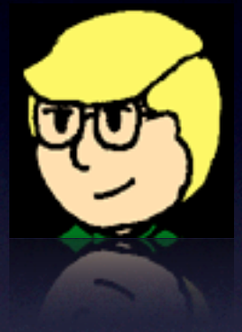
# Brian Ball



SEP
Network and Development
Computer Engineer
Web Developer
Perl
Ruby
Java
C#

myotherpants.com

# When all you have is a hammer ...

sheeshoo @ flickr



Who here is an Object-Oriented Programmer?
Is an algorithm an Object?
We tend to frame things in an object oriented context. That's what we know.
Some problems are more functional-based, algorithmic, recursive.
Functions give you a different set of blocks with which to build.

# F#

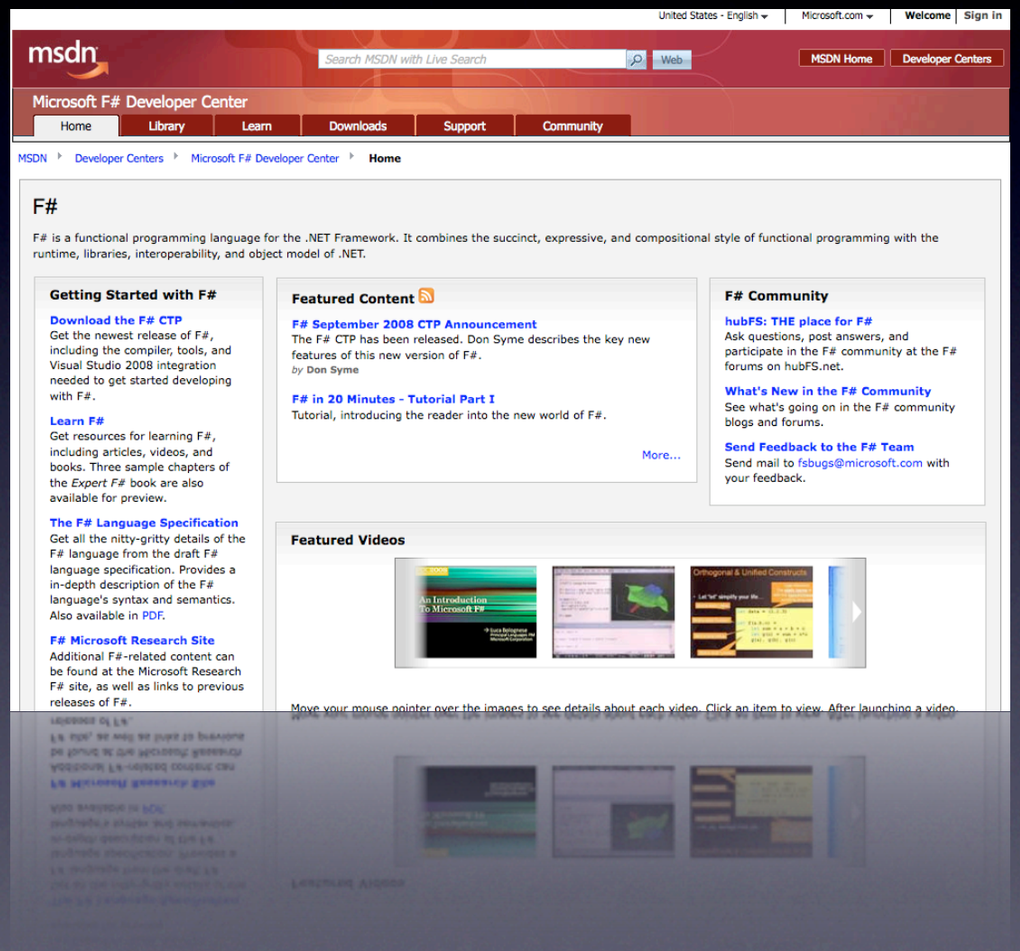
## Functional Programming in the .NET Framework



This gives you access to the CLR and interplay with all of your existing .NET objects, libraries, and programs. You can choose when you need the power and flexibility of functional programming, or when you need OOP.

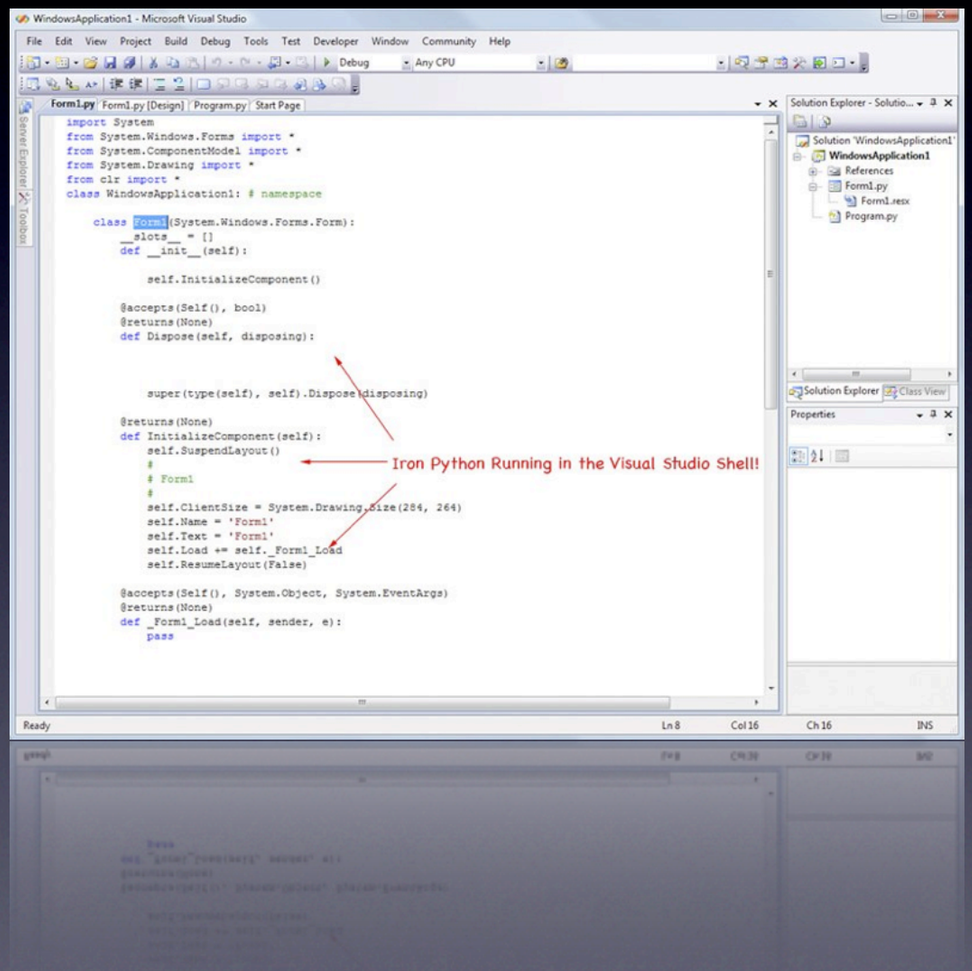
# Where do I get It?

msdn.microsoft.com/fsharp



# VS Shell

download
install
run *isolated mode*
quit
install F#
Run *F# Express*






Download and run VS Shell BEFORE installing FSharp

# What is Functional?

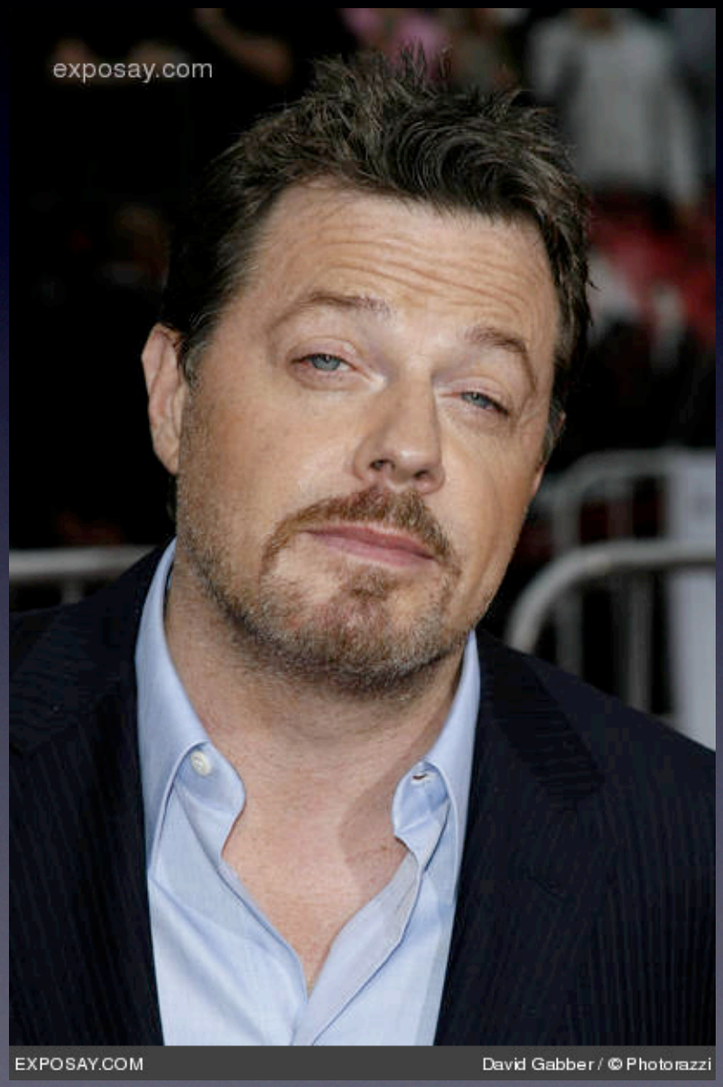


Actions as components of composition

Don’t fear the lambda

Nouns
Verbs

# Composition of Objects




Eddie Izzard is...

- Beard
- Eyes
- Hair
- Mouth
- Ears
- Nose



The oo way is to view the world as a jumble of objects.

# brian.Kick(bucket)



This this the OOP way.  I have an object, brian, who was an associated method, kick, that can act upon several objects.

# Verbs
# Nouns



Functional is about verbs that act on nouns.
Nouns can still be OO, in F Sharp, but the verbs are where you're focusing.

# Composition of Functions

Eddie Izzard is ...
- Running
- Jumping
- Climbing Trees

# kick brian bucket

# Magic




As they say in MIT, it's magic.
Abstractions on abstractions until you get to something so simple, it actually works.

http://swiss.csail.mit.edu/classes/6.001/abelson-sussman-lectures/

# Functional Design

- What are the values to represent
- What are the operations to build, understand, combine, or transform these values
- What equations and algebraic properties hold  between these values



Syme et al: p553

# What You Keep From OO

- DRY
- Single Responsibility
- Liskov Substitution Principle
- Law of Demeter



DRY - If you use a function once, you can use it many times. Functions as values allow you to quickly and naturally create the equivalent template methods or the strategy pattern

Single Responsibility - Self explanitory

Liskov - Differently, no inheritance, but you can think of functions that serve a type of purpose. SQRT estimation example

Law of Demeter - Clean layers of abstraction. Don't leak them. Yes, functions ARE in fact abstractions. Also, your data blobs.

# Defining a function or other bit

```
let PI = 3.1419

let double n = n * 2
```



```
let double n = n * 2

And for something recursive?
let rec factorial n =
      match n with
      | 1 -> 1
      | n -> n * (factorial n-1)
```

# Immutability

x != x + 1

~~let x = x + 1~~
let x = ref (new Tiger())
let mutable x = 5
x <- x + 1



let x = x + 1
let x = new Tiger() ref
let mutable x = 5

I first found this in XSLT, threw me for a loop. Then I read Joe Armstrong's rebuttle. x = x+1 really bothers mathematicians.
Functional programming has a preoccupation with "Side-Effects". If it WAS mutable, it would be shared-state, and that's why threading is so hard. Because of immutability, we know we don't have to worry about shared state

# Wing and a Prayer




Functional programming is also a wing an a prayer.
I'm going to do the same thing over and over and hope I find a place to stop.

# You know it, you love it.

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

# Pattern Matching

```
let rec factorial n =
    match n with
    | 1 -> 1
    | n -> n * (factorial n-1)
```



You match the cases explicitly...

# Pattern Matching with Guards

```
let rec factorial n =
    match n with
    | 1 -> 1
    | _ when n > 1 -> n * factorial n-1
    | _ -> failwith "Factorial is only true for n > 0"
```



```
let rec factorial n =
      match n with
      | 1 -> 1
      | _ when n > 1 -> n * factorial n-1
      | _ -> failwith "Factorial is only true for n > 0"
```

We can also pull apart complex data structions, like lists or descriminating types, match on the object's signature, or create our on "active patterns" to perform operations like match a regular expression.

# Tail Recursion

chris_gin @ flickr



F# knows when to use the stack to call a function and when it doesn't have to. This means you can use recursive algorithms to search impractically large data structures, like generated streams...

# Nested Functions

```
let factorial n =
    let rec helper n acc =
        match n with
        | 1 -> acc
        | _ -> helper (n-1) (n*acc)
    match n with
    | _ when n > 0 -> helper n 1
    | _ -> failwith "Factorial only holds for n > 0"
```



```
let factorial n =
    let rec helper n acc =
        match n with
        | 1 -> acc
        | _ -> helper (n-1) (n*acc)
    match n with
    | _ when n > 0 -> helper n 1
    | _ -> failwith "Factorial only holds for n > 0"
```

The outer function isn't recursive. The nested has any name we want, and isn't visible outside.

# Anonymous Functions

```
(fun x -> x.StartsWith(y))
```

```
(fun (x:string) -> x.StartsWith(y))
```



(fun x -> x.StartsWith(y))

Notice the variable y? It's a closure, it wraps values defined outside of it's normal scope.

It's actually a lie.
(fun (x:string) -> x.StartsWith(y))

We'll get into type inference in a little bit.

# Inferred Types

```
let factorial (n:int) : int =
  ...
```



Remember factorial? What happens if we call factorial with a string, “blah?”
F# knows about the types these operations can perform. Infact, the only problem we have with factorial is if we call it too large, we’ll have to deal with the fact an int doesn’t convert automagically to a bigint.
If F# doesn’t know, like in our anonymous function example, we can tell it what types we expect.

```
let startsWithY = (fun (x:string) :boolean -> x.StartsWith(y))
let StartsWith (y:string) (x:string) :bool = x.StartsWith(y)
```

# Functions as Values

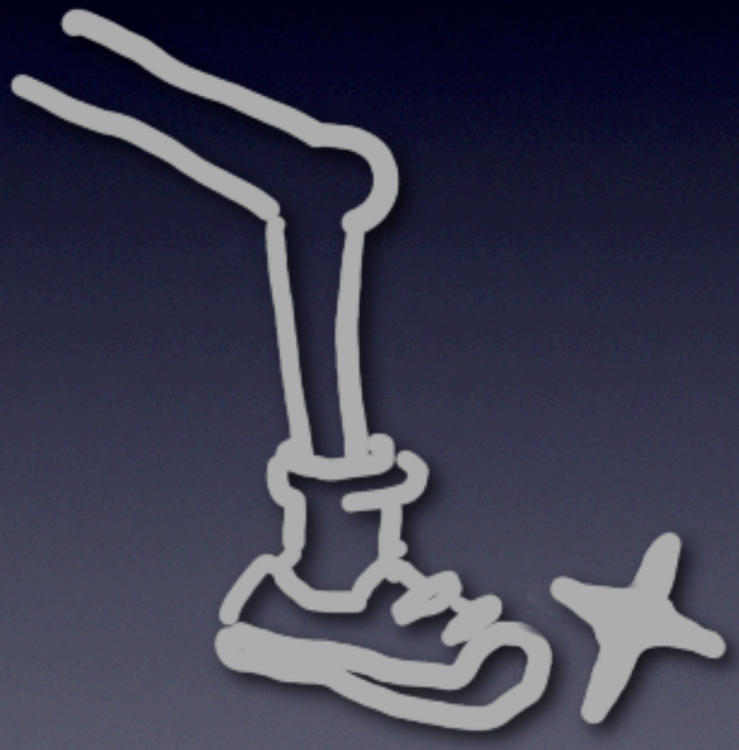
```
let double n = 2 * n

let makeItTwice = double

let ten = makeItTwice 5

let tripple = (fun x -> 3 * x)

List.map (fun x -> x*x) [1;2;3]

List.map tripple [1;2;3]
```



List.map (fun x -> x*x) [1;2;3]
The anonymous function is passed in, like a “strategy” or “command” pattern.
Like passing in a “comparitor” in the String.Compare( method )

# Currying

```
let theBoot = kick brian
```



# Currying

```
let multiply a b = a * b

let makeItTwice = multiply 2
```

# More Types

# Discriminated Unions Tuples

```
type Temperature =
    |Fahrenheit of int
    |Celsius of int

type Tree<'a> =
    |Tree of 'a * Tree<'a>*Tree<'a>
    |Tip of 'a
```




```
type Temperature =
    |Fahrenheit of int
    |Celsius of int

type Tree<'a> =
    |Tree of 'a * Tree<'a>*Tree<'a>
    |Tip of 'a
```

# Functions have types

```
type scaleInteger = int -> int
```

```
type multiplyFunction = int -> int -> int
```

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



but it's rare you'll explicitly define function profiles.
But you do need to read the compiler hints.

# Classes and Intefaces

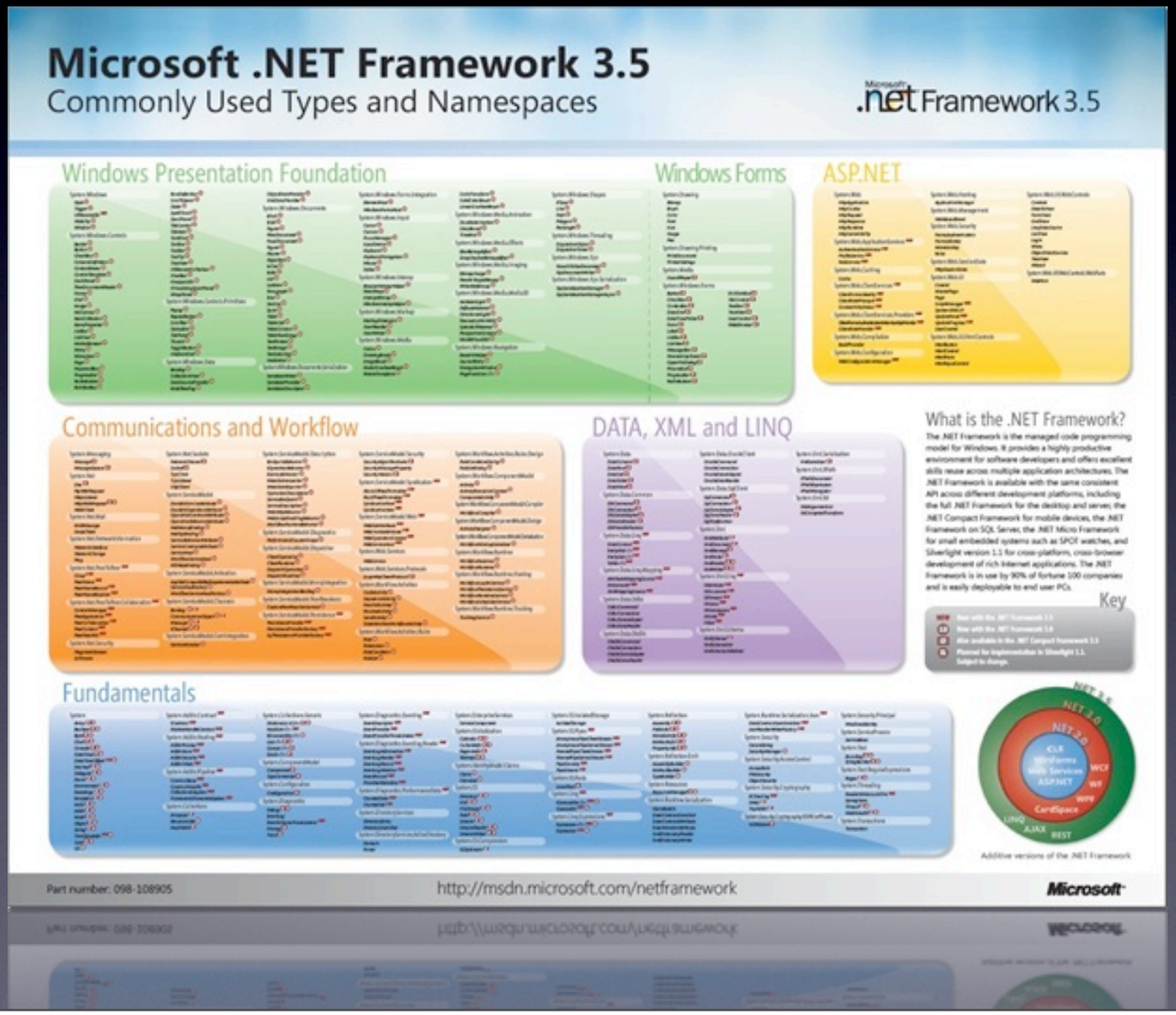
```
type IShape =
    abstract Contains : Point -> bool
    abstract BoundingBox : Rectangle
```

```
type Vector2D(dx:float, dy:float) =
    let len = sqrt(dx*dx + dy*dy)
    member v.DX = dx
    member v.Scale(k) = Vector2D(k*dx, k*dy)
    static member Zero = Vector2D(dx=0.0, dy=0.0)
```



```
type Vector2D(dx:float, dy:float) =
    let len = sqrt(dx*dx + dy*dy)
    member v.DX = dx
    member v.Scale(k) = Vector2D(k*dx, k*dy)
    static member Zero = Vector2D(dx=0.0, dy=0.0)
type IShape =
    abstract Contains : Point -> bool
    abstract BoundingBox : Rectangle
```

# .NET Interoperability






Some of the “infered” types are difficult to use, so your public interfaces need to be clean.

The VS doesn’t deal with “references” quite right, yet.

# .NET Annotations

```
[<TestFixture>]
type Test() =
  [<Test>]
  member x.EmptyString() =
    Assert.IsTrue(isPalindrome("mom"))
```



Yes, virginial, this is a full .NET language.

# Concurrency




Like they said in the Blues Brothers, “We got both kinds.  Threads AND Asynchrony”

# Threads

chefranden @ flickr



You know them.  Moving on....

# Asynchronous Workflows

Use Asynchronous API calls

Object.Begin*()

Seamless as possible



This is a wrapping for async calls that are already in the .NET apis

You know the ones, meant to handle blocking calls in SATA blocks.

# Asynchronous Workflows

```
let fetchAsync(nm, url:string) =
  async { let req = WebRequest.Create(url)
          let! resp = req.AsyncGetResponse()
          let stream = resp.GetResponseStream()
          let reader = new StreamReader(stream)
          let! html = reader.AsyncReadToEnd()
          do (printfn "Read %d characters for %s"
                      html.Length nm) }
[fetchAsync ("IMA","http://imamuseum.org");
 fetchAsync ("BoingBoing","http://boingboing.net")]
|> Async.Parallel
|> Async.Run
```



The let! means we're running a wrapper for begin and end async functions
The general form is a "workflow" that is extensible.

Oh, and the |> is the "pipe forward" operator that is the same as calling the functions backwards

# Message Passing

"Actor Pattern"

No Shared State

Blocking on a Thread-Safe Queue
With Marshaling

# Message Passing

```
type internal msg =
  |Increment of int |Fetch of AsyncReplyChannel<int>
  |Stop of string

let counter = MailboxProcessor.Start(fun inbox ->
  let rec loop(n) =
    async { let! msg = inbox.Receive()
            match msg with
             |Increment m -> return! loop(n+m)
             |Stop -> return
             |Fetch reply -> do reply.Reply(n)
                             return! loop(n) }
  loop(0))
```



Message passing is a means of communicating to another data to another thread without sharing memory. This reduces contentions of shared resources and eliminates the need for locks.
A mailbox is simple “Posted” with a message. The function inside the mailbox simply executes what’s requested. Tail recursion keeps this from stacking out. the argument to the loop is the mailboxe’s way of changing state.

This is similar to Erlang’s concurrency model with one exception, Erlang scales better, as in there is no difference between mailboxes in the same thread or mailboxes on different continents. I’m sure they’ll come up with a distributed mailbox model.

# A Geeky Example


lonelyfox @ flickr



Roll 1d4 + 3d6 + 3
I want to parse dice rollcodes for a dice rolling application

# Data Types

```
type token =
  |PLUS
  |INT of int
  |PERCENT
...

type tokenStream = token list

type output = token list * tokenStream
```

# Parser

lookFor atoken

concat [number ; letterD ; side ]
alternate [ fulld ; singledie ; justanumber ]
repeat plusNumber

normalizePercent someOutput
performAddition someOutput



Any function that takes an input stream and returns an output

Primitive proto parser – is this token the head of the input stream?

Operations to build parsers
combinations to build larger syntax
options in the syntax
repetition in the syntax

Simplify the output stream