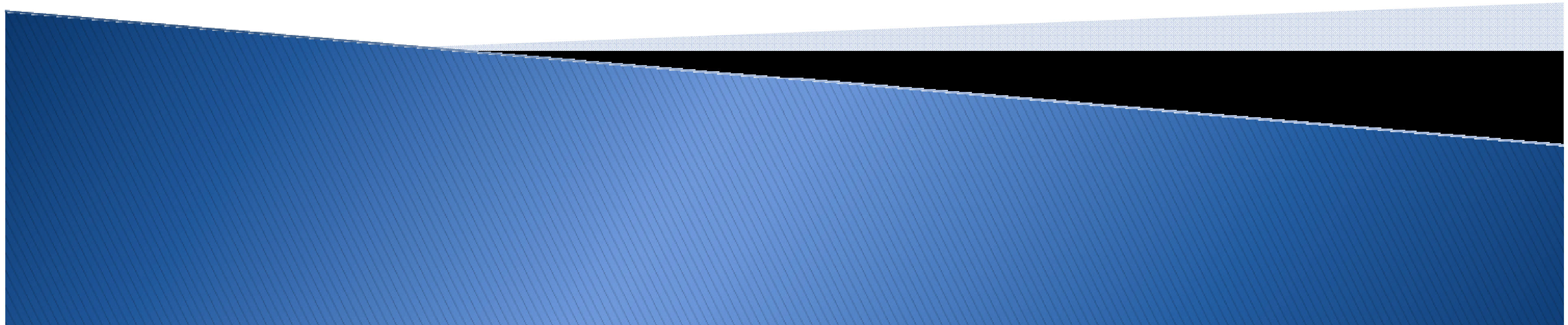
Advanced F#

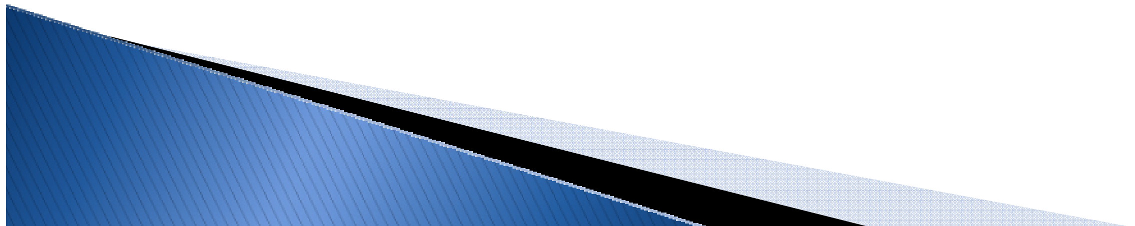
Asynchronous, Parallel, Language Oriented

The F# Team
Microsoft



Agenda

- Asynchronous and Parallel Programming with F# Workflows
- Some other F# Language Oriented Programming Techniques
- Lots of Examples

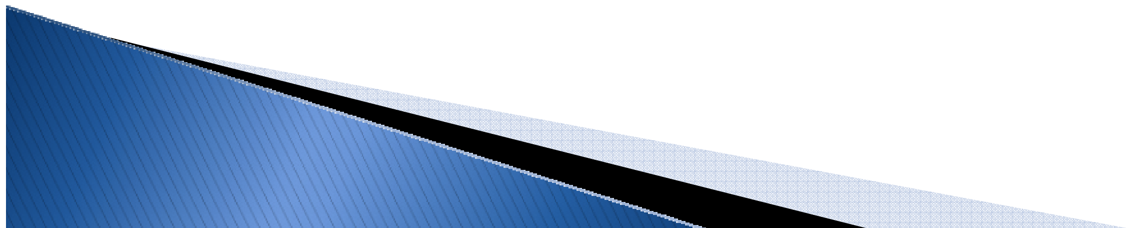


What is F# anyways?

F# is a .NET programming language

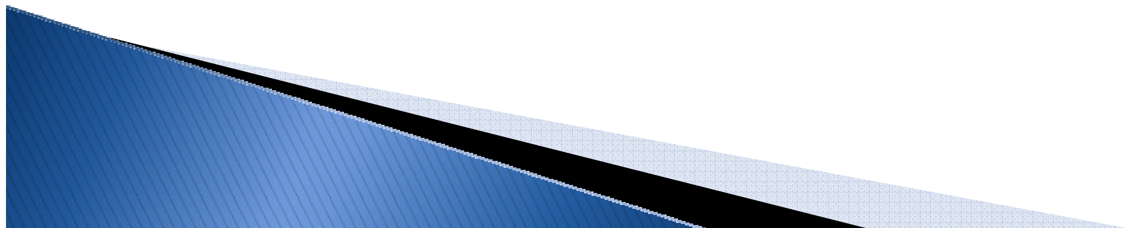
F# is:

- Functional
- Imperative
- Object Oriented



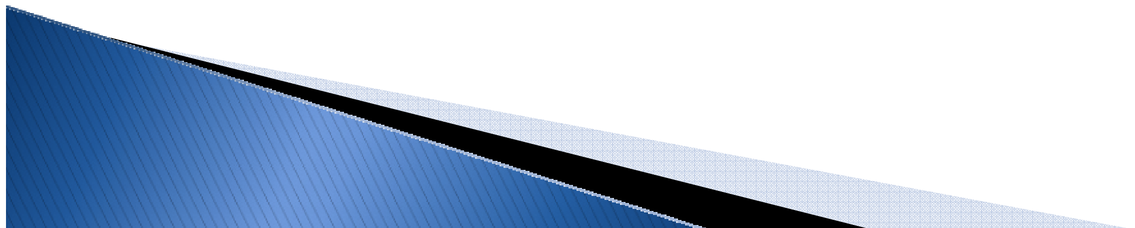
Functional

- Emphasis is on what is to be computed not how it happens
- Data is immutable by default
- Ability to express higher-order functions



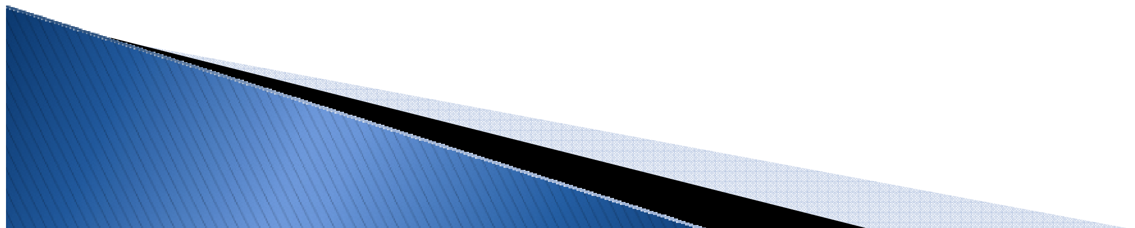
Imperative

- Side effects
- Ability to declare and mutate variables
- Control flow (while, for, if, etc.)

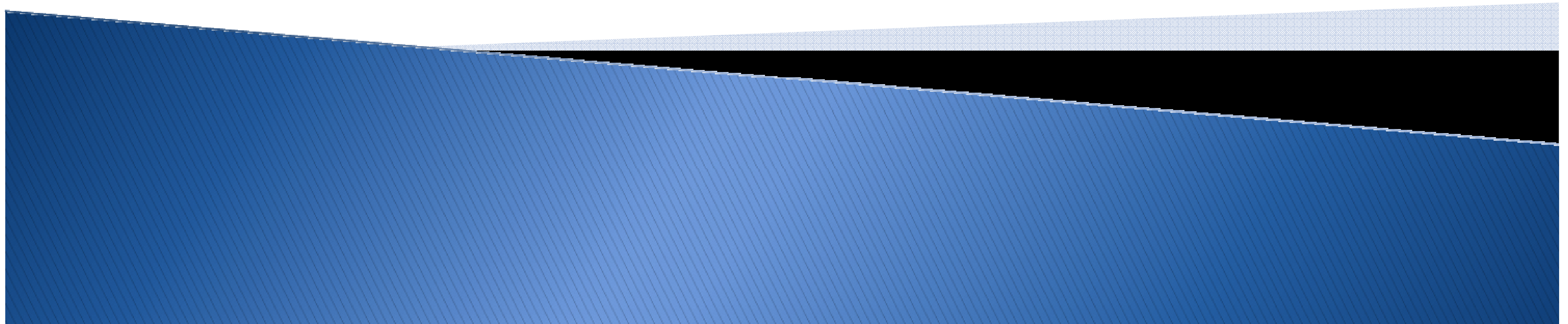


Object Oriented (and .NET)

- Classes and Structs
- Polymorphism and Inheritance
- Events
- Succinct, type-inferred classes



Asynchronous and Parallel Programming

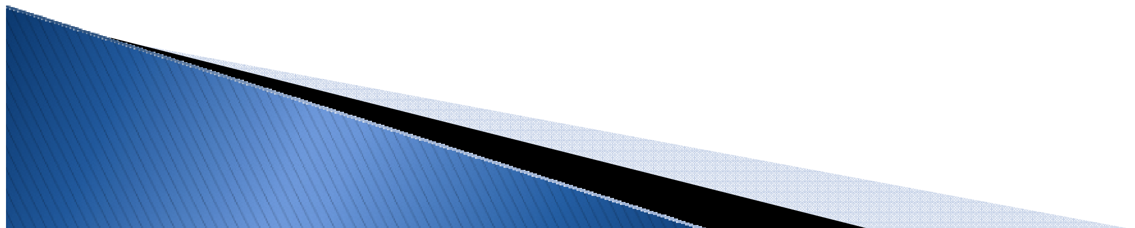


Concurrent programming with shared state…

…can be hard

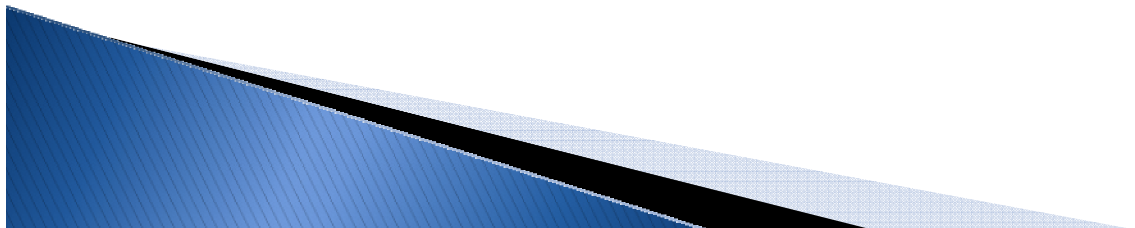
F# - Concurrent/Reactive/Parallel

- Concurrent: *Multiple threads* of execution
- Parallel: These execute *simultaneously*
- Asynchronous: Computations that complete "*later*"
- Reactive: *Waiting* and *responding* is normal



Why is it so hard?

- To get 50 web pages in parallel?
- To get from thread to thread?
- To create a worker thread that reads messages?
- To handle failure on worker threads?



Why isn't it this easy?

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

Why isn't it this easy?

```
let task =
    async { ...
          do! SwitchToNewThread()
          ...
          do! SwitchToThreadPool()
          ...
          do! SwitchToGuiThread()
          .... }
```

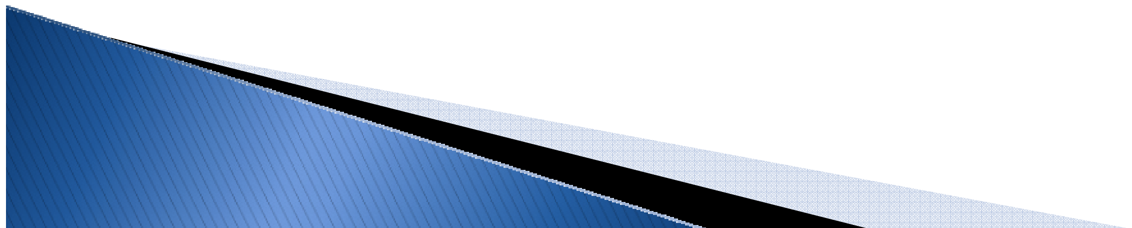
Some Foundation Technologies

OS Threads

System.Threading

.NET Thread Pool

Parallel Extensions for .NET



Taming Asynchronous I/O

Target:
make it easy
to use
Begin/End
operations

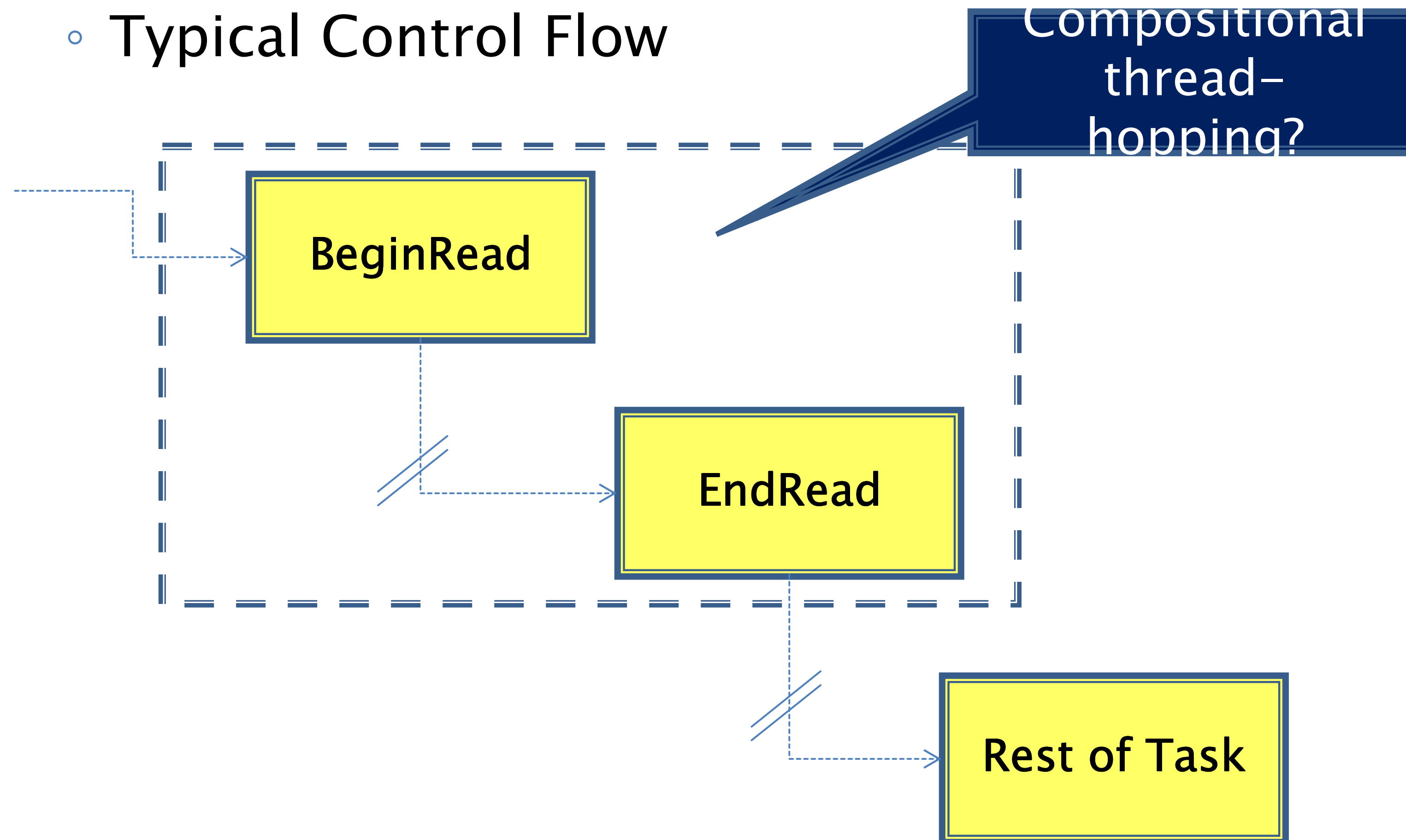
```
inStream. (pixels,0,numPi...s,
          ...cCallback(fun iar -> conti
          ...e=null) |> ignore

// Wait un...hed
continueEv...
```

BeginRead
BeginWrite
CanRead
CanSeek
CanTimeout
CanWrite
Close
CreateObjRef

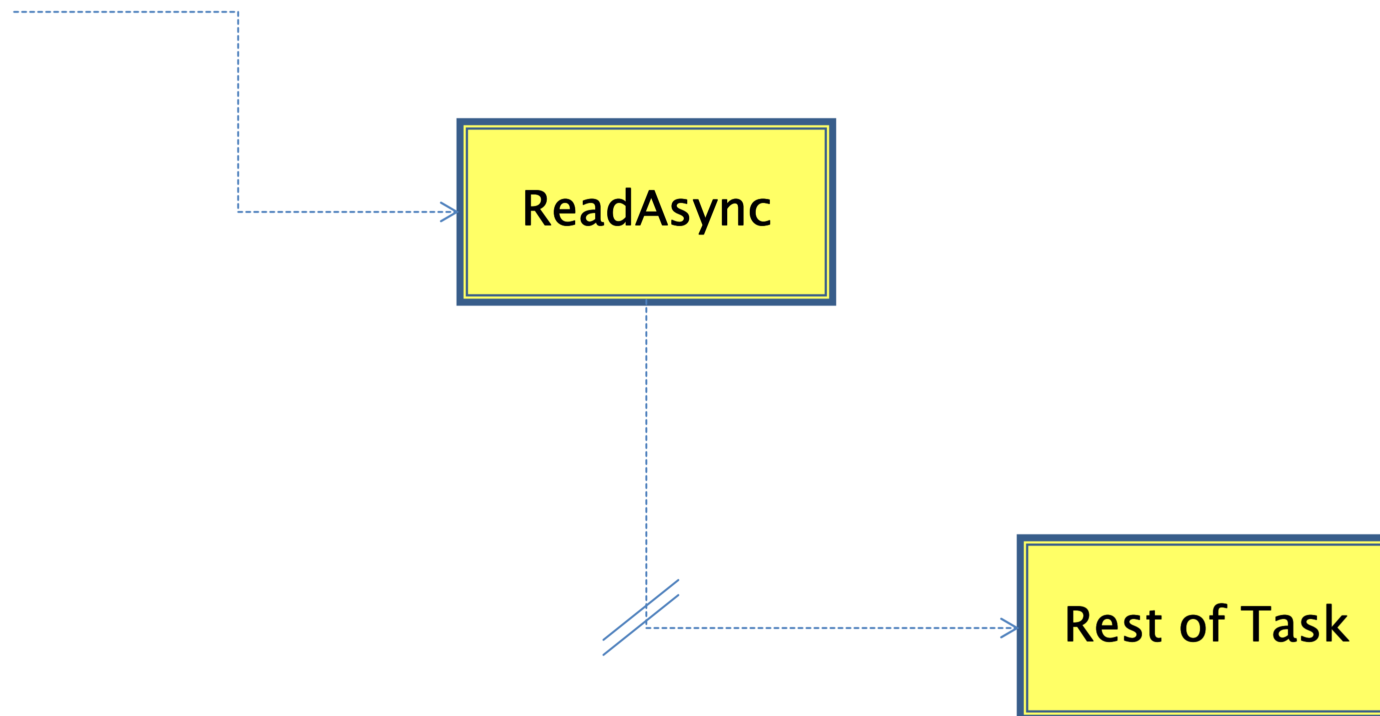
```
Stream.BeginRead : ...

Stream.EndRead : IAsyncResult * ...
```

Taming Asynchronous I/O

- Typical Control Flow



Taming Asynchronous I/O



Simple Examples

Compute 22 and 7 in parallel

```
Async.Parallel [ async { -> 2*2 + 3*6 };
                 async { -> 3 + 5 - 1 } ]
```

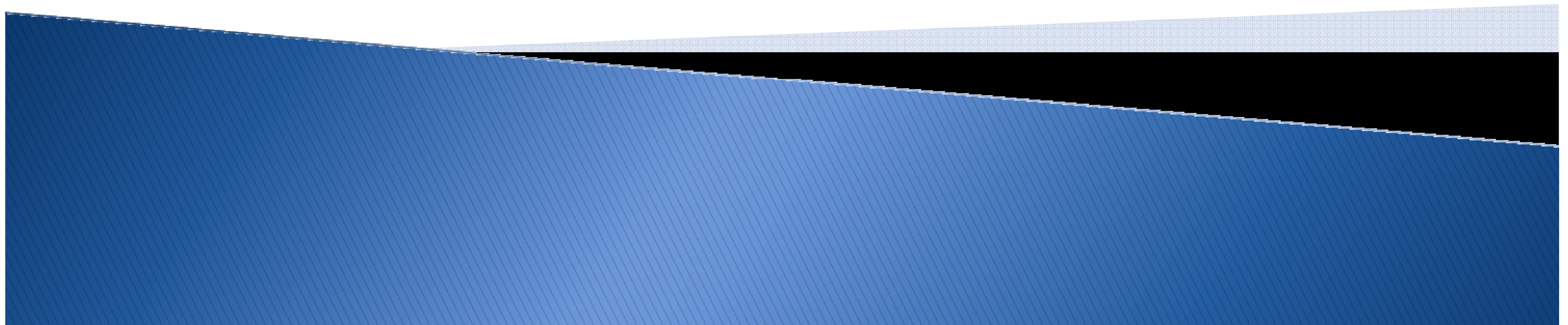
Get these three web pages and wait until all have come back

```
Async.Parallel [WebRequest.Async "http://www.live.com";
                WebRequest.Async "http://www.yahoo.com";
                WebRequest.Async "http://www.google.com" ]
```

```
let parArrMap f (arr: _[]) =
   Async.Run (Async.Parallel [| for x in arr -> async { -> f x } |])
```

Naive Parallel Array Map

Async Web Services



Taming Asynchronous I/O

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

public class BulkImageProcAsync
{
    public const String ImageBaseNa
    public const int numImages = 20
    public const int numPixels = 51

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

    // Threads must decrement NumIm
    // their access to it through a
    public static int NumImagesToFi
    public static Object[] NumImage
    // WaitObject is signalled when
    public static Object[] WaitObje
    public class ImageStateObject
    {
        public byte[] pixels;

    }
```

```
public static void ReadInImageCallback(IAsyncResult a
{
    ImageStateObject state = (ImageStateObject)asyncR
    Stream stream = state.fs;
    int bytesRead = stream.EndRead(asyncResult);
    if (bytesRead != numPixels)
        throw new Exception(String.Format
            ("In ReadInImageCallback, got the wrong n
            "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
    // It ends up swamping the threadpool, because th
    // threads are blocked on I/O requests that were
    // the threadpool.
    FileStream fs = new FileStream(ImageBaseName + st
        ".done", FileMode.Create, FileAccess.Write, F
        4096, false);
    fs.Write(state.pixels, 0, numPixels);
    fs.Close();

    much memory.
    sible is a good

    now.

    );
}
```

```
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 ... until all are finished
    bool mustBloc
    lock (NumImagesMutex)
    {
        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));
}
}
```

```
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 ])
```

Processing 200 images in parallel

Taming Asynchronous I/O

Equivalent F# code (same perf)

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

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

Open the file, synchronously

Read from the file, asynchronously

This object coordinates

Write the result, asynchronously

Generate the tasks and queue them in parallel

"!" = "asynchronous"

Taming Asynchronous I/O

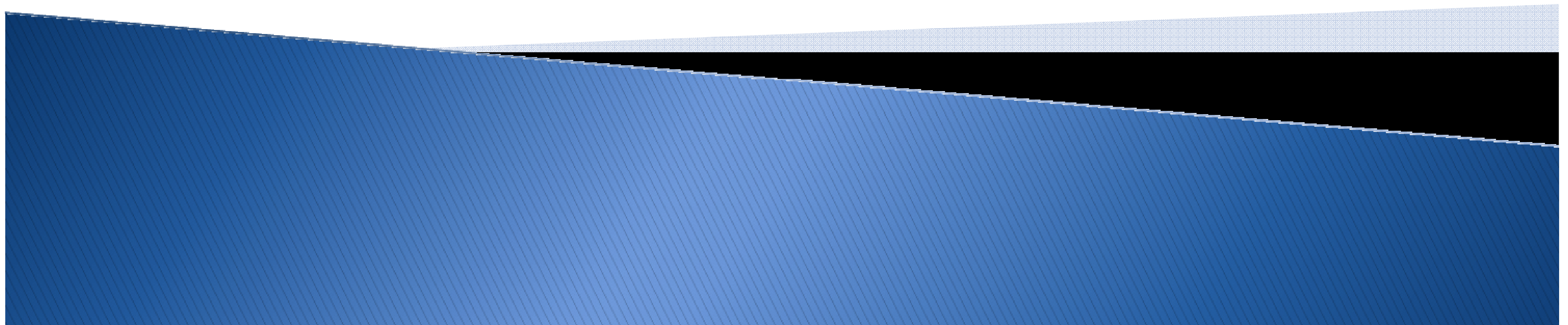
```
let ProcessImageAsync () =
    async { let  inStream  = File.OpenRead(sprintf "Image%d.tmp" i)
            let! pixels    = inStream.ReadAsync(numPixels)
            let  pixels'   = TransformImage(pixels,i)
            let  outStream = File.OpenWrite(sprintf "Image%d.done" i)
            do!  outStream.WriteAsync(pixels')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

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

- Create 10, 000s of "asynchronous tasks"
- Mostly queued, suspended and executed in the thread pool

- Exceptions can be handled properly
- Cancellation checks inserted automatically
- Resources can be disposed properly on failure
- CPU threads are not blocked

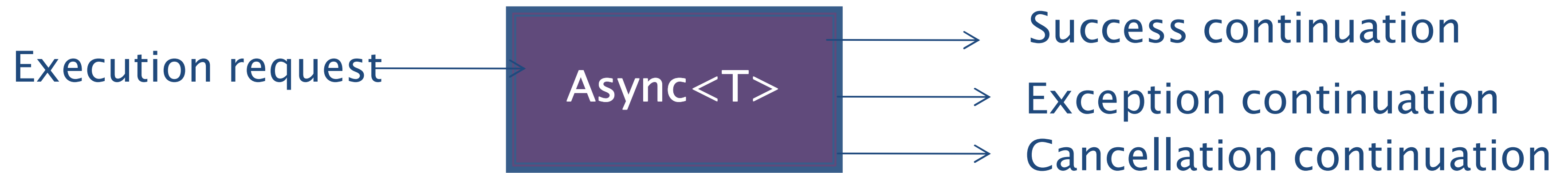
Demo: Asynchronous Image Processing

Don Syme



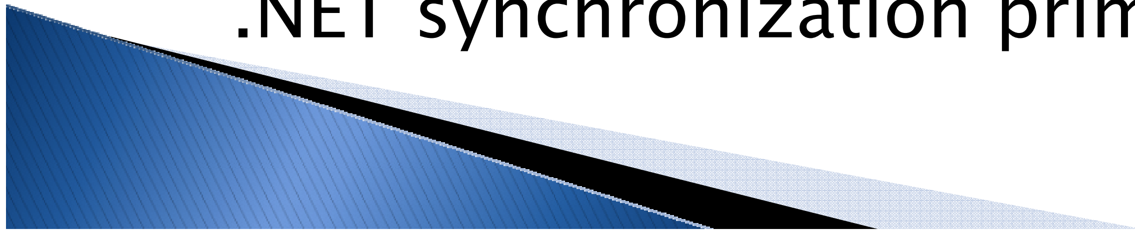
How does it work?

- Uses Computational LOP to make writing continuation-passing programs simpler and compositional



- Similar to techniques used in Haskell

- A wrapper over the .NET Thread Pool and .NET synchronization primitives



F# “Workflow” Syntax

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

Asynchronous "non-blocking" action

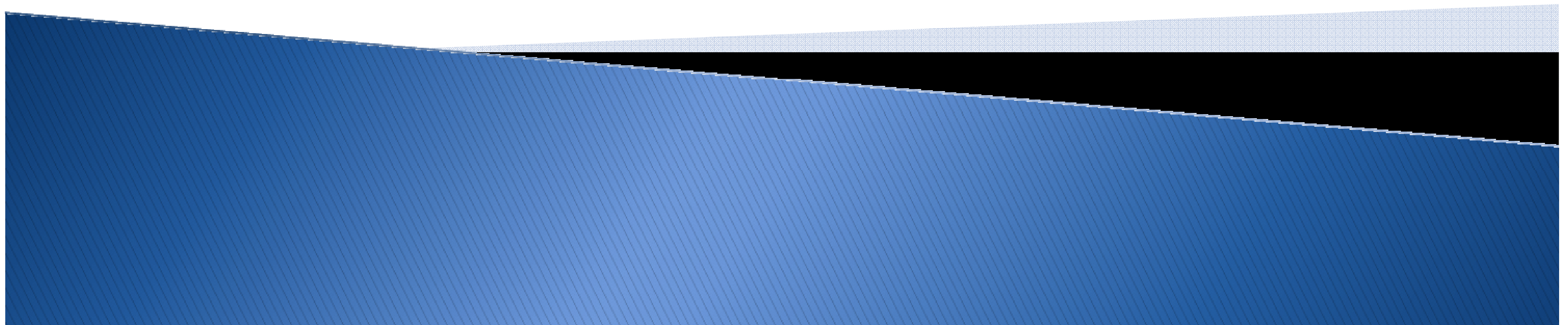
Continuation/ Event callback

You're actually writing this (approximately):

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

Part II: More Language Oriented Programming Techniques

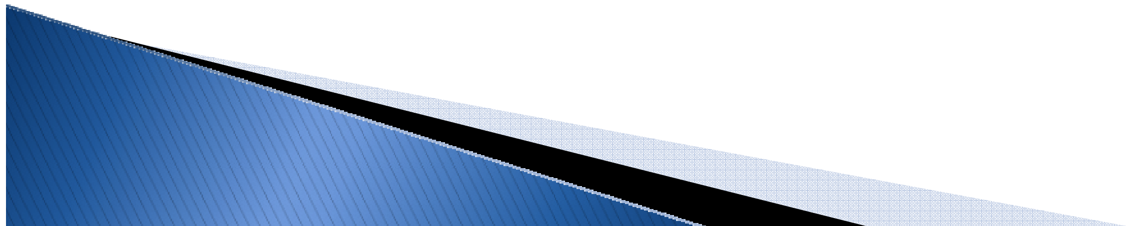
Don Syme



Domain Specific Language (DSL)

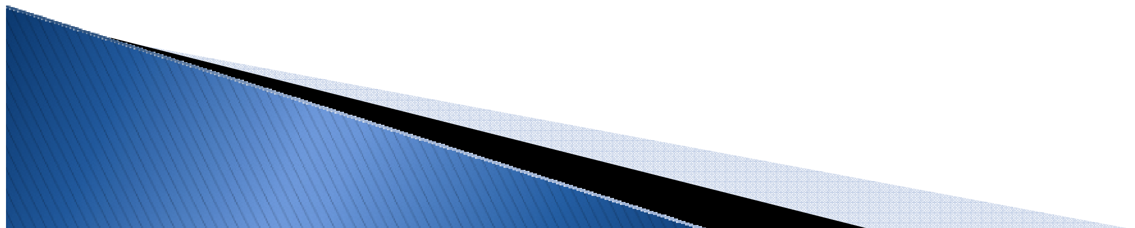
- A custom programming language designed to solve a specific set of problems

- Examples
  - Excel
  - Windows Shell
  - Regular Expressions
  - HTML



Language Oriented Programming

- A style where you apply the ideas of a DSL in a general purpose programming language
  - Bridges the gap between a separate, domain-specific language and the code you write
  - Ability to process problems described in a DSL

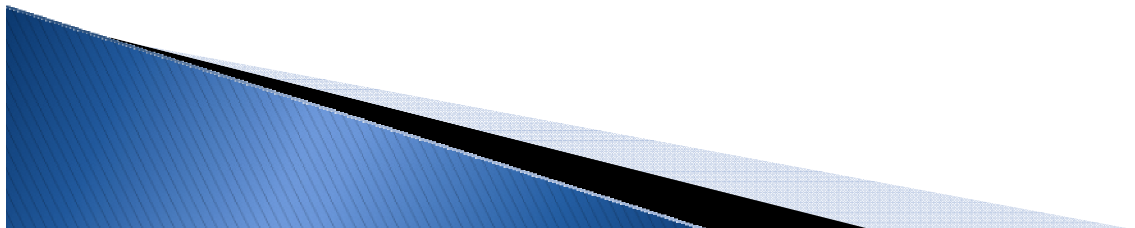


Examples

- XML a **concrete** language representation

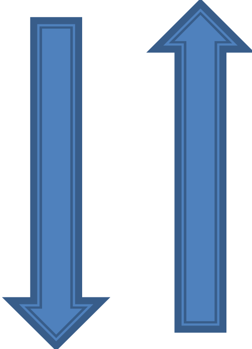
- A parser tree or object model is an **abstract** language representation

- Asynchronous workflows are a **integrated** language representation



LOP Taxonomy

The language is in the data



The language is in the code

Concrete Representations

Abstract Representations

Integrated Representations

XML, CSV, Text, Strings, JSON

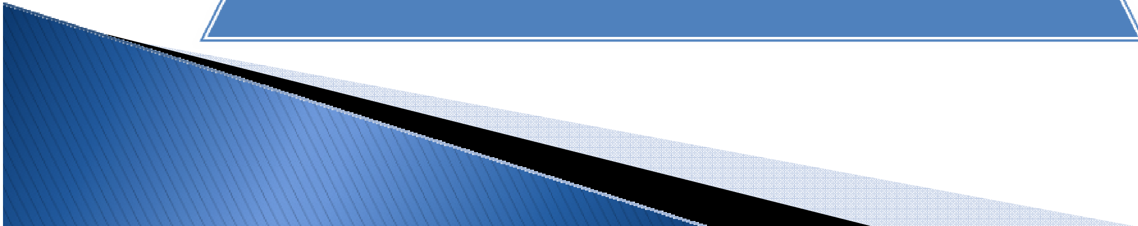
Parse Trees

Almost-Implicit Parallelism

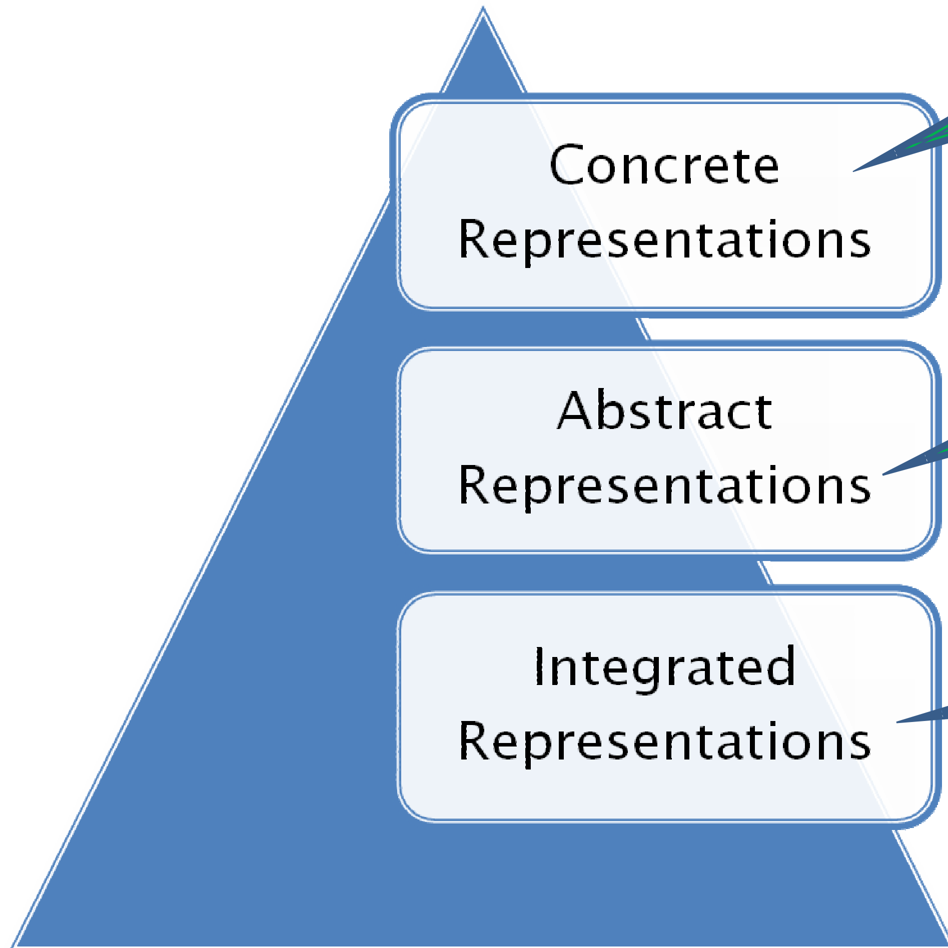
Queries

Exception Handling

Workflows



LOP Techniques



XML Libraries
RegExp Libraries
Lex/Yacc
...

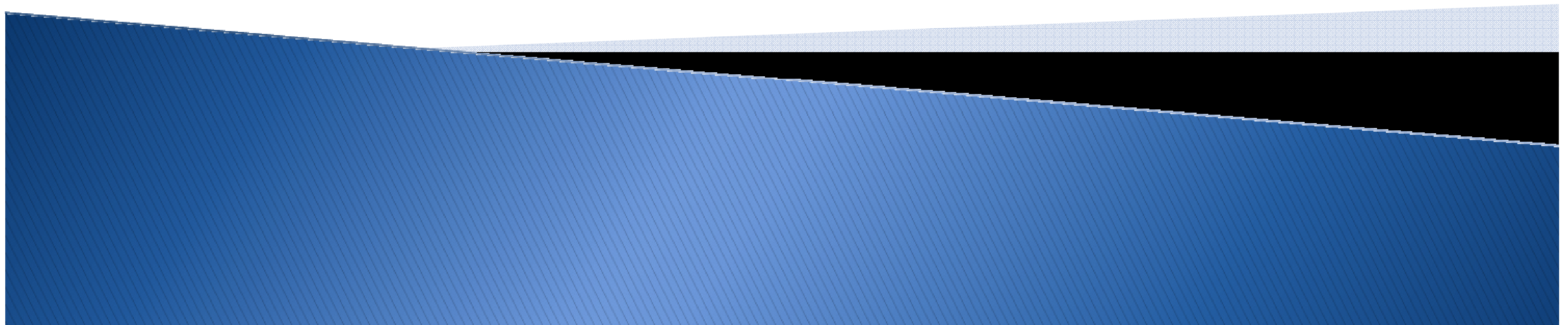
Discriminated Unions

Pattern Matching

F# Computation Expressions

Expression Trees

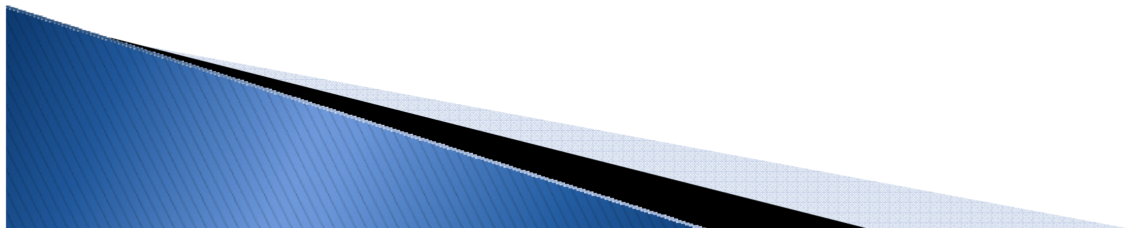
Onto Visual Studio!



Wrapping it Up

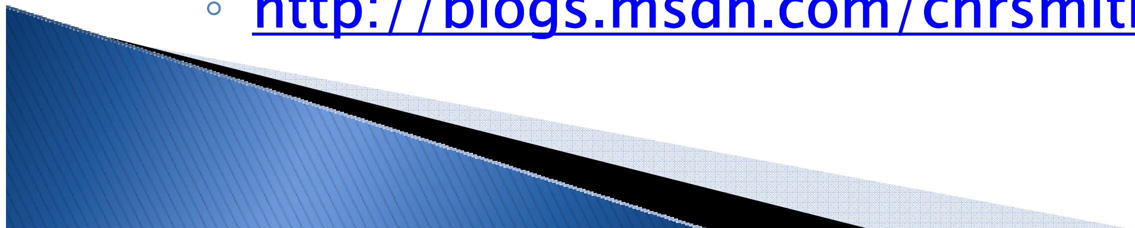
- F# has capabilities which enable LOP
  - Representing other languages in F#
  - Extracting other languages into F#
  - Allowing F# to process in other languages/domains

- LOP makes code that is cleaner and easier to understand



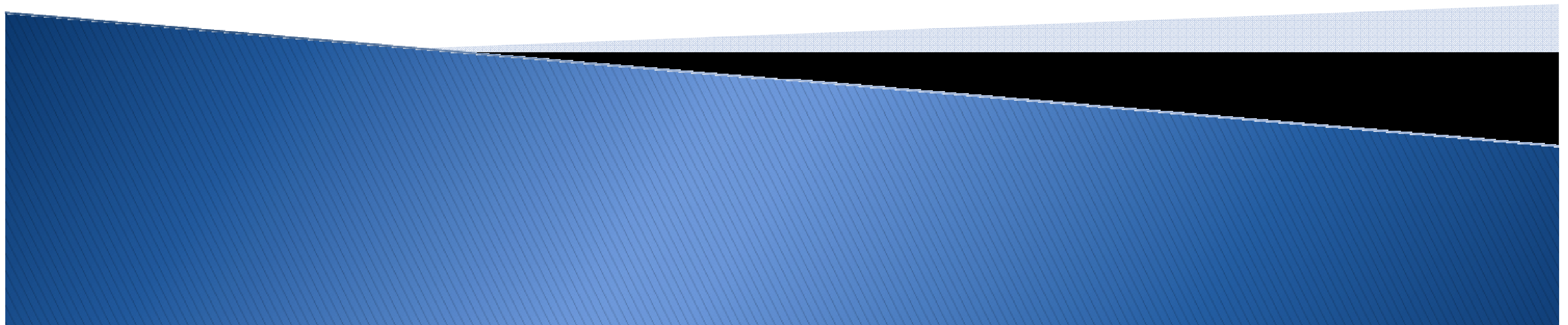
F# Resources

- Get F#
  - http://research.microsoft.com/fsharp
  - Includes add-in for VS2005 and VS2008
- Books
  - *Expert F#*
    - Don Syme, Adam Granicz, and Antonio Cisternino
  - *Foundations of F#*
    - Robert Pickering
- Websites
  - http://cs.hubfs.net/
  - http://blogs.msdn.com/chrsmith



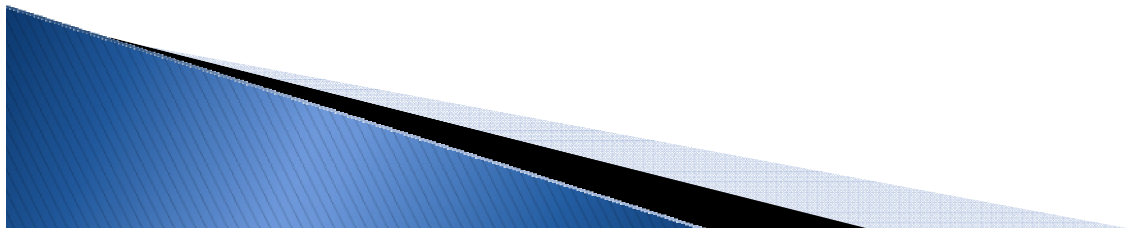
Demo: Asynchronous Web Crawling

Don Syme

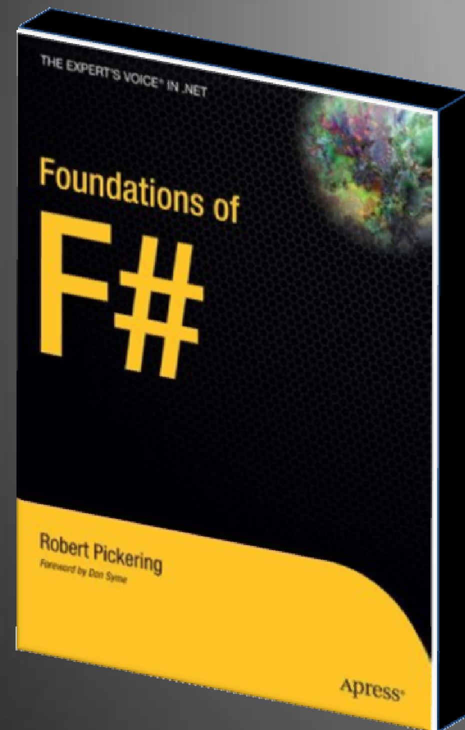


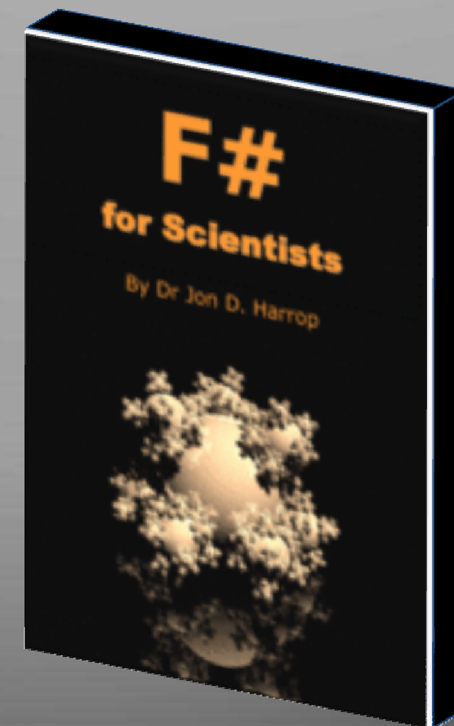
F# – Erlang-style Message Agents

```
open Microsoft.FSharp.Control.Mailboxes

let counter =
  MailboxProcessor.Create(fun inbox ->

    /// Loop, receiving messages
    let rec loop(n) =
        async { do printfn "n = %d" n
                let! msg = inbox.Receive()
                return! loop(n+msg) }

    /// Enter the loop
    loop(0))
```



Books about F#





http://research.microsoft.com/fsharp

Questions

Don Syme