F#

Succinct, Expressive, Functional

Don Syme,
Principal Researcher
Microsoft Research, Cambridge

Topics

- What is F# about?
- Some Simple F# Programming
- A Taste of Parallel/Reactive with F#

What is F# about?

Or: *Why is Microsoft investing in functional programming anyway?*

Simplicity

Economics

Fun, Fun and More Fun!

Simplicity

Code!

```
//F#
open System
let a = 2
Console.WriteLine a
```

```
//C#
using System;

namespace ConsoleApplication1
{
  class Program
  {
    static int a()
    {
        return 2;
    }
    static void Main(string[] args)
    {
        Console.WriteLine(a);
    }
  }
}
```

More Noise Than Signal!

Pleasure

```
type Command = Command of (Rover -> unit)

let BreakCommand     =
    Command(fun rover -> rover.Accelerate(-1.0))

let TurnLeftCommand  =
    Command(fun rover -> rover.Rotate(-5.0<degs>))
```

Pain

```
abstract class Command
 {
     public virtual void Execute();
 }
 abstract class MarsRoverCommand : Command
 {
     protected MarsRover Rover { get; priva

     public MarsRoverCommand(MarsRover rove
     {
         this.Rover = rover;
     }
 }
 class BreakCommand : MarsRoverCommand
 {
     public BreakCommand(MarsRover rover)
         : base(rover)
     {
     }
     public override void Execute()
     {
         Rover.Rotate(-5.0);
     }
 }
class TurnLeftCommand : MarsRoverCommand
 {
     public TurnLeftCommand(MarsRover rover
```

Pleasure

```
let swap (x, y) = (y, x)
```

```
let rotations (x, y, z) =
    [ (x, y, z);
      (z, x, y);
      (y, z, x) ]
```

```
let reduce f (x, y, z) =
    f x + f y + f z
```

Pain

```
Tuple<U,T> Swap<T,U>(Tuple<T,U> t)
{
    return new Tuple<U,T>(t.Item2, t.Item1)
}
```

```
ReadOnlyCollection<Tuple<T,T,T>>
   Rotations<T>(Tuple<T,T,T> t)
{
  new ReadOnlyCollection<int>
   (new Tuple<T,T,T>[]
     { new Tuple<T,T,T>(t.Item1,t.Item2,t.Item3)
       new Tuple<T,T,T>(t.Item3,t.Item1,t.Item2)
       new Tuple<T,T,T>(t.Item2,t.Item3,t.Item1)
   });
}
```

```
int Reduce<T>(Func<T,int> f,Tuple<T,T,T> t)
{
    return f(t.Item1) + f(t.Item2) + f (t.Item3)
}
```

Pleasure

```
type Expr =
    | True
    | And of Expr * Expr
    | Nand of Expr * Expr
    | Or of Expr * Expr
    | Xor of Expr * Expr
    | Not of Expr
```

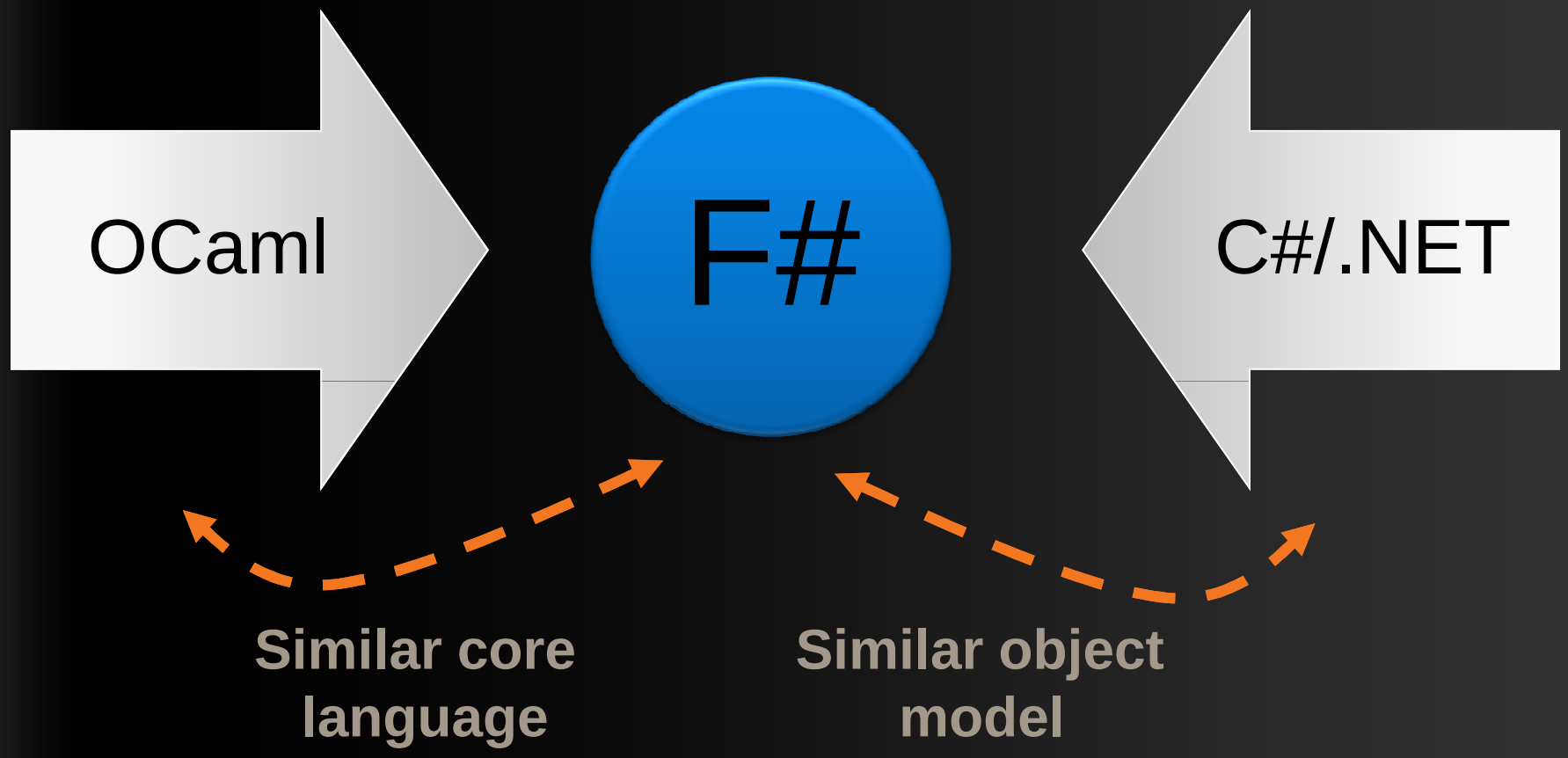
Pain

```
public abstract class Expr { }
public abstract class UnaryOp :Expr
{
    public Expr First { get; private set; }

    public UnaryOp(Expr first)
    {
        this.First = first;
    }
}

public abstract class BinExpr : Expr
{
    public Expr First { get; private set; }

    public Expr Second { get; private set;
   }

    public BinExpr(Expr first, Expr second)

    {
        this.First = first;
        this.Second = second;
    }
```

You
Can
Interoperate
With
Everything

People
Love
Programming
in
F#

F#: Influences



F#: Combining Paradigms

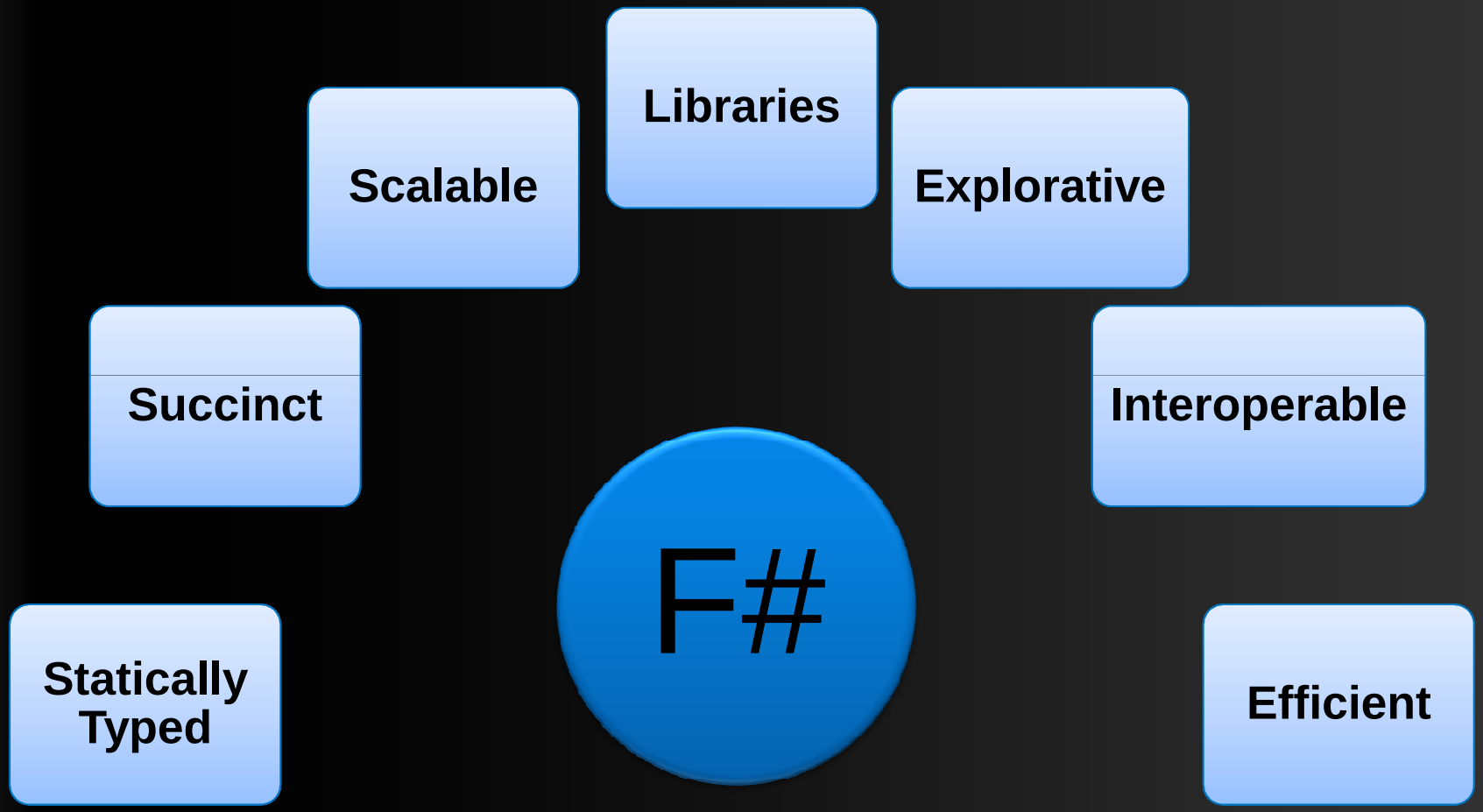
I've been coding in F# lately, for a production task.

*F# allows you to **move smoothly** in your programming style... I start with pure functional code, shift slightly towards an object-oriented style, and in production code, I sometimes have to do some imperative programming.*

*I can **start with a pure idea**, and still **finish my project with realistic code**. You're never disappointed in any phase of the project!*

Julien Laugel, Chief Software Architect, www.eurostocks.com

F#: The Combination Counts!



Let’s WebCrawl...

Orthogonal & Unified Constructs

- Let “let” simplify your life…

Type inference. The **safety** of C# with the **succinctness** of a scripting language

Bind a static value

Bind a static function

Bind a local value

Bind a local function

```
let data = (1, 2, 3)

let f (a, b, c) =
    let sum = a + b + c
    let g x = sum + x*x
    (g a, g b, g c)
```

Fundamentals - Whitespace Matters

```
let computeDeriative f x =
    let p1 = f (x - 0.05)

  let p2 = f (x + 0.05)

      (p2 – p1) / 0.1
```

Offside (bad indentation)

Fundamentals - Whitespace Matters

```
let computeDeriative f x =
    let p1 = f (x - 0.05)

    let p2 = f (x + 0.05)

    (p2 – p1) / 0.1
```

Orthogonal & Unified Constructs

- Functions: like delegates + unified and simple

```
(fun x -> x + 1

let f x = x + 1

(f, f)

val f : int -> int
```

One simple mechanism, many uses

Declare a funct

A pair of functio

A function type

```
predicate = 'T -> bool

send = 'T -> unit

threadStart = unit -> unit

comparer = 'T -> 'T -> int

hasher = 'T -> int

equality = 'T -> 'T -> bool
```

Functional– Pipelines

The pipeline operator

```
x |> f
```

Functional– Pipelines

Successive stages in a pipeline

```
x |> f1
  |> f2
  |> f3
```

Immutability the norm…

```
//-----------------------------------
// Part 1. Adjust some constants

let PI = 3.141592654

PI <- 4.0
```

This value is not

Error List

1 Error | Warnings | 0

	Descripti
1	This value ot mutable.

Values may not be changed

```
type Person =
    { Name : string;
      Birth: DateTime }

let bob =
    { Name = "bob";
      Birth = DateTime(15,8,1980) }

// OK
let bobJunior =
    { bob with Birth = DateTime(23,5,2006) }

// Not OK!
bob.Birth <- DateTime(23,5,2006)
```

Data is immutable by default

✓ Copy & Update

✘ Not Mutate

Error List

1 Error | 0 Warnings

	Description	File	Line	Column
1	error FS0005: This field is not mutable	test.fs	18	1

In Praise of Immutability

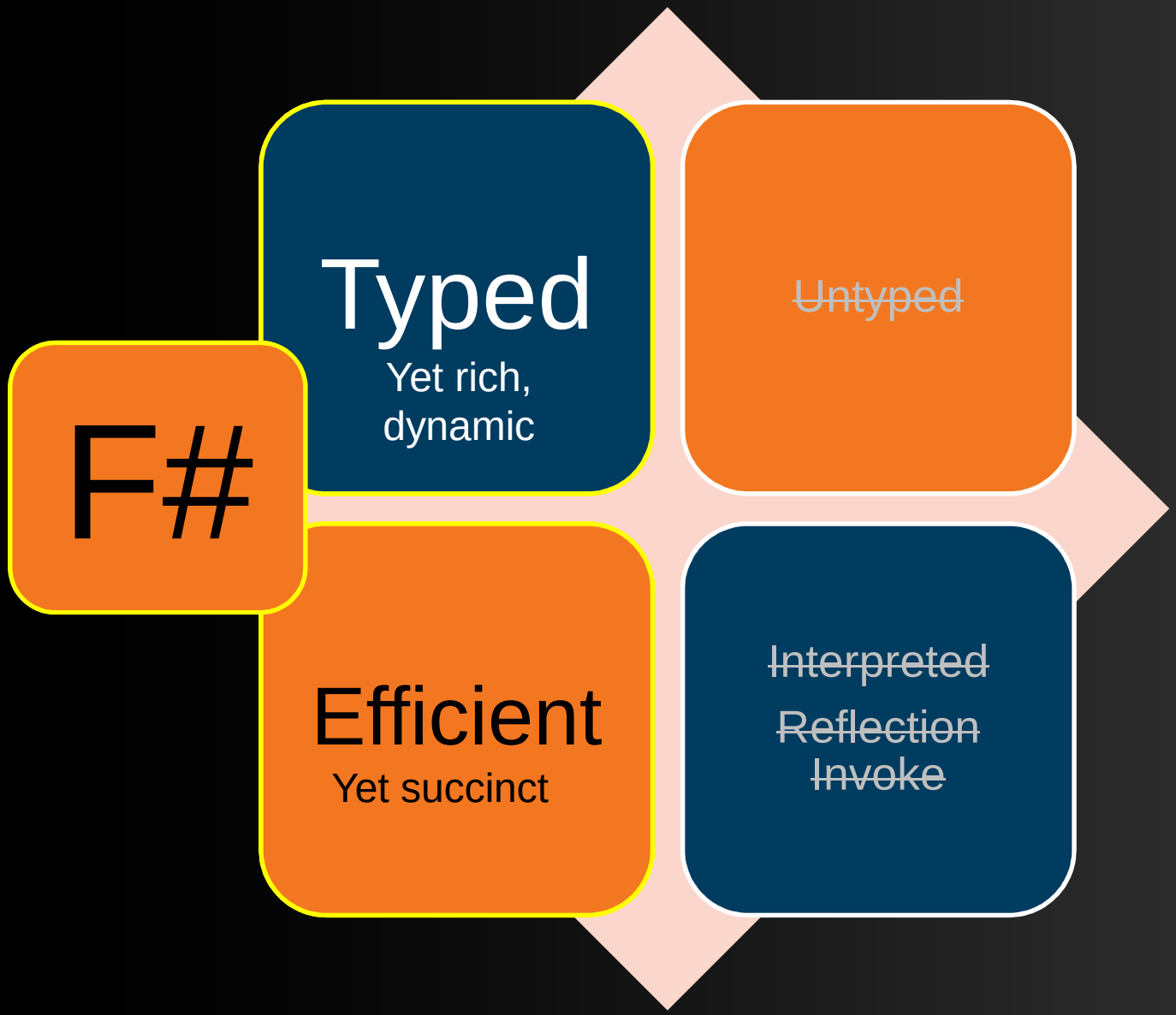
- Immutable objects can be relied upon
- Immutable objects can transfer between threads
- Immutable objects can be aliased safely
- Immutable objects lead to (different) optimization opportunities

Weakly Typed? Slow?

```
//F#
#light
open System
let a = 2
Console.WriteLine(a)
```

```
//C#
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static int a()
        {
            return 2;
        }

        static void Main(string[]

            Console.WriteLine(a);
        }
    }
}
```

Looks Weakly typed?
Maybe Dynamic?


F#
Typed
Yet rich, dynamic
~~Untyped~~
Efficient
Yet succinct
~~Interpreted~~
~~Reflection~~
~~Invoke~~

F# Objects

F# - Objects + Functional

```
type Vector2D (dx:double, dy:double) =

    member v.DX = dx

    member v.DY = dy

    member v.Length = sqrt (dx*dx+dy*dy)

    member v.Scale (k) = Vector2D (dx*k,dy*k)
```

Inputs to object construction

Exported properties

Exported method

F# - Objects + Functional

```
type Vector2D(dx:double,dy:double) =

    let norm2 = dx*dx+dy*dy

    member v.DX = dx

    member v.DY = dy

    member v.Length = sqrt(norm2)

    member v.Norm2 = norm2
```

Internal (pre-computed) values and functions

F# - Objects + Functional

```
type HuffmanEncoding(freq:seq<char*int>) =

   ...
   < 50 lines of beautiful functional code>
   ...

    member x.Encode(input: seq<char>) =
          encode(input)

    member x.Decode(input: seq<char>) =
          decode(input)
```

Immutable inputs

Internal tables

Publish access

F# - Objects + Functional

```
type Vector2D(dx:double,dy:double) =

    let mutable currDX = dx

    let mutable currDX = dy

    member v.DX = currDX

    member v.DY = currDY

    member v.Move(x,y) =
          currDX <- currDX+x
          currDY <- currDY+y
```

Internal state

Publish internal state

Mutate internal state

Interlude: Case Study

The Scale of Things

- **Weeks of data in training**:

  N,000,000,000 impressions, 6TB data
- **2 weeks of CPU time during training**:

  2 wks × 7 days × 86,400 sec/day =

  1,209,600 seconds
- **Learning algorithm speed requirement**:
  - **N,000** impression updates / sec
  - **N00.0 μs** per impression update

F# and adCenter

- 4 week project, 4 machine learning experts
- 100million probabilistic variables
- Processes 6TB of training data
- Real time processing

AdPredict: What We O

- Quick Coding
- Agile Coding
- Scripting
- Performance
- Memory-Faithful
- Succinct
- Symbolic
- .NET Integration

F#'s powerful type inference means less typing, more thinking

Type-inferred code is easily refactored

"Hands-on" exploration.

Immediate scaling to massive data sets

mega-data structures, 16GB machines

Live in the **domain**, not the language

Schema compilation and "Schedules"

Especially Excel, SQL Server

Smooth Transitions

- Researcher's Brain → Realistic, Efficient Code
- Realistic, Efficient Code → Component
- Component → Deployment

F# Async/Parallel

F# is a **Parallel** Language

(Multiple active **computations**)

F# is a **Reactive** Language

(Multiple pending **reactions**)

e.g.
GUI Event
Page Load
Timer Callback
Query Response
HTTP Response
Web Service Response
Disk I/O Completion
Agent Gets Message

async { ... }

A Building Block for Writing Reactive Code

```
async { ... }
```

- For users:

You can run it, but it may take a while

Or, your builder says...

OK, I can do the job, but I might have to talk to someone else about it. I'll get back to you when I'm done

async { ... }

```
async { let! image = ReadAsync "cat.jpg"
        let image2 = f image
        do! WriteAsync image2 "dog.jpg"
        do printfn "done!"
        return image2 }
```

Asynchronous "non-blocking" action

Continuation/ Event callback

You're actually writing this (approximately):

```
async.Delay(fun () ->
    async.Bind(ReadAsync "cat.jpg", (fun image ->
        let image2 = f image
        async.Bind(writeAsync "dog.jpg",(fun () ->
            printfn "done!"
            async.Return()))))))
```

Code: Web Translation

Typical F# Reactive Architecture

Single Threaded GUI

Or

Single Threaded Page Handler

Or

Command Line Driver

```
Async.Start (async { ... }
```

```
Async.Parallel [ ... ]
```

...

(queued)

```
new Agent<_>(async { ... })
```

```
new WebCrawler<_>()
    Internally: new Agent<_>(...) ...

    event x.Started
    event x.CrawledPage
    event x.Finished
```

Taming Asynchronous I/O

```
using System;
using System.IO;
using System.Threading;

public class BulkImageProcAsync
{
    public const String ImageBaseNam
    public const int numImages = 200
    public const int numPixels = 512

    // ProcessImage has a simple O(N
    // of times you repeat that loop
    // bound or more IO-bound.
    public static int processImageRe

    // Threads must decrement NumIma
    // their access to it through a
    public static int NumImagesToFin
    public static Object[] NumImages
    // WaitObject is signalled when
    public static Object[] WaitObjec
    public class ImageStateObject
    {
        public byte[] pixels;
        public int imageNum;
        [
    }
```

```
public static void ReadInImageCallback(IAsyncResult as
{
    ImageStateObject state = (ImageStateObject)asyncRe
    Stream stream = state.fs;
    int bytesRead = stream.EndRead(asyncResult);
    if (bytesRead != numPixels)
        throw new Exception(String.Format
            ("In ReadInImageCallback, got the wrong nu
            "bytes from the image: {0}.", bytesRead));
    ProcessImage(state.pixels, state.imageNum);
    stream.Close();

    // Now write out the image.
    // Using asynchronous I/O here appears not to be b
    // It ends up swamping the threadpool, because the
    // threads are blocked on I/O requests that were j
    // the threadpool.
    FileStream fs = new FileStream(ImageBaseName + sta
        ".done", FileMode.Create, FileAccess.Write, Fi
        4096, false);
    fs.Write(state.pixels, 0, numPixels);
    fs.Close();

                                                  much memory.
                                                  ible is a good

                                                  now.

    }
}
```

```
let ProcessImageAsync () =
    async { let  inStream  = File.OpenRead(sprintf "Image%d.tmp" i)
            let! pixels    = inStream.ReadAsync(numPixels)
            let  pixels'   = TransformImage(pixels,i)
            let  outStream = File.OpenWrite(sprintf "Image%d.done" i)
            do!  outStream.WriteAsync(pixels')
            do   Console.WriteLine "done!"  }

let ProcessImagesAsyncWorkflow() =
    Async.Run (Async.Parallel
                [ for i in 1 .. numImages -> ProcessImageAsync i ])
```

```
    public static void ProcessImagesInBulk()
    {
        Console.WriteLine("Processing images...  ");
        long t0 = Environment.TickCount;
        NumImagesToFinish = numImages;
        AsyncCallback readImageCallback = new
            AsyncCallback(ReadInImageCallback);
        for (int i = 0; i < numImages; i++)
        {
            ImageStateObject state = new ImageStateObject();
            state.pixels = new byte[numPixels];
            state.imageNum = i;
            // Very large items are read only once, so you can make the
            // buffer on the FileStream very small to save memory.
            FileStream fs = new FileStream(ImageBaseName + i + ".tmp",
                FileMode.Open, FileAccess.Read, FileShare.Read, 1, true);
            state.fs = fs;
            fs.BeginRead(state.pixels, 0, numPixels, readImageCallback,
                state);
        }

        // Determine whether all images are done being processed.
        // If
        bool mustBlo
        lock (NumImagesMute
        {
            if (NumImagesToFinish > 0)
                mustBlock = true;
        }
        if (mustBlock)
        {
            Console.WriteLine("All worke
                " Blocking until they complete. numLeft: {0}",
                NumImagesToFinish);
            Monitor.Enter(WaitObject);
            Monitor.Wait(WaitObject);
            Monitor.Exit(WaitObject);
        }
        long t1 = Environment.TickCount;
        Console.WriteLine("Total time processing images: {0}ms",
            (t1 - t0));
    }
}
```

Processing 200 images in parallel

Units of Measure

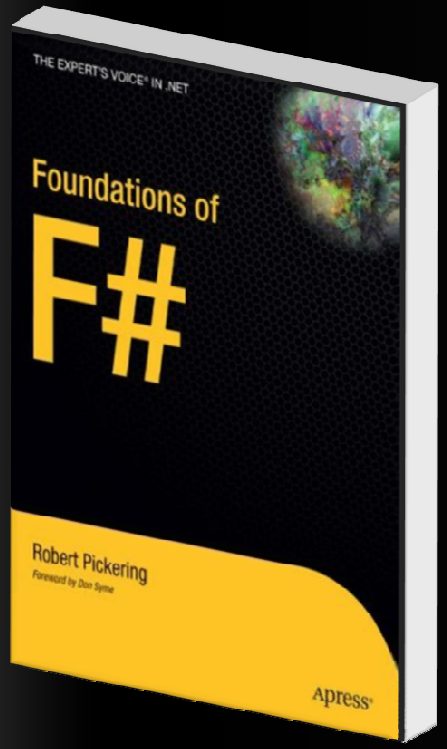
```
let EarthMass = 5.9736e24<kg>

// Average between pole and equator radii
let EarthRadius = 6371.0e3<m>

// Gravitational acceleration on surface of Earth
let g = PhysicalConstants.G * EarthMass / (EarthRadius * EarthRadius)
```

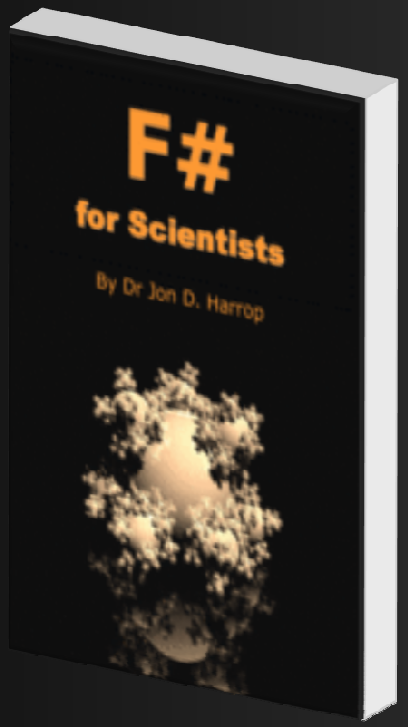
```
let EarthMass = 5.9736e24<Ma
let EarthRadius = 6371.0e3<Ma
let g = Math.PhysicalConstan
let
val g : float<m/s ^ 2>
///
```

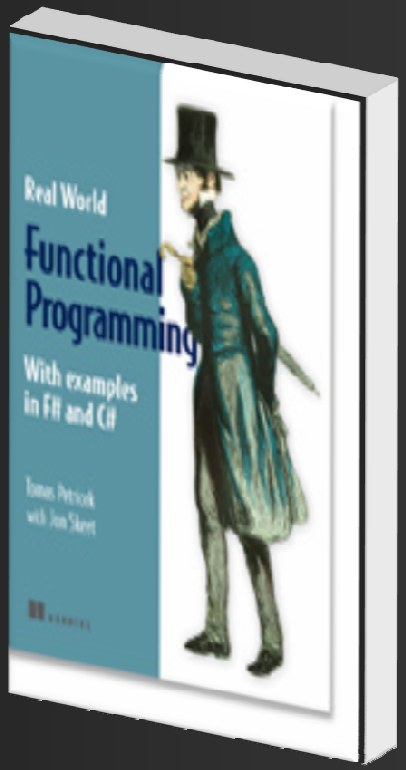
8 Ways to Learn

- FSI.exe
- Samples Included
- Go to definition
- Lutz' Reflector
- http://cs.hubfs.net
- Codeplex Fsharp Samples
- Books
- ML/Erlang/Haskell/ Clojure

Books about F#







Visit www.fsharp.net

Books about F#

Visit **www.fsharp.net**

F# Ahead

F# will be a supported language in Visual Studio 2010

- Next stop: Visual Studio 2010 Beta 2

Look for it soon!

Questions & Discussion