portrait of an artist...

- physics major
- embedded controllers
- software reliability
- dynamic languages
- network scaling
- questionable taste in music

http://github.com/feyeleanor

Eleanor McHugh

go...

small, safety-conscious systems language

concurrency, closures & garbage collection

consistently fast compilation

hello world

```
package main

import "fmt"

const(
  HELLO string = "hello"
  WORLD string = "world"
)

func main() {
  fmt.Println(HELLO, WORLD)
}
```

user-defined type

```
package Integer

type Int int

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

```
package Integer

type Buffer []Int

func (b Buffer) Eq(o Buffer) (r bool) {
  if len(b) == len(o) {
    for i := len(b) - 1; i > 0; i-- {
      if b[i] != o[i] {
        return
      }
      r = true
    }
    return
  }

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

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

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"

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)
  println("b[:2] = {", b[0], ",", b[1], "}")
}
```

produces:

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

```
package Integer
import "testing"

func TestSwap(t *testing.T) {
  i := Buffer{0, 1, 2, 3, 4, 5}
  b := i.Clone()
  b.Swap(1, 2)
  if !b[1:3].Eq(Buffer{2, 1}) {
    t.Fatalf("b[:5] = %v", b)
  }
}

func TestMove(t *testing.T) {
  i := Buffer{0, 1, 2, 3, 4, 5}
  b := i.Clone()
  b.Move(3, 2)
  if !b.Eq(Buffer{3, 4, 0, 1, 2, 5}) {
    t.Fatalf("b[:5] = %v", b)
  }
}
```

```
func TestAdd(t *testing.T) {
  i := Buffer{0, 1, 2, 3, 4, 5}
  b := i.Clone()
  b[0].Add(3)
  if b[0] != i[0] + 3 {
    t.Fatalf("b[:5] = %v", b)
  }
}
```

embedding

```
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]
}
```

```
package Integer
import "testing"

func TestVectorSwap(t *testing.T) {
  i := Vector{Buffer{0, 1, 2, 3, 4, 5}}
  v := i.Clone()
  v.Swap(1, 2)
  r := Vector{Buffer{0, 2, 1, 3, 4, 5}}
  switch {
  case !v.Match(&r):
    fallthrough
  case !v.Buffer.Match(r.Buffer):
    t.Fatalf("b[:5] = %v", v)
  }
}
```

```
package integer
import "testing"

func BenchmarkVectorClone6(b *testing.B) {
  v := Vector{Buffer{0, 1, 2, 3, 4, 5}}
  for i := 0; i < b.N; i++ {
    _ = v.Clone()
  }
}
```

```
func BenchmarkVectorSwap(b *testing.B) {
  b.StopTimer()
  v := Vector{Buffer{0, 1, 2, 3, 4, 5}}
  b.StartTimer()
  for i := 0; i < b.N; i++ {
    v.Swap(1, 2)
  }
}
```

```
$ go test -test.bench="Benchmark"
PASS
integer.BenchmarkVectorSwap    200000000               8 ns/op
integer.BenchmarkVectorClone6  10000000          300 ns/op
```

inference

```
package adder

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

type Calculator interface {
  Adder
  Reset()
}

type AddingMachine struct {
  Memory interface{}
  Adder
}
```

```
package adder

type IAdder []int

func (i IAdder) Add(j int) {
  i[0] += i[j]
}

func (i IAdder) Subtract(j int) {
  i[0] -= i[j]
}

func (i IAdder) Result() interface{} {
  return i[0]
}

func (i IAdder) Reset() {
  i[0] = *new(int)
}
```

```
package adder
import "testing"

func TestIAdder(t *testing.T) {
  error := "Result %v != %v"
  i := IAdder{0, 1, 2}
  i.Add(1)
  if i.Result().(int) != 1 { t.Fatalf(error, i.Result(), 1) }
  i.Subtract(2)
  if i.Result().(int) != -1 { t.Fatalf(error, i.Result()), -1 }
  var r Calculator = IAdder{-1, 1, 2}
  for n, v := range r.(IAdder) {
    if i[n] != v { t.Fatalf("Adder %v should be %v", i, r) }
  }
  r.Reset()
  if r.Result().(int) != *new(int) {
    t.Fatalf(error, r.Result(), *new(int))
  }
}
```

```
package adder
import "testing"

func TestIAdder(t *testing.T) {
  error := "Result %v != %v"
  i := IAdder{0, 1, 2}
  i.Add(1)
  if i.Result() != 1 { t.Fatalf(error, i.Result(), 1) }
  i.Subtract(2)
  if i.Result() != -1 { t.Fatalf(error, i.Result()), -1 }
  var r Calculator = IAdder{-1, 1, 2}
  for n, v := range r.(IAdder) {
    if i[n] != v { t.Fatalf("Adder %v should be %v", i, r) }
  }
  r.Reset()
  if r.Result() != *new(int) {
    t.Fatalf(error, r.Result(), *new(int))
  }
}
```

```
package adder

type FAdder []float32

func (f FAdder) Add(j int) {
  f[0] += f[j]
}

func (f FAdder) Subtract(j int) {
  f[0] -= f[j]
}

func (f FAdder) Result() interface{} {
  return f[0]
}

func (f FAdder) Reset() {
  f[0] = *new(float32)
}
```

```
package adder
import "testing"

func TestFAdder(t *testing.T) {
  error := "Result %v != %v"
  f := FAdder{0.0, 1.0, 2.0}
  f.Add(1)
  if f.Result() != 1.0 { t.Fatalf(error, f.Result(), 1.0) }
  f.Subtract(2)
  if i.Result() != -1.0 { t.Fatalf(error, i.Result()), -1.0 }
  var r Calculator = FAdder{-1.0, 1.0, 2.0}
  for n, v := range r.(FAdder) {
    if f[n] != v { t.Fatalf("Adder %v should be %v", f, r) }
  }
  r.Reset()
  if r.Result() != *new(float32) {
    t.Fatalf(error, r.Result(), *new(float32))
  }
}
```

```
package adder
import "testing"

func TestAddingMachine(t *testing.T) {
  error := "Result %v != %v"
  a := &AddingMachine{ Adder: FAdder{0.0, 1.0, 2.0} }
  a.Add(1)
  if f, ok := a.Result().(float32); !ok {
    t.Fatal("Result should be a float32")
  } else if f != 1.0 {
    t.Fatalf(error, a.Result(), 1.0)
  }
  a.Subtract(2)
  if a.Result().(float32) != -1.0 { t.Fatalf(error, a.Result(), -1.0) }
  r := FAdder{-1.0, 1.0, 2.0}
  for n, v := range a.Adder.(FAdder) {
    if r[n] != v { t.Fatalf("Adder %v should be %v", a, r) }
  }
}
```

concurrency

goroutines

concurrent execution stacks

initialised with a closure

scheduled automatically by the runtime

```
package main
import "fmt"

func main() {
  var c chan int
  c = make(chan int)
  go func() {
    for {
      fmt.Print(<-c)
    }
  }()
  for {
    select {
    case c <- 0:
    case c <- 1:
    }
  }
}
```

produces:

0110011110101l0...

```
package main
import "fmt"

func main() {
  var c chan int
  c = make(chan int, 16)
  go func() {
    for {
      fmt.Print(<-c)
    }
  }()
  go func() {
    select {
    case c <- 0:
    case c <- 1:
    }
  }()
  for {}
}
```

produces:

01100111010110...

```
package map_reduce

type SignalSource func(status chan bool)

func Wait(s SignalSource) {
  done := make(chan bool)
  defer close(done)
  go s(done)
  <-done
}
```

```
func WaitCount(count int, s SignalSource) {
  done := make(chan bool)
  defer close(done)
  go s(done)
  for i := 0; i < count; i++ {
    <- done
  }
}
```

```
package map_reduce

type Iteration func(k, x interface{})

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

```
package map_reduce

func Each(c interface{}, f Iteration) {
  switch c := c.(type) {
  case []int:                  WaitCount(len(c), func(done chan bool) {
                                   for i, v := range c {
                                       f.apply(i, v, done)
                                   }
                               })

  case map[int] int:           WaitCount(len(c), func(done chan bool) {
                                   for k, v := range c {
                                       f.apply(k, v, done)
                                   }
                               })

  }
}
```

```
package map_reduce

type Results chan interface{}

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

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

```
package map_reduce

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

func (t Transformation) GetValue(x interface{}) interface{} {
  return t(x)
}
```

```
func Map(c interface{}, t Transformation) (n interface{}) {
  var i Iteration
  switch c := c.(type) {
  case []int:                m := make([]int, len(c))
                                 i = func(k, x interface{}) { m[k] = t.GetValue(x) }
                                 n = m

  case map[int] int:         m := make(map[int] int)
                                 i = func(k, x interface{}) { m[k] = t.GetValue(x) }
                                 n = m
  }
  if i != nil {
    Wait(func(done chan bool) {
      Each(c, i)
      done <- true
    })
  }
  return
}
```

```
package main
import "fmt"
import . "map_reduce"

func main() {
  m := "%v = %v, sum = %v\n"
  s := []int{0, 1, 2, 3, 4, 5}
  sum := func(x, y interface{}) interface{} { return x.(int) + y.(int) }
  d := Map(s, func(x interface{}) interface{} { return x.(int) * 2 })
  x := <- Combination(sum).Reduce(s, 0)
  fmt.Printf("s", s, x)
  x = <- Combination(sum).Reduce(d, 0)
  fmt.Printf("d", d, x)
}
```

produces:

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