go

a crash course

http://golang.org/

a powerful language

- low ceremony C-style syntax
- garbage collection
- static inferred typing
- communication-based concurrency
- compiled
- statically linked

concrete types

```
package Integer

type Int int

func (i *Int) Add(x int) {
   *i += Int(x)
}

type Buffer []Int

func (b Buffer) Clone() Buffer {
   s := make(Buffer, len(b))
   copy(s, b)
   return s
}

func (b Buffer) Swap(i, j int) {
   b[i], b[j] = b[j], b[i]
}

func (b Buffer) Move(i, n int) {
   if n > len(b) - i {
      n = len(b) - i
   }
   segment_to_move := b[:i].Clone()
   copy(b, b[i:i + n])
   copy(b[n:i + n], segment_to_move)
}
```

```
package main
import "Integer"
import "fmt"

func main() {
   i := Integer.Buffer{0, 1, 2, 3, 4, 5}
   b := i.Clone()
   b.Swap(1, 2)
   b.Move(3, 2)
   b[0].Add(3)
   fmt.Printf("b[0:2] = %v\n", b[0:2])
}
```

produces:

b[0:2] = [6, 4]

```
package Vector
import . "Integer"

type Vector struct {
    Buffer
}

func (v *Vector) Clone() Vector {
    return Vector{v.Buffer.Clone()}
}

func (v *Vector) Slice(i, j int) Buffer {
    return v.Buffer[i:j]
}

func (v *Vector) Replace(o interface{}) {
    switch o := o.(type) {
    case *Vector:
        *v = *o
    case Vector:
        v = o
    case Buffer:
        v.Buffer = o
    }
}
```

```
package main
import "Vector"
import "fmt"

func main() {
    i := Vector{Buffer{0, 1, 2, 3, 4, 5}}
    v := i.Clone()
    v.Swap(1, 2)
    v.Move(3, 2)
    s := v.Slice(0, 2)
    s[0].Add(3)
    v.Replace(s)
    fmt.Println("b[0:2]", v.Buffer[0:2])
}
```

produces:

b[0:2] = [6, 4]

structural types

```
package main
import "fmt"

type Adder interface {
  Add(j int)
  Subtract(j int)
  Result() interface{}
}

type Calculator struct {
  Adder
}

type IntAdder []int

func (i IntAdder) Add(j int) {
  i[0] += i[j]
}
func (i IntAdder) Subtract(j int) {
  i[0] -= i[j]
}
func (i IntAdder) Result() interface{} {
  return i[0]
}
```

```
type FloatAdder []float

func (f FloatAdder) Add(j int) {
  f[0] += f[j]
}
func (f FloatAdder) Subtract(j int) {
  f[0] -= f[j]
}
func (f FloatAdder) Result() interface{} {
  return f[0]
}

func main() {
    c := Calculator{}
    c.Adder = IntAdder{0, 1, 2, 3, 4, 5}
    c.Add(1)
    c.Add(2)
    c.Subtract(3)
    fmt.Println("c.Result() =", c.Result())

    c.Adder = FloatAdder{0.0, 1.1, 2.2, 3.3, 4.4, 5.5}
    c.Add(1)
    c.Add(2)
    c.Subtract(3)
    fmt.Println("c.Result() =", c.Result())
}
```

produces:

c.Result() = 0

c.Result() = (0x10f94,0x34800000)

reflected types

```
package generalise
import "fmt"
import . "reflect"

func Allocate(i interface{}, limit... int) (n interface{}) {
    switch v := ValueOf(i); v.Kind() {
    case Slice:
        l := v.Cap()
        if len(limit) > 0 {
            l = limit[0]
        }
        n = MakeSlice(v.Type(), l, l).Interface()
    case Map:
        n = MakeMap(v.Type()).Interface()
    }
    return
}

func SwapSlices(i interface{}, d, s, n int) {
    if v := ValueOf(i); v.Kind() == Slice {
        source := v.Slice(s, s + n)
        destination := v.Slice(d, d + n)
        temp := ValueOf(Allocate(i, n))
        Copy(temp, destination)
        Copy(destination, source)
        Copy(source, temp)
    }
}
```

```
func Duplicate(i interface{}) (clone interface{}) {
    if clone = Allocate(i); clone != nil {
        switch clone := ValueOf(clone); clone.Kind() {
        case Slice:
            Copy(clone, ValueOf(i))
        case Map:
            m := ValueOf(i)
            for _, k := range m.MapKeys() {
                clone.SetMapIndex(k, m.MapIndex(k))
            }
        }
    }
    return
}
```

```
package main
import . "generalise"

func main() {
    error_text := "panic caused by"
    defer func() {
        if x := recover(); x != nil {
            fmt.Println(error_text, x)
        }
    }()

    s1 := []int{0, 1, 2, 3, 4, 5}
    fmt.Println("s1 =", s1)
    s2 := Duplicate(s1)
    fmt.Println("Duplicate(s1) =", s2)
    SwapSlices(s2, 0, 3, 3)
    fmt.Println("SwapSlices(s2, 0, 3, 3) =", s2)
    s3 := Allocate(s1, 1)
    fmt.Println("Allocate(s1, 1) =", s3)

    m := map[int] int{1: 1, 2: 2, 3: 3, 0: 0, 4: 4, 5: 5}
    fmt.Println("m =", m)
    n := Allocate(m)
    fmt.Println("Allocate(m) =", n)
    SwapSlices(m, 0, 3, 3)
}
```

produces:

```
s1 = [0 1 2 3 4 5]
Duplicate(s1) = [0 1 2 3 4 5]
SwapSlices(s2, 0, 3, 3) = [3 4 5 0 1 2]
Allocate(s1, 1) = [0]

m = map[3:3 0:0 1:1 4:4 5:5 2:2]
n = map[]
panic caused by map[3:3 0:0 1:1 4:4 5:5 2:2]
```

concurrency

```
package generalise

type SignalSource func(status chan bool)

func Pipeline(s... SignalSource) {
   done := make(chan bool)
   go func() {
      defer close(done)
      for _, f := range s {
         go f(done)
         <-done
      }
   }
   <- done
}
```

```
package generalise
import . "reflect"

type Iterator func(k, x interface{})

func (i Iterator) apply(k, v interface{}, c chan bool) {
    go func() {
        i(k, v)
        c <- true
    }()
}

type Series map[interface{}] interface{}

func (s Series) Apply(f Iterator) {
    Pipeline(func(done chan bool) {
        for k, v := range s {
            f.apply(k, v, done)
        }
        for i := 0; i < len(s); i++ {
            <- done
        }
    })
}
```

```
func Each(c interface{}, f Iterator) (i int) {
    s := make(map[interface{}] interface{})
    switch c := ValueOf(c); c.Kind() {
    case Slice:
        for i := 0; i < c.Len(); i++ {
            s[i] = c.Index(k).Interface()
        }
    case Map:
        for _, k := range c.Keys() {
            s[i] = c.MapIndex(k).Interface()
        }
    }
    return s.Apply(f)
}
```

```
package generalise
import . "reflect"

type Results chan interface{}

type Combinator func(x, y interface{}) interface{}

func (f Combinator) Reduce(c, s interface{}) (r Results) {
   r = make(Results)
   Each(c, func(k, x interface{}) {
      s = f(s, x)
   })
   r <- s
   return
}
```

```
package generalise
import . "reflect"

type Transformer func(x interface{}) interface{}

func (t Transformer) GetValue(x interface{}) Value {
    return ValueOf(t(x))
}

func (t Transformer) Map(c interface{}) (r interface{}) {
    switch n := ValueOf(Allocate(c)); n.Kind() {
    case Slice:
        Each(c, func(k, x interface{}) {
            Index(k.(int)).Set(t.GetValue(x))
        })
        r = n.Interface()
    case Map:
        Each(c, func(k, x interface{}) {
            n.SetElem(NewValue(k), t.GetValue(x))
        })
        r = n.Interface()
    default:
        r = Duplicate(c)
    }
    return
}
```

```
package main
import "fmt"
import . "generalise"

var adder Combinator = func(x, y interface{}) interface{} {
  return x.(int) + y.(int)
}

var multiplier Transformer = func(x interface{}) interface{} {
  return x.(int) * 2
}

func main() {
   s := []int{0, 1, 2, 3, 4, 5}
   fmt.Println("s =", s)
   fmt.Println("sum s =", <- adder.Reduce(s, 0))

   c := multiplier.Map(s)
   fmt.Println("c =", c)
   fmt.Println("sum c =", <- adder.Reduce(c, 0))
}
```

produces:

```
s = [0 1 2 3 4 5]
sum s = 15
c = [0 2 4 6 8 10]
sum c = 30
```