allows us to combine two parsers into a more complex parser. As the result of `bind` is a parser that in turn can be combined again.

```
> run (bind digit (fun v -> bind digit (fun w -> bind digit (fun u -> return_ (v,w,u)))))
```

```
"123";;
val it : (char * char * char) option * int = (Some ('1', '2', '3'), 3)
```

`bind` will be the fundamental way we combine parsers although we will define helper functions to simplify the syntax.

One of the things that can simplify syntax are computation expressions. They are easy to define:

```
type ParserBuilder() =
  member x.Bind       (t, uf) = bind      t   uf
  member x.Return     v       = return_   v
  member x.ReturnFrom t       = t

// 'parser' enables us to combine parsers using 'parser { ... }' syntax
let parser = ParserBuilder()
```

FSI

```
let p = parser {
          let! v = digit
          let! u = digit
          return v,u
        }
run p "123"
val p : Parser<char * char> = Parser <fun:bind@49-1>
val it : (char * char) option * int = (Some ('1', '2'), 2)
```

This is equivalent to:

```
> let p = bind digit (fun v -> bind digit (fun u -> return_ (v,u)))
run p "123";;
val p : Parser<char * char> = Parser <fun:bind@49-1>
val it : (char * char) option * int = (Some ('1', '2'), 2)
```

Another fundamental parser combinator we are going to use alot is `orElse`:

```
// 'orElse' creates a parser that runs parser 't' first, if that is successful
//  the result is returned otherwise the result of parser 'u' is returned
let orElse t u    = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    ->
    let (Parser ups) = u
    ups (s, p)
  | Some tv -> succeedWith tv tp
```

This allows us to define `letterOrDigit` like this:

```
> let letterOrDigit = orElse letter digit;;
val letterOrDigit : Parser<char> = Parser <fun:orElse@70-1>
> run letterOrDigit "123";;
val it : char option * int = (Some '1', 1)
> run letterOrDigit "hello";;
```

```
val it : char option * int = (Some 'h', 1)
> run letterOrDigit "!!!";;
val it : char option * int = (null, 0)
```

A note on Infix operators

A common concern over FP is the use of unusual infix operators like `>>=`, `>=>`, `<-` and so on. However, most aren't concerned over the use of `+`, `-`, `*`, `/` and `%`, these are well known operators used to compose values. However, a big part in FP is about composing not just values but functions as well. To an intermediate FP developer the infix operators `>>=`, `>=>`, `<-` are well-known and should have specific signatures as well as semantics.

For the functions we have defined so far we would define the following infix operators used to combine parsers:

```
let (>>=)   t   uf  = bind t uf
let (<|>)   t   u   = orElse t u
```

So `>>=` means `bind` and `<|>` means `orElse`.

This allows us combine parsers more succinct:

```
let letterOrDigit = letter <|> digit
let p = digit >>= fun v -> digit >>= fun u -> return_ (v,u)
```

In order to define some advanced parser combinators that will allow us to parse more complex expression we define a few more simple parser combinators:

```
// 'map' runs parser 't' and maps the result using 'm'
let map m t       = t >>= (m >> return_)
let (>>!) t m     = map m t
let (>>%) t v     = t >>! (fun _ -> v)

// 'opt' takes a parser 't' and creates a parser that always succeed but
//  if parser 't' fails the new parser will produce the value 'None'
let opt t         = (t >>! Some) <|> (return_ None)

// 'pair' runs parser 't' and 'u' and returns a pair of 't' and 'u' results
let pair t u      =
  parser {
    let! tv = t
    let! tu = u
    return tv, tu
  }
```

We are ready to define `many` and `sepBy` which are more advanced as they apply the input parsers until they fail. Then `many` and `sepBy` returns the aggregated result:

```
// 'many' applies parser 't' until it fails and returns all successful
//  parser results as a list
let many t =
  let ot = opt t
  let rec loop vs = ot >>= function Some v -> loop (v::vs) | None -> return_ (List.rev vs)
```

```
  loop []

// 'sepBy' applies parser 't' separated by 'sep'.
//  The values are reduced with the function 'sep' returns
let sepBy t sep     =
  let ots = opt (pair sep t)
  let rec loop v = ots >>= function Some (s, n) -> loop (s v n) | None -> return_ v
  t >>= loop
```

Creating a simple expression parser

With the tools we created we can now define a parser for simple expressions like `1+2*3`

We start from the bottom by defining a parser for integers `pint`

```
// 'pint' parses an integer
let pint =
  let f s v = 10*s + int v - int '0'
  parser {
    let! digits = many digit
    return!
      match digits with
      | [] -> fail
      | vs -> return_ (List.fold f 0 vs)
  }
```

We try to parse as many digits as we can, the result is `char list`. If the list is empty we `fail`, otherwise we fold the characters into an integer.

Testing `pint` in FSI:

```
> run pint "123";;
val it : int option * int = (Some 123, 3)
```

In addition we need to parse the different kind of operators used to combine integer values:

```
// operator parsers, note that the parser result is the operator function
let padd      = char '+' >>% (+)
let psubtract = char '-' >>% (-)
let pmultiply = char '*' >>% (*)
let pdivide   = char '/' >>% (/)
let pmodulus  = char '%' >>% (%)
```

FSI:

```
> run padd "+";;
val it : (int -> int -> int) option * int = (Some <fun:padd@121-1>, 1)
```

Tying it all together:

```
// 'pmullike' parsers integers separated by operators with same precedence as multiply
let pmullike  = sepBy pint (pmultiply <|> pdivide <|> pmodulus)
// 'paddlike' parsers sub expressions separated by operators with same precedence as add
```

```
let paddlike  = sepBy pmullike (padd <|> psubtract)
// 'pexpr' is the full expression
let pexpr     =
  parser {
    let! v = paddlike
    let! _ = eos      // To make sure the full string is consumed
    return v
  }
```

Running it all in FSI:

```
> run pexpr "2+123*2-3";;
val it : int option * int = (Some 245, 9)
```

Conclusion

By defining `Parser<'T>`, `return_`, `bind` and making sure they obey the monadic laws we have built a simple but powerful Monadic Parser Combinator framework.

Monads and Parsers go together because Parsers are executed on a parser state. Monads allows us to combine parsers while hiding the parser state thus reducing clutter and improving composability.

The framework we have created is slow and produces no error messages, this in order to keep the code succinct. FParsec provide both acceptable performance as well as excellent error messages.

However, an example alone cannot give understanding of Monads. One has to practice Monads.

Here are some examples on Monads you can try to implement in order to reach your won understanding:

1. State Monad - Allows hidden environment state to be carried implicitly
2. Tracer Monad - Allows trace state to be carried implicitly. A variant of State Monad
3. Turtle Monad - A Monad for creating Turtle (Logos) programs. A variant of State Monad
4. Continuation Monad - A coroutine Monad. An example of this is `async` in F#

The best thing in order to learn would be to come up with an application for Monads in a domain you are comfortable with. For me that was parsers.

Full source code:

```
// A Parser<'T> is a function that takes a string and position
//  and returns an optionally parsed value and a position
//  A parsed value means the position points to the character following the parsed value
//  No parsed value indicates a parse failure at the position
type Parser<'T> = Parser of (string*int -> 'T option*int)

// Runs a parser 't' on the input string 's'
let run t s =
  let (Parser tps) = t
  tps (s, 0)

// Different ways to create parser result
```

```
let succeedWith v p = Some v, p
let failAt        p = None  , p

// The 'satisfy' parser succeeds if the character at the current position
//  passes the 'sat' function
let satisfy sat : Parser<char> = Parser <| fun (s, p) ->
  if p < s.Length && sat s.[p] then succeedWith s.[p] (p + 1)
  else failAt p

// 'eos' succeeds if the position is beyond the last character.
//  Useful when testing if the parser have consumed the full string
let eos : Parser<unit> = Parser <| fun (s, p) ->
  if p < s.Length then failAt p
  else succeedWith () p

let anyChar       = satisfy (fun _ -> true)
let char ch       = satisfy ((=) ch)
let digit         = satisfy System.Char.IsDigit
let letter        = satisfy System.Char.IsLetter

// 'fail' is a parser that always fails
let fail<'T>      = Parser <| fun (s, p) -> failAt p
// 'return_' is a parser that always succeed with value 'v'
let return_ v     = Parser <| fun (s, p) -> succeedWith v p

// 'bind' let us combine two parser into a more complex parser
let bind t uf     = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    -> None, tp
  | Some tv ->
    let u = uf tv
    let (Parser ups) = u
    ups (s, tp)

type ParserBuilder() =
  member x.Bind       (t, uf) = bind      t   uf
  member x.Return     v       = return_   v
  member x.ReturnFrom t       = t

// 'parser' enables us to combine parsers using 'parser { ... }' syntax
let parser = ParserBuilder()

// 'orElse' creates a parser that runs parser 't' first, if that is successful
//  the result is returned otherwise the result of parser 'u' is returned
let orElse t u    = Parser <| fun (s, p) ->
  let (Parser tps) = t
  let tov, tp = tps (s, p)
  match tov with
  | None    ->
    let (Parser ups) = u
    ups (s, p)
  | Some tv -> succeedWith tv tp

let (>>=) t uf    = bind t uf
let (<|>) t u     = orElse t u

// 'map' runs parser 't' and maps the result using 'm'
let map m t       = t >>= (m >> return_)
let (>>!) t m     = map m t
```

```
let (>>%) t v     = t >>! (fun _ -> v)

// 'opt' takes a parser 't' and creates a parser that always succeed but
//  if parser 't' fails the new parser will produce the value 'None'
let opt t         = (t >>! Some) <|> (return_ None)

// 'pair' runs parser 't' and 'u' and returns a pair of 't' and 'u' results
let pair t u      =
  parser {
    let! tv = t
    let! tu = u
    return tv, tu
  }

// 'many' applies parser 't' until it fails and returns all successful
//  parser results as a list
let many t =
  let ot = opt t
  let rec loop vs = ot >>= function Some v -> loop (v::vs) | None -> return_ (List.rev vs)
  loop []

// 'sepBy' applies parser 't' separated by 'sep'.
//  The values are reduced with the function 'sep' returns
let sepBy t sep     =
  let ots = opt (pair sep t)
  let rec loop v = ots >>= function Some (s, n) -> loop (s v n) | None -> return_ v
  t >>= loop

// A simplistic integer expression parser

// 'pint' parses an integer
let pint =
  let f s v = 10*s + int v - int '0'
  parser {
    let! digits = many digit
    return!
      match digits with
      | [] -> fail
      | vs -> return_ (List.fold f 0 vs)
  }

// operator parsers, note that the parser result is the operator function
let padd      = char '+' >>% (+)
let psubtract = char '-' >>% (-)
let pmultiply = char '*' >>% (*)
let pdivide   = char '/' >>% (/)
let pmodulus  = char '%' >>% (%)

// 'pmullike' parsers integers separated by operators with same precedence as multiply
let pmullike  = sepBy pint (pmultiply <|> pdivide <|> pmodulus)
// 'paddlike' parsers sub expressions separated by operators with same precedence as add
let paddlike  = sepBy pmullike (padd <|> psubtract)
// 'pexpr' is the full expression
let pexpr     =
  parser {
    let! v = paddlike
    let! _ = eos      // To make sure the full string is consumed
    return v
  }
```

Computation Expressions provide an alternative syntax to chain Monads

Related to Monads are `F#` computation expressions (`CE`). A programmer typically implements a `CE` to provide an alternative approach to chaining Monads, ie instead of this:

```
let v = m >>= fun x -> n >>= fun y -> return_ (x, y)
```

You can write this:

```
let v = ce {
    let! x = m
    let! y = n
    return x, y
  }
```

Both styles are equivalent and it's up to developer preference which one to pick.

In order to demonstrate how to implement a `CE` imagine you like all traces to include a correlation id. This correlation id will help correlating traces that belong to the same call. This is very useful when have log files that contains traces from concurrent calls.

The problem is that it's cumbersome to include the correlation id as an argument to all functions. As Monads allows carrying implicit state we will define a Log Monad to hide the log context (ie the correlation id).

We begin by defining a log context and the type of a function that traces with log context:

```
type Context =
  {
    CorrelationId : Guid
  }
  static member New () : Context = { CorrelationId = Guid.NewGuid () }

type Function<'T> = Context -> 'T

// Runs a Function<'T> with a new log context
let run t = t (Context.New ())
```

We also define two trace functions that will log with the correlation id from the log context:

```
let trace v   : Function<_> = fun ctx -> printfn "CorrelationId: %A - %A" ctx.CorrelationId v
let tracef fmt              = kprintf trace fmt
```

`trace` is a `Function<unit>` which means it will be passed a log context when invoked. From the log context we pick up the correlation id and traces it together with `v`

In addition we define `bind` and `return_` and as they follow the Monad Laws this forms our Log Monad.

```
let bind t uf : Function<_> = fun ctx ->
  let tv = t ctx  // Invoke t with the log context
```

```
  let u  = uf tv  // Create u function using result of t
  u ctx           // Invoke u with the log context

// >>= is the common infix operator for bind
let inline (>>=) (t, uf) = bind t uf

let return_ v : Function<_> = fun ctx -> v
```

Finally we define `LogBuilder` that will enable us to use `CE` syntax to chain `Log` Monads.

```
type LogBuilder() =
  member x.Bind   (t, uf) = bind t uf
  member x.Return v       = return_ v

// This enables us to write function like: let f = log { ... }
let log = Log.LogBuilder ()
```

We can now define our functions that should have the implicit log context:

```
let f x y =
  log {
    do! Log.tracef "f: called with: x = %d, y = %d" x y
    return x + y
  }

let g =
  log {
    do! Log.trace "g: starting..."
    let! v = f 1 2
    do! Log.tracef "g: f produced %d" v
    return v
  }
```

We execute g with:

```
printfn "g produced %A" (Log.run g)
```

Which prints:

```
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "g: starting..."
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "f: called with: x = 1, y = 2"
CorrelationId: 33342765-2f96-42da-8b57-6fa9cdaf060f - "g: f produced 3"
g produced 3
```

Notice that the CorrelationId is implicitly carried from `run` to `g` to `f` which allows us the correlate the log entries during trouble shooting.

`CE` has lot more features but this should help you get started defining your own `CE`:s.

Full code:

```
module Log =
  open System
  open FSharp.Core.Printf
```

```
  type Context =
    {
      CorrelationId : Guid
    }
    static member New () : Context = { CorrelationId = Guid.NewGuid () }

  type Function<'T> = Context -> 'T

  // Runs a Function<'T> with a new log context
  let run t = t (Context.New ())

  let trace v   : Function<_> = fun ctx -> printfn "CorrelationId: %A - %A" ctx.CorrelationId
v
  let tracef fmt              = kprintf trace fmt

  let bind t uf : Function<_> = fun ctx ->
    let tv = t ctx  // Invoke t with the log context
    let u  = uf tv  // Create u function using result of t
    u ctx           // Invoke u with the log context

  // >>= is the common infix operator for bind
  let inline (>>=) (t, uf) = bind t uf

  let return_ v : Function<_> = fun ctx -> v

  type LogBuilder() =
    member x.Bind   (t, uf) = bind t uf
    member x.Return v       = return_ v

// This enables us to write function like: let f = log { ... }
let log = Log.LogBuilder ()

let f x y =
  log {
    do! Log.tracef "f: called with: x = %d, y = %d" x y
    return x + y
  }

let g =
  log {
    do! Log.trace "g: starting..."
    let! v = f 1 2
    do! Log.tracef "g: f produced %d" v
    return v
  }

[<EntryPoint>]
let main argv =
  printfn "g produced %A" (Log.run g)
  0
```

Read Monads online: https://riptutorial.com/fsharp/topic/3320/monads

Chapter 18: Operators

Examples

How to compose values and functions using common operators

In Object Oriented Programming a common task is to compose objects (values). In Functional Programming it is as common task to compose values as well as functions.

We are used to compose values from our experience of other programming languages using operators like `+`, `-`, `*`, `/` and so on.

Value composition

```
let x = 1 + 2 + 3 * 2
```

As functional programming composes functions as well as values it's not surprising there are common operators for function composition like `>>`, `<<`, `|>` and `<|`.

Function composition

```
// val f : int -> int
let f v   = v + 1
// val g : int -> int
let g v   = v * 2

// Different ways to compose f and g
// val h : int -> int
let h1 v  = g (f v)
let h2 v  = v |> f |> g   // Forward piping of 'v'
let h3 v  = g <| (f <| v) // Reverse piping of 'v' (because <| has left associcativity we need
())
let h4    = f >> g        // Forward functional composition
let h5    = g << f        // Reverse functional composition (closer to math notation of 'g o
f')
```

In `F#` forward piping is preferred over reverse piping because:

1. Type inference (generally) flows from left-to-right so it's natural for values and functions to flow left-to-right as well
2. Because `<|` and `<<` should have right associativity but in `F#` they are left associative which forces us to insert ()
3. Mixing forward and reverse piping generally don't work because they have the same precedence.

Monad composition

As Monads (like `Option<'T>` or `List<'T>`) are commonly used in functional programming there are also common but less known operators to compose functions working with Monads like `>>=`, `>=>`,

`<|>` and `<*>`.

```
let (>>=) t uf  = Option.bind uf t
let (>=>) tf uf = fun v -> tf v >>= uf
// val oinc   : int -> int option
let oinc   v    = Some (v + 1)    // Increment v
// val ofloat : int -> float option
let ofloat v    = Some (float v)  // Map v to float

// Different ways to compose functions working with Option Monad
// val m : int option -> float option
let m1 v  = Option.bind (fun v -> Some (float (v + 1))) v
let m2 v  = v |> Option.bind oinc |> Option.bind ofloat
let m3 v  = v >>= oinc >>= ofloat
let m4    = oinc >=> ofloat

// Other common operators are <|> (orElse) and <*> (andAlso)

// If 't' has Some value then return t otherwise return u
let (<|>) t u =
  match t with
  | Some _  -> t
  | None    -> u

// If 't' and 'u' has Some values then return Some (tv*uv) otherwise return None
let (<*>) t u =
  match t, u with
  | Some tv, Some tu  -> Some (tv, tu)
  | _                 -> None

// val pickOne : 'a option -> 'a option -> 'a option
let pickOne t u v = t <|> u <|> v

// val combine : 'a option -> 'b option  -> 'c option -> (('a*'b)*'c) option
let combine t u v = t <*> u <*> v
```

Conclusion

To new functional programmers function composition using operators might seem opaque and obscure but that is because the meaning of these operators aren't as commonly known as operators working on values. However, with some training using `|>`, `>>` , `>>=` and `>=>` becomes as natural as using `+`, `-`, `*` and `/`.

Latebinding in F# using ? operator

In a statically typed language like `F#` we work with types well-known at compile-time. We consume external data sources in a type-safe manner using type providers.

However, occassionally there's need to use late binding (like `dynamic` in `C#`). For instance when working with `JSON` documents that have no well-defined schema.

To simplify working with late binding `F#` provides supports dynamic lookup operators `?` and `?<-`.

Example:

```
// (?) allows us to lookup values in a map like this: map?MyKey
let inline (?)   m k   = Map.tryFind k m
// (?<-) allows us to update values in a map like this: map?MyKey <- 123
let inline (?<-) m k v = Map.add k v m

let getAndUpdate (map : Map<string, int>) : int option*Map<string, int> =
  let i = map?Hello       // Equivalent to map |> Map.tryFind "Hello"
  let m = map?Hello <- 3  // Equivalent to map |> Map.add "Hello" 3
  i, m
```

It turns out that the `F#` support for late binding is simple yet flexible.

Read Operators online: https://riptutorial.com/fsharp/topic/4641/operators

Chapter 19: Option types

Examples

Definition of Option

An `Option` is a discriminated union with two cases, `None` or `Some`.

```
type Option<'T> = Some of 'T | None
```

Use Option<'T> over null values

In functional programming languages like `F# null` values are considered potentially harmful and poor style (non-idiomatic).

Consider this `C#` code:

```
string x = SomeFunction ();
int    l = x.Length;
```

`x.Length` will throw if `x` is `null` let's add protection:

```
string x = SomeFunction ();
int    l = x != null ? x.Length : 0;
```

Or:

```
string x = SomeFunction () ?? "";
int    l = x.Length;
```

Or:

```
string x = SomeFunction ();
int    l = x?.Length;
```

In idiomatic `F# null` values aren't used so our code looks like this:

```
let x = SomeFunction ()
let l = x.Length
```

However, sometimes there's a need for representing empty or invalid values. Then we can use `Option<'T>`:

```
let SomeFunction () : string option = ...
```

`SomeFunction` either returns `Some string` value or `None`. We extract the `string` value using pattern

matching

```
let v =
  match SomeFunction () with
  | Some x  -> x.Length
  | None    -> 0
```

The reason this code is less fragile than:

```
string x = SomeFunction ();
int    l = x.Length;
```

Is because we can't call `Length` on a `string option`. We need to extract the `string` value using pattern matching and by doing so we are guaranteed that the `string` value is safe to use.

Option Module enables Railway Oriented Programming

Error handling is important but can make an elegant algorithm into a mess. Railway Oriented Programming (`ROP`) is used to make error handling elegant and composable.

Consider the simple function `f`:

```
let tryParse s =
  let b, v = System.Int32.TryParse s
  if b then Some v else None

let f (g : string option) : float option =
  match g with
  | None    -> None
  | Some s  ->
    match tryParse s with           // Parses string to int
    | None              -> None
    | Some v when v < 0 -> None     // Checks that int is greater than 0
    | Some v -> v |> float |> Some  // Maps int to float
```

The purpose of `f` is to parse the input `string` value (if there is `Some`) into an `int`. If the `int` is greater than `0` we cast it into a `float`. In all other cases we bail out with `None`.

Although, an extremely simple function the nested `match` decrease readability significantly.

`ROP` observes we have two kind of execution paths in our program

1. Happy path - Will eventually compute `Some` value
2. Error path - All other paths produces `None`

Since the error paths are more frequent they tend to take over the code. We would like that the happy path code is the most visible code path.

An equivalent function `g` using `ROP` could look like this:

```
let g (v : string option) : float option =
  v
```

```
  |> Option.bind    tryParse  // Parses string to int
  |> Option.filter  ((<) 0)   // Checks that int is greater than 0
  |> Option.map     float     // Maps int to float
```

It looks a lot like how we tend to process lists and sequences in `F#`.

One can see an `Option<'T>` like a `List<'T>` that only may contain `0` or `1` element where `Option.bind` behaves like `List.pick` (conceptually `Option.bind` maps better to `List.collect` but `List.pick` might be easier to understand).

`bind`, `filter` and `map` handles the error paths and `g` only contain the happy path code.

All functions that directly accepts `Option<_>` and returns `Option<_>` are directly composable with `|>` and `>>`.

`ROP` therefore increases readability and composability.

Using Option types from C#

It is not a good idea to expose Option types to C# code, as C# does not have a way to handle them. The options are either to introduce `FSharp.Core` as a dependency in your C# project (which is what you'd have to do if you're consuming an F# library *not* designed for interop with C#), or to change `None` values to `null`.

Pre-F# 4.0

The way to do this is create a conversion function of your own:

```
let OptionToObject opt =
    match opt with
    | Some x -> x
    | None -> null
```

For value types you'd have to resort to boxing them or using `System.Nullable`.

```
let OptionToNullable x =
    match x with
    | Some i -> System.Nullable i
    | None -> System.Nullable ()
```

F# 4.0

In F# 4.0, the functions `ofObj`, `toObj`, `ofNullable`, and `toNullable` where introduced to the `Option` module. In F# interactive they can be used as follows:

```
let l1 = [ Some 1 ; None ; Some 2]
let l2 = l1 |> List.map Option.toNullable;;
```

```
// val l1 : int option list = [Some 1; null; Some 2]
// val l2 : System.Nullable<int> list = [1; null; 2]

let l3 = l2 |> List.map Option.ofNullable;;
// val l3 : int option list = [Some 1; null; Some 2]

// Equality
l1 = l2 // fails to compile: different types
l1 = l3 // true
```

Note that `None` compiles to `null` internally. However as far as F# is concerned it is a `None`.

```
let lA = [Some "a"; None; Some "b"]
let lB = lA |> List.map Option.toObj

// val lA : string option list = [Some "a"; null; Some "b"]
// val lB : string list = ["a"; null; "b"]

let lC = lB |> List.map Option.ofObj
// val lC : string option list = [Some "a"; null; Some "b"]

// Equality
lA = lB // fails to compile: different types
lA = lC // true
```

Read Option types online: https://riptutorial.com/fsharp/topic/3175/option-types

Chapter 20: Pattern Matching

Remarks

Pattern Matching is a powerful feature of many functional languages as it often allows branching to be handled very succinctly compared to using multiple `if`/`else if`/`else` style statements. However given enough options and "when" guards, Pattern Matching can also become verbose and difficult to understand at a glance.

When this happens F#'s Active Patterns can be a great way to give meaningful names to the matching logic, which simplifies the code and also enables reuse.

Examples

Matching Options

Pattern matching can be useful to handle Options:

```
let result = Some("Hello World")
match result with
| Some(message) -> printfn message
| None -> printfn "Not feeling talkative huh?"
```

Pattern matching checks the entire domain is covered

```
let x = true
match x with
| true -> printfn "x is true"
```

yields a warning

> C:\Program Files (x86)\Microsoft VS Code\Untitled-1(2,7): warning FS0025: Incomplete pattern matches on this expression. For example, the value 'false' may indicate a case not covered by the pattern(s).

This is because not all of the possible bool values were covered.

bools can be explicitly listed but ints are harder to list out

```
let x = 5
match x with
```

```
| 1 -> printfn "x is 1"
| 2 -> printfn "x is 2"
| _ -> printfn "x is something else"
```

here we use the special _ character. The _ matches all other possible cases.

The _ can get you into trouble

consider a type we create ourselves it looks like this

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
```

We might write a match with expession that looks like this

```
match sobriety with
| Sober -> printfn "drive home"
| _ -> printfn "call an uber"
```

The above code makes sense. We are assuming if you aren't sober you should call an uber so we use the _ to denote that

We later refactor our code to this

```
type Sobriety =
    | Sober
    | Tipsy
    | Drunk
    | Unconscious
```

The F# compiler should give us a warning and prompt us to refactor our match expression to have the person seek medical attention. Instead the match expression silently treats the unconscious person as if they were only tipsy. The point is you should opt to explicitly list out cases when possible to avoid logic errors.

Cases are evaluated from top to bottom and the first match is used

Incorrect usage:

In the following snippet, the last match will never be used:

```
let x = 4
match x with
| 1 -> printfn "x is 1"
| _ -> printfn "x is anything that wasn't listed above"
| 4 -> printfn "x is 4"
```

prints

> x is anything that wasn't listed above

Correct usage:

Here, both `x = 1` and `x = 4` will hit their specific cases, while everything else will fall through to the default case _:

```
let x = 4
match x with
| 1 -> printfn "x is 1"
| 4 -> printfn "x is 4"
| _ -> printfn "x is anything that wasn't listed above"
```

prints

> x is 4

When guards let you add arbitrary conditionals

```
type Person = {
    Age : int
    PassedDriversTest : bool }

let someone = { Age = 19; PassedDriversTest = true }

match someone.PassedDriversTest with
| true when someone.Age >= 16 -> printfn "congrats"
| true -> printfn "wait until you are 16"
| false -> printfn "you need to pass the test"
```

Read Pattern Matching online: https://riptutorial.com/fsharp/topic/1335/pattern-matching

Chapter 21: Porting C# to F#

Examples

POCOs

Some of the simplest kinds of classes are POCOs.

```
// C#
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime Birthday { get; set; }
}
```

In F# 3.0, auto-properties similar to C# auto-properties were introduced,

```
// F#
type Person() =
    member val FirstName = "" with get, set
    member val LastName = "" with get, set
    member val BirthDay = System.DateTime.Today with get, set
```

Creation of an instance of either is similar,

```
// C#
var person = new Person { FirstName = "Bob", LastName = "Smith", Birthday = DateTime.Today };
// F#
let person = new Person(FirstName = "Bob", LastName = "Smith")
```

If you can use immutable values, a record type is much more idiomatic F#.

```
type Person = {
    FirstName:string;
    LastName:string;
    Birthday:System.DateTime
}
```

And this record can be created:

```
let person = { FirstName = "Bob"; LastName = "Smith"; Birthday = System.DateTime.Today }
```

Records can also be created based on other records by specifiying the existing record and adding `with`, then a list of fields to override:

```
let formal = { person with FirstName = "Robert" }
```

Class Implementing an Interface

Classes implement an interface to meet the interface's contract. For example, a C# class may implement `IDisposable`,

```
public class Resource : IDisposable
{
    private MustBeDisposed internalResource;

    public Resource()
    {
        internalResource = new MustBeDisposed();
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing){
        if (disposing){
            if (resource != null) internalResource.Dispose();
        }
    }
}
```

To implement an interface in F#, use `interface` in the type definition,

```
type Resource() =
    let internalResource = new MustBeDisposed()

    interface IDisposable with
        member this.Dispose(): unit =
            this.Dispose(true)
            GC.SuppressFinalize(this)

    member __.Dispose disposing =
        match disposing with
        | true  -> if (not << isNull) internalResource then internalResource.Dispose()
        | false -> ()
```

Read Porting C# to F# online: https://riptutorial.com/fsharp/topic/6828/porting-csharp-to-fsharp

Chapter 22: Records

Examples

Add member functions to records

```
type person = {Name: string; Age: int} with   // Defines person record
    member this.print() =
        printfn "%s, %i" this.Name this.Age

let user = {Name = "John Doe"; Age = 27} // creates a new person

user.print() // John Doe, 27
```

Basic usage

```
type person = {Name: string; Age: int}   // Defines person record

let user1 = {Name = "John Doe"; Age = 27} // creates a new person
let user2 = {user1 with Age = 28}         // creates a copy, with different Age
let user3 = {user1 with Name = "Jane Doe"; Age = 29} //creates a copy with different Age and
Name

let printUser user =
    printfn "Name: %s, Age: %i" user.Name user.Age

printUser user1 // Name: John Doe, Age: 27
printUser user2 // Name: John Doe, Age: 28
printUser user3 // Name: Jane Doe, Age: 29
```

Read Records online: https://riptutorial.com/fsharp/topic/1136/records

Chapter 23: Reflection

Examples

Robust reflection using F# quotations

Reflection is useful but fragile. Consider this:

```
let mi  = typeof<System.String>.GetMethod "StartsWith"
```

The problems with this kind of code are:

1. The code doesn't work because there are several overloads of `String.StartsWith`
2. Even if there wouldn't be any overloads right now later versions of the library might add an overload causing a runtime crash
3. Refactoring tools like `Rename methods` is broken with reflection.

This means we get a runtime crashes for something that is known compile-time. That seems suboptimal.

Using `F#` quotations it's possible to avoid all the problems above. We define some helper functions:

```
open FSharp.Quotations
open System.Reflection

let getConstructorInfo (e : Expr<'T>) : ConstructorInfo =
  match e with
  | Patterns.NewObject (ci, _) -> ci
  | _ -> failwithf "Expression has the wrong shape, expected NewObject (_, _) instead got: %A"
e


let getMethodInfo (e : Expr<'T>) : MethodInfo =
  match e with
  | Patterns.Call (_, mi, _) -> mi
  | _ -> failwithf "Expression has the wrong shape, expected Call (_, _, _) instead got: %A" e
```

We use the functions like this:

```
printfn "%A" <| getMethodInfo <@ "".StartsWith "" @>
printfn "%A" <| getMethodInfo <@ List.singleton 1 @>
printfn "%A" <| getConstructorInfo <@ System.String [||] @>
```

This prints:

```
Boolean StartsWith(System.String)
Void .ctor(Char[])
Microsoft.FSharp.Collections.FSharpList`1[System.Int32] Singleton[Int32](Int32)
```

`<@ ... @>`

means that instead of executing the expression inside `F#` generates an expression tree representing the expression. `<@ "".StartsWith "" @>` generates an expression tree that looks like this: `Call (Some (Value ("")), StartsWith, [Value ("")])`. This expression tree matches what `getMethodInfo` expects and it will return the correct method info.

This resolves all the problems listed above.

Read Reflection online: https://riptutorial.com/fsharp/topic/4124/reflection

Chapter 24: Sequence

Examples

Generate sequences

There are multiple ways to create a sequence.

You can use functions from the Seq module:

```
// Create an empty generic sequence
let emptySeq = Seq.empty

// Create an empty int sequence
let emptyIntSeq = Seq.empty<int>

// Create a sequence with one element
let singletonSeq = Seq.singleton 10

// Create a sequence of n elements with the specified init function
let initSeq = Seq.init 10 (fun c -> c * 2)

// Combine two sequence to create a new one
let combinedSeq = emptySeq |> Seq.append singletonSeq

// Create an infinite sequence using unfold with generator based on state
let naturals = Seq.unfold (fun state -> Some(state, state + 1)) 0
```

You can also use sequence expression:

```
// Create a sequence with element from 0 to 10
let intSeq = seq { 0..10 }

// Create a sequence with an increment of 5 from 0 to 50
let intIncrementSeq = seq{ 0..5..50 }

// Create a sequence of strings, yield allow to define each element of the sequence
let stringSeq = seq {
    yield "Hello"
    yield "World"
}

// Create a sequence from multiple sequence, yield! allow to flatten sequences
let flattenSeq = seq {
    yield! seq { 0..10 }
    yield! seq { 11..20 }
}
```

Introduction to sequences

A sequence is a series of elements that can be enumerated. It is an alias of System.Collections.Generic.IEnumerable and lazy. It stores a series of elements of the same type

(can be any value or object, even another sequence). Functions from the Seq.module can be used to operate on it.

Here is a simple example of a sequence enumeration:

```
let mySeq = { 0..20 } // Create a sequence of int from 0 to 20
mySeq
|> Seq.iter (printf "%i ") // Enumerate each element of the sequence and print it
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Seq.map

```
let seq = seq {0..10}

s |> Seq.map (fun x -> x * 2)

> val it : seq<int> = seq [2; 4; 6; 8; ...]
```

Apply a function to every element of a sequence using Seq.map

Seq.filter

Suppose that we have a sequence of integers and we want to create a sequence that contains only the even integers. We can obtain the latter by using the `filter` function of the Seq module. The `filter` function has the type signature `('a -> bool) -> seq<'a> -> seq<'a>`; this indicates that it accepts a function that returns true or false (sometimes called a predicate) for a given input of type `'a` and a sequence that comprises values of type `'a` to yield a sequence that comprises values of type `'a`.

```
// Function that tests if an integer is even
let isEven x = (x % 2) = 0

// Generates an infinite sequence that contains the natural numbers
let naturals = Seq.unfold (fun state -> Some(state, state + 1)) 0

// Can be used to filter the naturals sequence to get only the even numbers
let evens = Seq.filter isEven naturals
```

Infinite repeating sequences

```
let data = [1; 2; 3; 4; 5;]
let repeating = seq {while true do yield! data}
```

Repeating sequences can be created using a `seq {}` computation expression

Read Sequence online: https://riptutorial.com/fsharp/topic/2354/sequence

Chapter 25: Sequence Workflows

Examples

yield and yield!

In sequence workflows, `yield` adds a single item into the sequence being built. (In monadic terminology, it is `return`.)

```
> seq { yield 1; yield 2; yield 3 }
val it: seq<int> = seq [1; 2; 3]

> let homogenousTup2ToSeq (a, b) = seq { yield a; yield b }
> tup2Seq ("foo", "bar")
val homogenousTup2ToSeq: 'a * 'a -> seq<'a>
val it: seq<string> = seq ["foo"; "bar"]
```

`yield!` (pronounced *yield bang*) inserts all the items of another sequence into this sequence being built. Or, in other words, it appends a sequence. (In relation to monads, it is `bind`.)

```
> seq { yield 1; yield! [10;11;12]; yield 20 }
val it: seq<int> = seq [1; 10; 11; 12; 20]

// Creates a sequence containing the items of seq1 and seq2 in order
> let concat seq1 seq2 = seq { yield! seq1; yield! seq2 }
> concat ['a'..'c'] ['x'..'z']
val concat: seq<'a> -> seq<'a> -> seq<'a>
val it: seq<int> = seq ['a'; 'b'; 'c'; 'x'; 'y'; 'z']
```

Sequences created by sequence workflows are also lazy, meaning that items of the sequence don't actually get evaluated until they're needed. A few ways to force items include calling `Seq.take` (pulls the first n items into a sequence), `Seq.iter` (applies a function to each item for executing side effects), or `Seq.toList` (converts a sequence to a list). Combining this with recursion is where `yield!` really starts to shine.

```
> let rec numbersFrom n = seq { yield n; yield! numbersFrom (n + 1) }
> let naturals = numbersFrom 0
val numbersFrom: int -> seq<int>
val naturals: seq<int> = seq [0; 1; 2; ...]

// Just like Seq.map: applies a mapping function to each item in a sequence to build a new
sequence
> let rec map f seq1 =
      if Seq.isEmpty seq1 then Seq.empty
      else seq { yield f (Seq.head seq1); yield! map f (Seq.tail seq1) }
> map (fun x -> x * x) [1..10]
val map: ('a -> 'b) -> seq<'a> -> 'b
val it: seq<int> = seq [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

for

`for` sequence expression is designed to look just like its more famous cousin, the imperative for-loop. It "loops" through a sequence and evaluates the body of each iteration into the sequence it is generating. Just like everything sequence related, it is NOT mutable.

```
> let oneToTen = seq { for x in 1..10 -> x }
val oneToTen: seq<int> = seq [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
// Or, equivalently:
> let oneToTen = seq { for x in 1..10 do yield x }
val oneToTen: seq<int> = seq [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

// Just like Seq.map: applies a mapping function to each item in a sequence to build a new
sequence
> let map mapping seq1 = seq { for x in seq1 do yield mapping x }
> map (fun x -> x * x) [1..10]
val map: ('a -> 'b) -> seq<'a> -> seq<'b>
val it: seq<int> = seq [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]

// An infinite sequence of consecutive integers starting at 0
> let naturals =
      let numbersFrom n = seq { yield n; yield! numbersFrom (n + 1) }
      numbersFrom 0
// Just like Seq.filter: returns a sequence consisting only of items from the input sequence
that satisfy the predicate
> let filter predicate seq1 = seq { for x in seq1 do if predicate x then yield x }
> let evenNaturals = naturals |> filter (fun x -> x % 2 = 0)
val naturals: seq<int> = seq [1; 2; 3; ...]
val filter: ('a -> bool) -> seq<'a> -> seq<'a>
val evenNaturals: seq<int> = seq [2; 4; 6; ...]

// Just like Seq.concat: concatenates a collection of sequences together
> let concat seqSeq = seq { for seq in seqSeq do yield! seq }
> concat [[1;2;3];[10;20;30]]
val concat: seq<#seq<'b>> -> seq<'b>
val it: seq<int> = seq [1; 2; 3; 10; 20; 30]
```

Read Sequence Workflows online: https://riptutorial.com/fsharp/topic/2785/sequence-workflows

Chapter 26: Statically Resolved Type Parameters

Syntax

- `s` is an instance of `^a` you want to accept at compile-time, which can be anything that implements the members you actually call using the syntax.
- `^a` is similar to generics which would be `'a` (or `'A` or `'T` for example) but these are compile-time resolved, and allow for anything that fits all the requested usages within the method. (no interfaces required)

Examples

Simple usage for anything that has a Length member

```
let inline getLength s = (^a: (member Length: _) s)
//usage:
getLength "Hello World" // or "Hello World" |> getLength
// returns 11
```

Class, Interface, Record usage

```
// Record
type Ribbon = {Length:int}
// Class
type Line(len:int) =
    member x.Length = len
type IHaveALength =
    abstract Length:int

let inline getLength s = (^a: (member Length: _) s)
let ribbon = {Length=1}
let line = Line(3)
let someLengthImplementer =
    { new IHaveALength with
        member __.Length = 5}
printfn "Our ribbon length is %i" (getLength ribbon)
printfn "Our Line length is %i" (getLength line)
printfn "Our Object expression length is %i" (getLength someLengthImplementer)
```

Static member call

this will accept any type with a method named `GetLength` that takes nothing and returns an int:

```
((^a : (static member GetLength : int) ()))
```

Read Statically Resolved Type Parameters online:
https://riptutorial.com/fsharp/topic/7228/statically-resolved-type-parameters

Chapter 27: Strings

Examples

String literals

```
let string1 = "Hello" //simple string

let string2 = "Line\nNewLine" //string with newline escape sequence

let string3 = @"Line\nSameLine" //use @ to create a verbatim string literal

let string4 = @"Line""with""quoutes inside" //double quoute to indicate a single quoute inside
@ string

let string5 = """single "quote" is ok""" //triple-quote string literal, all symbol including
quote are verbatim

let string6 = "ab
cd"// same as "ab\ncd"

let string7 = "xx\
    yy" //same as "xxyy", backslash at the end contunies the string without new line, leading
whitespace on the next line is ignored
```

Simple string formatting

There are several ways to format and get a string as a result.

The .NET way is by using `String.Format` or `StringBuilder.AppendFormat`:

```
open System
open System.Text

let hello = String.Format ("Hello {0}", "World")
// return a string with "Hello World"

let builder = StringBuilder()
let helloAgain = builder.AppendFormat ("Hello {0} again!", "World")
// return a StringBuilder with "Hello World again!"
```

F# has also functions to format string in a C-style. There are equivalents for each .NET functions:

- `sprintf` (String.Format) :

```
open System

let hello = sprintf "Hello %s" "World"
// "Hello World", "%s" is for string

let helloInt = sprintf "Hello %i" 42
// "Hello 42", "%i" is for int
```

```
let helloFloat = sprintf "Hello %f" 4.2
// "Hello 4.2000", "%f" is for float

let helloBool = sprintf "Hello %b" true
// "Hello true", "%b" is for bool

let helloNativeType = sprintf "Hello %A again!" ("World", DateTime.Now)
// "Hello {formatted date}", "%A" is for native type

let helloObject = sprintf "Hello %O again!" DateTime.Now
// "Hello {formatted date}", "%O" is for calling ToString
```

- `bprintf` (StringBuilder.AppendFormat):

```
open System
open System.Text

let builder = StringBuilder()

// Attach the StringBuilder to the format function with partial application
let append format = Printf.bprintf builder format

// Same behavior as sprintf but strings are appended to a StringBuilder
append "Hello %s again!\n" "World"
append "Hello %i again!\n" 42
append "Hello %f again!\n" 4.2
append "Hello %b again!\n" true
append "Hello %A again!\n" ("World", DateTime.Now)
append "Hello %O again!\n" DateTime.Now

builder.ToString() // Get the result string
```

Using those functions instead of the .NET functions provides some advantages:

- Type safety
- Partial application
- F# native type support

Read Strings online: https://riptutorial.com/fsharp/topic/1397/strings

Chapter 28: The "unit" type

Examples

What good is a 0-tuple?

A 2-tuple or a 3-tuple represent a group of related items. (Points in 2D space, RGB values of a color, etc.) A 1-tuple is not very useful since it could easily be replaced with a single `int`.

A 0-tuple seems even more useless since it contains absolutely *nothing*. Yet it has properties that make it very useful in functional languages like F#. For example, the 0-tuple type has exactly *one* value, usually represented as `()`. All 0-tuples have this value so it's essentially a singleton type. In most functional programming languages, including F#, this is called the `unit` type.

Functions that return `void` in C# will return the `unit` type in F#:

```
let printResult = printfn "Hello"
```

Run that in the F# interactive interpreter, and you'll see:

```
val printResult : unit = ()
```

This means that the value `printResult` is of type `unit`, and has the value `()` (the empty tuple, the one and only value of the `unit` type).

Functions can take the `unit` type as a parameter, too. In F#, functions may look like they're taking no parameters. But in fact, they're taking a single parameter of type `unit`. This function:

```
let doMath() = 2 + 4
```

is actually equivalent to:

```
let doMath () = 2 + 4
```

That is, a function that takes one parameter of type `unit` and returns the `int` value 6. If you look at the type signature that the F# interactive interpreter prints when you define this function, you'll see:

```
val doMath : unit -> int
```

The fact that all functions will take at least one parameter and return a value, even if that value is sometimes a "useless" value like `()`, means that function composition is a lot easier in F# than in languages that don't have the `unit` type. But that's a more advanced subject which we'll get to later on. For now, just remember that when you see `unit` in a function signature, or `()` in a function's parameters, that's the 0-tuple type that serves as the way to say "This function takes, or returns, no meaningful values."

Deferring execution of code

We can use the `unit` type as a function argument to define functions that we don't want executed until later. This is often useful in asynchronous background tasks, when the main thread may want trigger some predefined functionality of the background thread, like maybe moving it to a new file, or if a a let-binding should not be run immediately:

```
module Time =
    let now = System.DateTime.Now   // value is set and fixed for duration of program
    let now() = System.DateTime.Now // value is calculated when function is called (each time)
```

In the following code, we define code to start a "worker" which simply prints out the value it is working on every 2 seconds. The worker then returns two functions which may be used to control it - one which moves it to the next value to work on, and one which stops it from working. These must be functions, because we do not want their bodies to be executed until we choose to, otherwise the worker would immediately move to the second value and shutdown without having done anything.

```
let startWorker value =
    let current = ref value
    let stop = ref false
    let nextValue () = current := !current + 1
    let stopOnNextTick () = stop := true
    let rec loop () = async {
        if !stop then
            printfn "Stopping work."
            return ()
        else
            printfn "Working on %d." !current
            do! Async.Sleep 2000
            return! loop () }
    Async.Start (loop ())
    nextValue, stopOnNextTick
```

We can then start a worker by doing

```
let nextValue, stopOnNextTick = startWorker 12
```

and the work will begin - if we are in F# interactive, we will see the messages printed out in the console every two seconds. We can then run

```
nextValue ()
```

and we will see the messages indicating that value being worked on has moved to the next one.

When it is time to finish working, we can run the

```
stopOnNextTick ()
```

function, which will print out the closing message, then exit.

The `unit` type is important here to signify "no input" - the functions already have all the information they need to work built into them, and the caller is not allowed to change that.

Read The "unit" type online: https://riptutorial.com/fsharp/topic/2513/the--unit--type

Chapter 29: Type and Module Extensions

Remarks

In all cases when extending types and modules, the extending code must be added/loaded before the code that is to call it. It must also be made available to the calling code by opening/importing the relevant namespaces.

Examples

Adding new methods/properties to existing types

F# allows functions to be added as "members" to types when they are defined (for example, Record Types). However F# also allows new instance members to be added to *existing* types - even ones declared elsewhere and in other .net languages.

The following example adds a new instance method `Duplicate` to all instances of `String`.

```
type System.String with
    member this.Duplicate times =
        Array.init times (fun _ -> this)
```

Note: `this` is an arbitrarily chosen variable name to use to refer to the instance of the type that is being extended - `x` would work just as well, but would perhaps be less self-describing.

It can then be called in the following ways.

```
// F#-style call
let result1 = "Hi there!".Duplicate 3

// C#-style call
let result2 = "Hi there!".Duplicate(3)

// Both result in three "Hi there!" strings in an array
```

This functionality is very similar to Extension Methods in C#.

New properties can also be added to existing types in the same way. They will automatically become properties if the new member takes no arguments.

```
type System.String with
    member this.WordCount =
        ' ' // Space character
        |> Array.singleton
        |> fun xs -> this.Split(xs, StringSplitOptions.RemoveEmptyEntries)
        |> Array.length

let result = "This is an example".WordCount
// result is 4
```

Adding new static functions to existing types

F# allow existing types to be extended with new static functions.

```
type System.String with
    static member EqualsCaseInsensitive (a, b) = String.Equals(a, b,
StringComparison.OrdinalIgnoreCase)
```

This new function can be invoked like this:

```
let x = String.EqualsCaseInsensitive("abc", "aBc")
// result is True
```

This feature can mean that rather than having to create "utility" libraries of functions, they can be added to relevant existing types. This can be useful to create more F#-friendly versions of functions that allow features such as currying.

```
type System.String with
    static member AreEqual comparer a b = System.String.Equals(a, b, comparer)

let caseInsensitiveEquals = String.AreEqual StringComparison.OrdinalIgnoreCase

let result = caseInsensitiveEquals "abc" "aBc"
// result is True
```

Adding new functions to existing modules and types using Modules

Modules can be used to add new functions to existing Modules and Types.

```
namespace FSharp.Collections

module List =
    let pair item1 item2 = [ item1; item2 ]
```

The new function can then be called as if it was an original member of List.

```
open FSharp.Collections

module Testing =
    let result = List.pair "a" "b"
    // result is a list containing "a" and "b"
```

Read Type and Module Extensions online: https://riptutorial.com/fsharp/topic/2977/type-and-module-extensions

Chapter 30: Type Providers

Examples

Using the CSV Type Provider

Given the following CSV file:

```
Id,Name
1,"Joel"
2,"Adam"
3,"Ryan"
4,"Matt"
```

You can read the data with the following script:

```
#r "FSharp.Data.dll"
open FSharp.Data

type PeopleDB = CsvProvider<"people.csv">

let people = PeopleDB.Load("people.csv") // this can be a URL

let joel = people.Rows |> Seq.head

printfn "Name: %s, Id: %i" joel.Name joel.Id
```

Using the WMI Type Provider

The WMI type provider allows you to query WMI services with strong typing.

To output the results of a WMI query as JSON,

```
open FSharp.Management
open Newtonsoft.Json

// `Local` is based off of the WMI available at localhost.
type Local = WmiProvider<"localhost">

let data =
    [for d in Local.GetDataContext().Win32_DiskDrive -> d.Name, d.Size]

printfn "%A" (JsonConvert.SerializeObject data)
```

Read Type Providers online: https://riptutorial.com/fsharp/topic/1631/type-providers

Chapter 31: Types

Examples

Introduction to Types

Types can represents various kind of things. It can be a single data, a set of data or a function.

In F#, we can group the types into two categories.:

- F# types:

```
// Functions
let a = fun c -> c

// Tuples
let b = (0, "Foo")

// Unit type
let c = ignore

// Records
type r = { Name : string; Age : int }
let d = { Name = "Foo"; Age = 10 }

// Discriminated Unions
type du = | Foo | Bar
let e = Bar

// List and seq
let f = [ 0..10 ]
let g = seq { 0..10 }

// Aliases
type MyAlias = string
```

- .NET types
    - Built-in type (int, bool, string,...)
    - Classes, Structs & Interfaces
    - Delegates
    - Arrays

Type Abbreviations

Type abbreviations allow you to create aliases on existing types to give them a more meaningful senses.

```
// Name is an alias for a string
type Name = string
```

```
// PhoneNumber is an alias for a string
type PhoneNumber = string
```

Then you can use the alias just as any other type:

```
// Create a record type with the alias
type Contact = {
    Name : Name
    Phone : PhoneNumber }

// Create a record instance
// We can assign a string since Name and PhoneNumber are just aliases on string type
let c = {
    Name = "Foo"
    Phone = "00 000 000" }

printfn "%A" c

// Output
// {Name = "Foo";
// Phone = "00 000 000";}
```

Be careful, aliases does not check for type consistency. This means that two aliases that target the same type can be assigned to each other:

```
let c = {
    Name = "Foo"
    Phone = "00 000 000" }
let d = {
    Name = c.Phone
    Phone = c.Name }

printfn "%A" d

// Output
// {Name = "00 000 000";
// Phone = "Foo";}
```

Types are created in F# using type keyword

`F#` uses the `type` keyword to create different kind of types.

1. Type aliases
2. Discriminated union types
3. Record types
4. Interface types
5. Class types
6. Struct types

Examples with equivalent `C#` code where possible:

```
// Equivalent C#:
//  using IntAliasType = System.Int32;
type IntAliasType = int // As in C# this doesn't create a new type, merely an alias
```

```
type DiscriminatedUnionType =
  | FirstCase
  | SecondCase  of int*string

  member x.SomeProperty = // We can add members to DU:s
    match x with
    | FirstCase         -> 0
    | SecondCase (i, _) -> i

type RecordType =
  {
    Id    : int
    Name  : string
  }
  static member New id name : RecordType = // We can add members to records
    { Id = id; Name = name } // { ... } syntax used to create records

// Equivalent C#:
//  interface InterfaceType
//  {
//    int     Id    { get; }
//    string  Name  { get; }
//    int Increment (int i);
//  }
type InterfaceType =
  interface // In order to create an interface type, can also use [<Interface>] attribute
    abstract member Id        : int
    abstract member Name      : string
    abstract member Increment : int -> int
  end

// Equivalent C#:
//  class ClassType : InterfaceType
//  {
//    static int increment (int i)
//    {
//      return i + 1;
//    }
//
//    public ClassType (int id, string name)
//    {
//      Id    = id    ;
//      Name  = name  ;
//    }
//
//    public int    Id    { get; private set; }
//    public string Name  { get; private set; }
//    public int   Increment (int i)
//    {
//      return increment (i);
//    }
//  }
type ClassType (id : int, name : string) = // a class type requires a primary constructor
  let increment i = i + 1 // Private helper functions

  interface InterfaceType with // Implements InterfaceType
    member x.Id           = id
    member x.Name         = name
    member x.Increment i  = increment i
```

```
// Equivalent C#:
//  class SubClassType : ClassType
//  {
//    public SubClassType (int id, string name) : base(id, name)
//    {
//    }
//  }
type SubClassType (id : int, name : string) =
  inherit ClassType (id, name) // Inherits ClassType

// Equivalent C#:
//  struct StructType
//  {
//    public StructType (int id)
//    {
//      Id = id;
//    }
//
//    public int Id { get; private set; }
//  }
type StructType (id : int) =
  struct  // In order create a struct type, can also use [<Struct>] attribute
    member x.Id = id
  end
```

Type Inference

Acknowledgement

This example is adapted from this article on type inference

What is type Inference?

Type Inference is the mechanism that allows the compiler to deduce what types are used and where. This mechanism is based on an algorithm often called “Hindley-Milner” or “HM”. See below some of the rules for determine the types of simple and function values:

- Look at the literals
- Look at the functions and other values something interacts with
- Look at any explicit type constraints
- If there are no constraints anywhere, automatically generalize to generic types

Look at the literals

The compiler can deduce types by looking at the literals. If the literal is an int and you are adding “x” to it, then “x” must be an int as well. But if the literal is a float and you are adding “x” to it, then “x” must be a float as well.

Here are some examples:

```
let inferInt x = x + 1
let inferFloat x = x + 1.0
let inferDecimal x = x + 1m     // m suffix means decimal
```

```
let inferSByte x = x + 1y       // y suffix means signed byte
let inferChar x = x + 'a'       // a char
let inferString x = x + "my string"
```

Look at the functions and other values it interacts with

If there are no literals anywhere, the compiler tries to work out the types by analysing the functions and other values that they interact with.

```
let inferInt x = x + 1
let inferIndirectInt x = inferInt x       //deduce that x is an int

let inferFloat x = x + 1.0
let inferIndirectFloat x = inferFloat x   //deduce that x is a float

let x = 1
let y = x     //deduce that y is also an int
```

Look at any explicit type constraints or annotations

If there are any explicit type constraints or annotations specified, then the compiler will use them.

```
let inferInt2 (x:int) = x                // Take int as parameter
let inferIndirectInt2 x = inferInt2 x    // Deduce from previous that x is int

let inferFloat2 (x:float) = x                // Take float as parameter
let inferIndirectFloat2 x = inferFloat2 x    // Deduce from previous that x is float
```

Automatic generalization

If after all this, there are no constraints found, the compiler just makes the types generic.

```
let inferGeneric x = x
let inferIndirectGeneric x = inferGeneric x
let inferIndirectGenericAgain x = (inferIndirectGeneric x).ToString()
```

Things that can go wrong with type inference

The type inference isn't perfect, alas. Sometimes the compiler just doesn't have a clue what to do. Again, understanding what is happening will really help you stay calm instead of wanting to kill the compiler. Here are some of the main reasons for type errors:

- Declarations out of order
- Not enough information
- Overloaded methods

Declarations out of order

A basic rule is that you must declare functions before they are used.

This code fails:

```
let square2 x = square x   // fails: square not defined
let square x = x * x
```

But this is ok:

```
let square x = x * x
let square2 x = square x   // square already defined earlier
```

Recursive or simultaneous declarations

A variant of the “out of order” problem occurs with recursive functions or definitions that have to refer to each other. No amount of reordering will help in this case – we need to use additional keywords to help the compiler.

When a function is being compiled, the function identifier is not available to the body. So if you define a simple recursive function, you will get a compiler error. The fix is to add the “rec” keyword as part of the function definition. For example:

```
// the compiler does not know what "fib" means
let fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
   // error FS0039: The value or constructor 'fib' is not defined
```

Here’s the fixed version with “rec fib” added to indicate it is recursive:

```
let rec fib n =              // LET REC rather than LET
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

Not enough information

Sometimes, the compiler just doesn’t have enough information to determine a type. In the following example, the compiler doesn’t know what type the Length method is supposed to work on. But it can’t make it generic either, so it complains.

```
let stringLength s = s.Length
  // error FS0072: Lookup on object of indeterminate type
  // based on information prior to this program point.
  // A type annotation may be needed ...
```

These kinds of error can be fixed with explicit annotations.

```
let stringLength (s:string) = s.Length
```

Overloaded methods

When calling an external class or method in .NET, you will often get errors due to overloading.

In many cases, such as the concat example below, you will have to explicitly annotate the

parameters of the external function so that the compiler knows which overloaded method to call.

```
let concat x = System.String.Concat(x)           //fails
let concat (x:string) = System.String.Concat(x)  //works
let concat x = System.String.Concat(x:string)    //works
```

Sometimes the overloaded methods have different argument names, in which case you can also give the compiler a clue by naming the arguments. Here is an example for the StreamReader constructor.

```
let makeStreamReader x = new System.IO.StreamReader(x)        //fails
let makeStreamReader x = new System.IO.StreamReader(path=x)   //works
```

Read Types online: https://riptutorial.com/fsharp/topic/3559/types

Chapter 32: Units of Measure

Remarks

Units at Runtime

Units of Measure are used only for static checking by the compiler, and are not available at runtime. They cannot be used in reflection or in methods like `ToString`.

For example, C# gives a `double` with no units for a field of type `float<m>` defined by and exposed from an F# library.

Examples

Ensuring Consistent Units in Calculations

Units of measure are additional type annotations that can be added to floats or integers. They can be used to verify at compile time that calculations are using units consistently.

To define annotations:

```
[<Measure>] type m // meters
[<Measure>] type s // seconds
[<Measure>] type accel = m/s^2 // acceleration defined as meters per second squared
```

Once defined, annotations can be used to verify that an expression results in the expected type.

```
// Compile-time checking that this function will return meters, since (m/s^2) * (s^2) -> m
// Therefore we know units were used properly in the calculation.
let freeFallDistance (time:float<s>) : float<m> =
    0.5 * 9.8<accel> * (time*time)

// It is also made explicit at the call site, so we know that the parameter passed should be
in seconds
let dist:float<m> = freeFallDistance 3.0<s>
printfn "%f" dist
```

Conversions between units

```
[<Measure>] type m // meters
[<Measure>] type cm // centimeters

// Conversion factor
let cmInM = 100<cm/m>

let distanceInM = 1<m>
let distanceInCM = distanceInM * cmInM // 100<cm>
```

```
// Conversion function
let cmToM (x : int<cm>) = x / 100<cm/m>
let mToCm (x : int<m>) = x * 100<cm/m>

cmToM 100<cm> // 1<m>
mToCm 1<m> // 100<cm>
```

Note that the F# compiler does not know that `1<m>` equals `100<cm>`. As far as it cares, the units are separate types. You can write similar functions to convert from meters to kilograms, and the compiler would not care.

```
[<Measure>] type kg

// Valid code, invalid physics
let kgToM x = x / 100<kg/m>
```

It is not possible to define units of measure as multiples of other units such as

```
// Invalid code
[<Measure>] type m = 100<cm>
```

However, to define units "per something", for example Hertz, measuring frequency, is simply "per second", is quite simple.

```
// Valid code
[<Measure>] type s
[<Measure>] type Hz = /s

1 / 1<s> = 1 <Hz> // Evaluates to true

[<Measure>] type N = kg m/s // Newtons, measuring force. Note lack of multiplication sign.

// Usage
let mass = 1<kg>
let distance = 1<m>
let time = 1<s>

let force = mass * distance / time // Evaluates to 1<kg m/s>
force = 1<N> // Evaluates to true
```

Using LanguagePrimitives to preserve or set units

When a function doesn't preserve units automatically due to lower-level operations, the `LanguagePrimitives` module can be used to set units on the primitives that support them:

```
/// This cast preserves units, while changing the underlying type
let inline castDoubleToSingle (x : float<'u>) : float32<'u> =
    LanguagePrimitives.Float32WithMeasure (float32 x)
```

To assign units of measure to a double-precision floating-point value, simply multiply by one with correct units:

```
[<Measure>]
type USD

let toMoneyImprecise (amount : float) =
   amount * 1.<USD>
```

To assign units of measure to a unit-less value that isn't System.Double, for example, arriving from a library written in another language, use a conversion:

```
open LanguagePrimitives

let toMoney amount =
   amount |> DecimalWithMeasure<'u>
```

Here are the function types reported by F# interactive:

```
val toMoney : amount:decimal -> decimal<'u>
val toMoneyImprecise : amount:float -> float<USD>
```

Unit-of-measure type parameters

The `[<Measure>]` attribute can be used on type parameters to declare types that are generic with respect to units of measure:

```
type CylinderSize<[<Measure>] 'u> =
    { Radius : float<'u>
      Height : float<'u> }
```

Test usage:

```
open Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols

/// This has type CylinderSize<m>.
let testCylinder =
    { Radius = 14.<m>
      Height =  1.<m> }
```

Use standardized unit types to maintain compatibility

For example, types for SI units have been standardized in the F# core library, in `Microsoft.FSharp.Data.UnitSystems.SI`. Open the appropriate sub-namespace, `UnitNames` or `UnitSymbols`, to use them. Or, if only a few SI units are required, they can be imported with type aliases:

```
/// Seconds, the SI unit of time. Type abbreviation for the Microsoft standardized type.
type [<Measure>] s = Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols.s
```

Some users tend to do the following, which **should not be done** whenever a definition is already available:

```
/// Seconds, the SI unit of time
type [<Measure>] s // DO NOT DO THIS! THIS IS AN EXAMPLE TO EXPLAIN A PROBLEM.
```

The difference becomes apparent when interfacing with other code that refers to the standard SI types. Code that refers to the standard units is compatible, while code that defines its own type is incompatible with any code not using its specific definition.

Therefore, always use the standard types for SI units. It doesn't matter whether you refer to `UnitNames` or `UnitSymbols`, since equivalent names within those two refer to the same type:

```
open Microsoft.FSharp.Data.UnitSystems.SI

/// This is valid, since both versions refer to the same authoritative type.
let validSubtraction = 1.<UnitSymbols.s> - 0.5<UnitNames.second>
```

Read Units of Measure online: https://riptutorial.com/fsharp/topic/1055/units-of-measure

Chapter 33: Using F#, WPF, FsXaml, a Menu, and a Dialog Box

Introduction

The goal here is to build a simple application in F# using the Windows Presentation Foundation (WPF) with traditional menus and dialog boxes. It stems from my frustration in trying to wade through hundreds of sections of documentation, articles and posts dealing with F# and WPF. In order to do anything with WPF, you seem to have to know everything about it. My purpose here is to provide a possible way in, a simple desktop project that can serve as a template for your apps.

Examples

Set up the Project

We'll assume you're doing this in Visual Studio 2015 (VS 2015 Community, in my case). Create an empty Console project in VS. In Project | Properties change the Output Type to Windows Application.

Next, use NuGet to add FsXaml.Wpf to the project; this package was created by the estimable Reed Copsey, Jr., and it greatly simplifies using WPF from F#. On installation, it will add a number of other WPF assemblies, so you will not have to. There are other similar packages to FsXaml, but one of my goals was to keep the number of tools as small as possible in order to make the overall project as simple and maintaiable as possible.

In addition, add UIAutomationTypes as a reference; it comes as part of .NET.

Add the "Business Logic"

Presumably, your program will do something. Add your working code to the project in place of Program.fs. In this case, our task is to draw spirograph curves on a Window Canvas. This is accomplished using Spirograph.fs, below.

```
namespace Spirograph

// open System.Windows does not automatically open all its sub-modules, so we
// have to open them explicitly as below, to get the resources noted for each.
open System                             // for Math.PI
open System.Windows                     // for Point
open System.Windows.Controls            // for Canvas
open System.Windows.Shapes              // for Ellipse
open System.Windows.Media               // for Brushes

// ------------------------------------------------------------------------------
// This file is first in the build sequence, so types should be defined here
type DialogBoxXaml  = FsXaml.XAML<"DialogBox.xaml">
type MainWindowXaml = FsXaml.XAML<"MainWindow.xaml">
```

```
type App            = FsXaml.XAML<"App.xaml">

// ----------------------------------------------------------------------------
// Model: This draws the Spirograph
type MColor = | MBlue   | MRed | MRandom

type Model() =
  let mutable myCanvas: Canvas = null
  let mutable myR              = 220    // outer circle radius
  let mutable myr              = 65     // inner circle radius
  let mutable myl              = 0.8    // pen position relative to inner circle
  let mutable myColor          = MBlue  // pen color

  let rng                      = new Random()
  let mutable myRandomColor    = Color.FromRgb(rng.Next(0, 255) |> byte,
                                               rng.Next(0, 255) |> byte,
                                               rng.Next(0, 255) |> byte)

  member this.MyCanvas
    with get() = myCanvas
    and  set(newCanvas) = myCanvas <- newCanvas

  member this.MyR
    with get() = myR
    and  set(newR) = myR <- newR

  member this.Myr
    with get() = myr
    and  set(newr) = myr <- newr

  member this.Myl
    with get() = myl
    and  set(newl) = myl <- newl

  member this.MyColor
    with get() = myColor
    and  set(newColor) = myColor <- newColor

  member this.Randomize =
    // Here we randomize the parameters. You can play with the possible ranges of
    // the parameters to find randomized spirographs that are pleasing to you.
    this.MyR      <- rng.Next(100, 500)
    this.Myr      <- rng.Next(this.MyR / 10, (9 * this.MyR) / 10)
    this.Myl      <- 0.1 + 0.8 * rng.NextDouble()
    this.MyColor  <- MRandom
    myRandomColor <- Color.FromRgb(rng.Next(0, 255) |> byte,
                                   rng.Next(0, 255) |> byte,
                                   rng.Next(0, 255) |> byte)

  member this.DrawSpirograph =
    // Draw a spirograph. Note there is some fussing with ints and floats; this
    // is required because the outer and inner circle radii are integers. This is
    // necessary in order for the spirograph to return to its starting point
    // after a certain number of revolutions of the outer circle.

    // Start with usual recursive gcd function and determine the gcd of the inner
    // and outer circle radii. Everything here should be in integers.
    let rec gcd x y =
        if y = 0 then x
        else gcd y (x % y)
```

```
    let g = gcd this.MyR this.Myr             // find greatest common divisor
    let maxRev = this.Myr / g                 // maximum revs to repeat

    // Determine width and height of window, location of center point, scaling
    // factor so that spirograph fits within the window, ratio of inner and outer
    // radii.

    // Everything from this point down should be float.
    let width, height = myCanvas.ActualWidth, myCanvas.ActualHeight
    let cx, cy = width / 2.0, height / 2.0    // coordinates of center point
    let maxR   = min cx cy                    // maximum radius of outer circle
    let scale  = maxR / float(this.MyR)             // scaling factor
    let rRatio = float(this.Myr) / float(this.MyR)  // ratio of the radii

    // Build the collection of spirograph points, scaled to the window.
    let points = new PointCollection()
    for degrees in [0 .. 5 .. 360 * maxRev] do
      let angle = float(degrees) * Math.PI / 180.0
      let x, y = cx + scale * float(this.MyR) *
                 ((1.0-rRatio)*Math.Cos(angle) +
                  this.Myl*rRatio*Math.Cos((1.0-rRatio)*angle/rRatio)),
                 cy + scale * float(this.MyR) *
                 ((1.0-rRatio)*Math.Sin(angle) -
                  this.Myl*rRatio*Math.Sin((1.0-rRatio)*angle/rRatio))
      points.Add(new Point(x, y))

    // Create the Polyline with the above PointCollection, erase the Canvas, and
    // add the Polyline to the Canvas Children
    let brush = match this.MyColor with
                | MBlue   -> Brushes.Blue
                | MRed    -> Brushes.Red
                | MRandom -> new SolidColorBrush(myRandomColor)

    let mySpirograph = new Polyline()
    mySpirograph.Points <- points
    mySpirograph.Stroke <- brush

    myCanvas.Children.Clear()
    this.MyCanvas.Children.Add(mySpirograph) |> ignore
```

Spirograph.fs is the first F# file in the compilation order, so it contains the definitions of the types we will need. Its job is to draw a spirograph on the main window Canvas based on parameters entered in a dialog box. Since there are lots of references on how to draw a spirograph, we won't go into that here.

Create the main window in XAML

You have to create a XAML file that defines the main window that contains our menu and drawing space. Here's the XAML code in MainWindow.xaml:

```
<!-- This defines the main window, with a menu and a canvas. Note that the Height
     and Width are overridden in code to be 2/3 the dimensions of the screen -->
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Spirograph" Height="200" Width="300">
    <!-- Define a grid with 3 rows: Title bar, menu bar, and canvas. By default
```

```
         there is only one column -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <!-- Define the menu entries -->
        <Menu  Grid.Row="0">
            <MenuItem Header="File">
                <MenuItem Header="Exit"
                          Name="menuExit"/>
            </MenuItem>
            <MenuItem Header="Spirograph">
                <MenuItem Header="Parameters..."
                          Name="menuParameters"/>
                <MenuItem Header="Draw"
                          Name="menuDraw"/>
            </MenuItem>
            <MenuItem Header="Help">
                <MenuItem Header="About"
                          Name="menuAbout"/>
            </MenuItem>
        </Menu>
        <!-- This is a canvas for drawing on. If you don't specify the coordinates
             for Left and Top you will get NaN for those values -->
        <Canvas Grid.Row="1" Name="myCanvas" Left="0" Top="0">
        </Canvas>
    </Grid>
</Window>
```

Comments are usually not included in XAML files, which I think is a mistake. I've added some comments to all the XAML files in this project. I don't assert they are the best comments ever written, but they at least show how a comment should be formatted. Note that nested comments are not allowed in XAML.

Create the dialog box in XAML and F#

The XAML file for the spirograph parameters is below. It includes three text boxes for the spirograph parameters and a group of three radio buttons for color. When we give radio buttons the same group name - as we have here - WPF handles the on/off switching when one is selected.

```
<!-- This first part is boilerplate, except for the title, height and width.
     Note that some fussing with alignment and margins may be required to get
     the box looking the way you want it. -->
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Parameters" Height="200" Width="250">
    <!-- Here we define a layout of 3 rows and 2 columns below the title bar -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <!-- Define a label and a text box for the first three rows. Top row is
             the integer radius of the outer circle -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="0"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="R: Outer"
                   Height="24" Width='65'/>
            <TextBox x:Name="radiusR"  Margin="0,0,0,0.5" Width="120"
                     VerticalAlignment="Bottom" Height="20">Integer</TextBox>
        </StackPanel>
        <!-- This defines a label and text box for the integer radius of the
             inner circle -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="1"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="r: Inner"
                   Height="24" Width='65'/>
            <TextBox x:Name="radiusr"  Margin="0,0,0,0.5" Width="120"
                     VerticalAlignment="Bottom" Height="20" Text="Integer"/>
        </StackPanel>
        <!-- This defines a label and text box for the float ratio of the inner
             circle radius at which the pen is positioned -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="2"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,0,1" Content="l: Ratio"
                   Height="24" Width='65'/>
            <TextBox x:Name="ratiol"  Margin="0,0,0,1" Width="120"
                     VerticalAlignment="Bottom" Height="20" Text="Float"/>
        </StackPanel>
        <!-- This defines a radio button group to select color -->
        <StackPanel Orientation="Horizontal" Grid.Column="0" Grid.Row="3"
                    Grid.ColumnSpan="2">
            <Label VerticalAlignment="Top" Margin="5,6,4,5.333" Content="Color"
                   Height="24"/>
            <RadioButton x:Name="buttonBlue" Content="Blue" GroupName="Color"
                         HorizontalAlignment="Left"  VerticalAlignment="Top"
                         Click="buttonBlueClick"
                         Margin="5,13,11,3.5" Height="17"/>
            <RadioButton x:Name="buttonRed"  Content="Red"  GroupName="Color"
                         HorizontalAlignment="Left" VerticalAlignment="Top"
                         Click="buttonRedClick"
                         Margin="5,13,5,3.5" Height="17" />
            <RadioButton x:Name="buttonRandom"  Content="Random"
                         GroupName="Color" Click="buttonRandomClick"
                         HorizontalAlignment="Left" VerticalAlignment="Top"
                         Margin="5,13,5,3.5" Height="17" />
        </StackPanel>
        <!-- These are the standard OK/Cancel buttons -->
        <Button Grid.Row="4" Grid.Column="0" Name="okButton"
                Click="okButton_Click" IsDefault="True">OK</Button>
        <Button Grid.Row="4" Grid.Column="1" Name="cancelButton"
                IsCancel="True">Cancel</Button>
    </Grid>
</Window>
```

Now we add the code behind for the Dialog.Box. By convention, the code used to handle the interface of the dialog box with the rest of the program is named XXX.xaml.fs, where the associated XAML file is named XXX.xaml.

```
namespace Spirograph

open System.Windows.Controls

type DialogBox(app: App, model: Model, win: MainWindowXaml) as this =
  inherit DialogBoxXaml()

  let myApp   = app
  let myModel = model
  let myWin   = win

  // These are the default parameters for the spirograph, changed by this dialog
  // box
  let mutable myR = 220                // outer circle radius
  let mutable myr = 65                 // inner circle radius
  let mutable myl = 0.8                // pen position relative to inner circle
  let mutable myColor = MBlue          // pen color

  // These are the dialog box controls. They are initialized when the dialog box
  // is loaded in the whenLoaded function below.
  let mutable RBox: TextBox = null
  let mutable rBox: TextBox = null
  let mutable lBox: TextBox = null

  let mutable blueButton: RadioButton   = null
  let mutable redButton: RadioButton    = null
  let mutable randomButton: RadioButton = null

  // Call this functions to enable or disable parameter input depending on the
  // state of the randomButton. This is a () -> () function to keep it from
  // being executed before we have loaded the dialog box below and found the
  // values of TextBoxes and RadioButtons.
  let enableParameterFields(b: bool) =
    RBox.IsEnabled <- b
    rBox.IsEnabled <- b
    lBox.IsEnabled <- b

  let whenLoaded _ =
    // Load and initialize text boxes and radio buttons to the current values in
    // the model. These are changed only if the OK button is clicked, which is
    // handled below. Also, if the color is Random, we disable the parameter
    // fields.
    RBox <- this.FindName("radiusR") :?> TextBox
    rBox <- this.FindName("radiusr") :?> TextBox
    lBox <- this.FindName("ratiol")  :?> TextBox

    blueButton   <- this.FindName("buttonBlue")   :?> RadioButton
    redButton    <- this.FindName("buttonRed")    :?> RadioButton
    randomButton <- this.FindName("buttonRandom") :?> RadioButton

    RBox.Text <- myModel.MyR.ToString()
    rBox.Text <- myModel.Myr.ToString()
    lBox.Text <- myModel.Myl.ToString()

    myR <- myModel.MyR
    myr <- myModel.Myr
    myl <- myModel.Myl

    blueButton.IsChecked   <- new System.Nullable<bool>(myModel.MyColor = MBlue)
    redButton.IsChecked    <- new System.Nullable<bool>(myModel.MyColor = MRed)
    randomButton.IsChecked <- new System.Nullable<bool>(myModel.MyColor = MRandom)
```

```
  myColor <- myModel.MyColor
  enableParameterFields(not (myColor = MRandom))

let whenClosing _ =
  // Show the actual spirograph parameters in a message box at close. Note the
  // \n in the sprintf gives us a linebreak in the MessageBox. This is mainly
  // for debugging, and it can be deleted.
  let s = sprintf "R = %A\nr = %A\nl = %A\nColor = %A"
                  myModel.MyR myModel.Myr myModel.Myl myModel.MyColor
  System.Windows.MessageBox.Show(s, "Spirograph") |> ignore
  ()

let whenClosed _ =
  ()

do
  this.Loaded.Add whenLoaded
  this.Closing.Add whenClosing
  this.Closed.Add whenClosed

override this.buttonBlueClick(sender: obj,
                              eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MBlue
  enableParameterFields(true)
  ()

override this.buttonRedClick(sender: obj,
                             eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MRed
  enableParameterFields(true)
  ()

override this.buttonRandomClick(sender: obj,
                                eArgs: System.Windows.RoutedEventArgs) =
  myColor <- MRandom
  enableParameterFields(false)
  ()

override this.okButton_Click(sender: obj,
                             eArgs: System.Windows.RoutedEventArgs) =
  // Only change the spirograph parameters in the model if we hit OK in the
  // dialog box.
  if myColor = MRandom
  then myModel.Randomize
  else myR <- RBox.Text |> int
       myr <- rBox.Text |> int
       myl <- lBox.Text |> float

       myModel.MyR   <- myR
       myModel.Myr   <- myr
       myModel.Myl   <- myl
       model.MyColor <- myColor

  // Note that setting the DialogResult to nullable true is essential to get
  // the OK button to work.
  this.DialogResult <- new System.Nullable<bool> true
  ()
```

Much of the code here is devoted to ensuring that the spirograph parameters in Spirograph.fs

match those shown in this dialog box. Note that there is no error checking: If you enter a floating point for the integers expected in the top two parameter fields, the program will crash. So, please add error checking in your own effort.

Note also that the parameter input fields are disabled with Random color is picked in the radio buttons. It's here just to show how it can be done.

In order to move data back and forth between the dialog box and the program I use the System.Windows.Element.FindName() to find the appropriate control, cast it to the control it should be, and then get the relevant settings from the Control. Most other example programs use data binding. I did not for two reasons: First, I couldn't figure out how to make it work, and second, when it didn't work I got no error message of any kind. Maybe someone who visits this on StackOverflow can tell me how to use data binding without including a whole new set of NuGet packages.

Add the code behind for MainWindow.xaml

```
namespace Spirograph

type MainWindow(app: App, model: Model) as this =
  inherit MainWindowXaml()

  let myApp   = app
  let myModel = model

  let whenLoaded _ =
    ()

  let whenClosing _ =
    ()

  let whenClosed _ =
    ()

  let menuExitHandler _ =
    System.Windows.MessageBox.Show("Good-bye", "Spirograph") |> ignore
    myApp.Shutdown()
    ()

  let menuParametersHandler _ =
    let myParametersDialog = new DialogBox(myApp, myModel, this)
    myParametersDialog.Topmost <- true
    let bResult = myParametersDialog.ShowDialog()
    myModel.DrawSpirograph
    ()

  let menuDrawHandler _ =
    if myModel.MyColor = MRandom then myModel.Randomize
    myModel.DrawSpirograph
    ()

  let menuAboutHandler _ =
    System.Windows.MessageBox.Show("F#/WPF Menus & Dialogs", "Spirograph")
    |> ignore
    ()
```

```
  do
    this.Loaded.Add whenLoaded
    this.Closing.Add whenClosing
    this.Closed.Add whenClosed
    this.menuExit.Click.Add menuExitHandler
    this.menuParameters.Click.Add menuParametersHandler
    this.menuDraw.Click.Add menuDrawHandler
    this.menuAbout.Click.Add menuAboutHandler
```

There's not a lot going on here: We open the Parameters dialog box when required and we have the option of redrawing the spirograph with whatever the current parameters are.

Add the App.xaml and App.xaml.fs to tie everything together

```
<!-- All boilerplate for now -->
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Here's the code behind:

```
namespace Spirograph

open System
open System.Windows
open System.Windows.Controls

module Main =
  [<STAThread; EntryPoint>]
  let main _ =
    // Create the app and the model with the "business logic", then create the
    // main window and link its Canvas to the model so the model can access it.
    // The main window is linked to the app in the Run() command in the last line.
    let app = App()
    let model = new Model()
    let mainWindow = new MainWindow(app, model)
    model.MyCanvas <- (mainWindow.FindName("myCanvas") :?> Canvas)

    // Make sure the window is on top, and set its size to 2/3 of the dimensions
    // of the screen.
    mainWindow.Topmost <- true
    mainWindow.Height  <-
      (System.Windows.SystemParameters.PrimaryScreenHeight * 0.67)
    mainWindow.Width   <-
      (System.Windows.SystemParameters.PrimaryScreenWidth * 0.67)

    app.Run(mainWindow) // Returns application's exit code.
```

App.xaml is all boilerplate here, mainly to show where application resources, such as icons, graphics, or external files - can be declared. The companion App.xaml.fs pulls together the Model and the MainWindow, sizes the MainWindow to two-thirds of the available screen size, and runs it.

When you build this, remember to make sure that the Build property for each xaml file is set to

Resource. Then you can either run through the debugger or compile to an exe file. Note that you cannot run this using the F# interpreter: The FsXaml package and the interpreter are incompatible.

There you have it. I hope you can use this as a starting point for your own applications, and in doing so you can extend your knowlege beyond what is shown here. Any comments and suggestions will be appreciated.

Read Using F#, WPF, FsXaml, a Menu, and a Dialog Box online:
https://riptutorial.com/fsharp/topic/9145/using-fsharp--wpf--fsxaml--a-menu--and-a-dialog-box

Credits

S. No	Chapters	Contributors
1	Getting started with F#	Anonymous, Boggin, Brett Jackson, Community, FireAlkazar, goric, Joel Martinez, Jono Job, Matas Vaitkevicius, Ringil, rmunn
2	1 : F# WPF Code Behind Application with FsXaml	Bent Tranberg
3	Active Patterns	Erik Schierboom, FuleSnabel, goric, Honza Brestan, Julien Pires, Ringil
4	Classes	asibahi, inzi, RamenChef, Tomasz Maczyński
5	Design pattern implementation in F#	FuleSnabel, Ringil
6	Discriminated Unions	chillitom, Erik Schierboom, Estanislau Trepat, gdziadkiewicz, goric, GregC, James McCalden, Joel Martinez, Martin4ndersen, Vandroiy, VillasV
7	F# on .NET Core	Boggin, Joel Martinez
8	F# Performance Tips and Tricks	FuleSnabel, Paul Westcott, Ringil, s952163
9	Folds	Jean-Claude Colette, Zaid Ajaj
10	Functions	asibahi, Julien Pires, rmunn, ronilk
11	Generics	Jake Lishman
12	Introduction to WPF in F#	Funk
13	Lazy Evaluation	inzi
14	Lists	asibahi, Jean-Claude Colette, Ringil, Zaid Ajaj
15	Mailbox Processor	Honza Brestan
16	Memoization	Jean-Claude Colette, Julien Pires, Ringil
17	Monads	FuleSnabel

18	Operators	FuleSnabel
19	Option types	asibahi, chillitom, FuleSnabel
20	Pattern Matching	asibahi, James McCalden, Jono Job, Ringil, rmunn, t3dodson, Tormod Haugene
21	Porting C# to F#	jdphenix, marklam, RamenChef
22	Records	eirik, goric, Ringil
23	Reflection	FuleSnabel
24	Sequence	Foggy Finder, inzi, James McCalden, Julien Pires, s952163
25	Sequence Workflows	Jwosty
26	Statically Resolved Type Parameters	Maslow
27	Strings	FireAlkazar, Julien Pires
28	The "unit" type	4444, Abel, Jake Lishman, rmunn
29	Type and Module Extensions	Jono Job
30	Type Providers	GregC, jdphenix, Joel Martinez
31	Types	Cedric Royer-Bertrand, FuleSnabel, Julien Pires
32	Units of Measure	asibahi, goric, GregC, Vandroiy
33	Using F#, WPF, FsXaml, a Menu, and a Dialog Box	Bob McCrory, Goswin