FUNCTIONAL PROGRAMMING

GO vs.

ELIXIR

- Dynamically typed, functional language
- Runs on Erlang VM *BEAM*
- Interoperable with Erlang

LAMBDA CALCULUS

Function:

$$\lambda x.\, x$$

Application:

$$(\lambda x.\, x)y$$

Evaluation:

$$(\lambda x.\, x)y = [y/x]x = y$$

WARM UP

```
func add(x, y int) int {
	return x + y
}

func eval(fun func(int, int) int, x, y int) int {
	return fun(x, y)
}

eval(add, 5, 10) // --> 15
```

WARM UP

```elixir
defmodule Calc do
  def add(x, y), do: x + y
  def eval(fun, x, y), do: fun.(x, y)
end

Calc.eval(&Calc.add/2, 5, 10) # --> 15
```

```elixir
add = fn(x, y) -> x + y end
eval = fn(fun, x, y) -> fun.(x, y)
eval.(add, 5, 10) # --> 15
```

PARAMETRIC POLYMORPHISM

```haskell
id :: a -> a
id x = x
```

TYPE SYSTEMS

```go
func id(x interface{}) interface{} {
        return x
}

id(10).(int)
id(true).(int)
```

TYPE SYSTEMS

```go
//just for improved readability
type any interface{}
type function func(any) any

func compose(g, f function) function {
        return func(x any) any {
                return g(f(x))
        }
}

func pow2(x any) any {
        return x.(int) * x.(int)
}

func sqrt(x any) any {
```

TYPE SYSTEMS

```elixir
hype = "Elixir is awesome"
awesomeness = 42
hype + awesomeness # This is an error
```

```elixir
id = fn a -> a end
id.("some") #--> "some"
```

TYPE SYSTEMS

```
compose = fn(g, f) ->
                fn(arg) -> g.(f.(arg)) end
          end
pow2 = fn(x) -> x * x end
sqrt = fn(x) -> :math.sqrt(x) end

same = compose.(sqrt, pow2)
same.(10) #--> 10.0
```

IMMUTABILITY

- immutable by default
- labels instead of variables
- rebinding and shadowing of labels

IMMUTABILITY

- keyword const
- manual cloning
- call-by-value

IMMUTABILITY

```
package rational

type Rational struct {
	numerator   int
	denominator int
}

func NewRational(numerator int, denominator int) Rational {
	//creates a new Rational number
}

func (x Rational) Multiply(y Rational) Rational {
	return NewRational(x.numerator*y.numerator, x.denominator*y.d
}
```

PURITY

#1 The function depends on its arguments only and is idempotent. This also excludes mutable references and things such as I/O streams.

#2 The function has no side effects, which means the evaluation does not involve any mutation. Note, this also applies to effects appearing to the outer world like I/O.

FUNCTIONAL PROGRAMMER'S TOOLBOX

```elixir
defmodule Length do
        def of([]), do: 0
        def of([_ | tail]), do: 1 + of(tail)
end
```

FUNCTIONAL PROGRAMMER'S TOOLBOX

```
def empty_map?(map) when map_size(map) == 0, do: true
def empty_map?(map) when is_map(map), do: false
```

FUNCTIONAL PROGRAMMER'S TOOLBOX

```elixir
String.split(String.upcase("No pipe sucks"))
#-->["NO", "PIPE", "SUCKS"]

"Pipe rocks" |> String.upcase() |> String.split()
#-->["PIPE", "ROCKS"]
```

STANDARD LIBRARY

> *I wanted to see how hard it was to implement this sort of thing in Go, with as nice an API as I could manage. It wasn't hard. Having written it a couple of years ago, I haven't had occasion to use it once. Instead, I just use "for" loops. You shouldn't use it either.*

STANDARD LIBRARY

```
range = [1, 2, 3, 4]
range  |> Enum.map(fn x -> x * 2 end) #--> [2, 4, 6, 8]
       |> Enum.reduce(fn x, acc -> x + acc end) #--> 20
```

PERFORMANCE

$$f_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f_{n-1} + f_{n-2} & n > 2 \end{cases}$$

```
def fibonacci(0), do: 0
def fibonacci(1), do: 1
def fibonacci(n), do: fibonacci(n-1) + fibonacci(n-2)
```

PERFORMANCE

```
def fibonacci(n), do: pfib(n, 1, 0)
defp pfib(0, _, result), do: result
defp pfib(n, next, result), do: pfib(n-1, next+result, next)
```

CONCLUSION