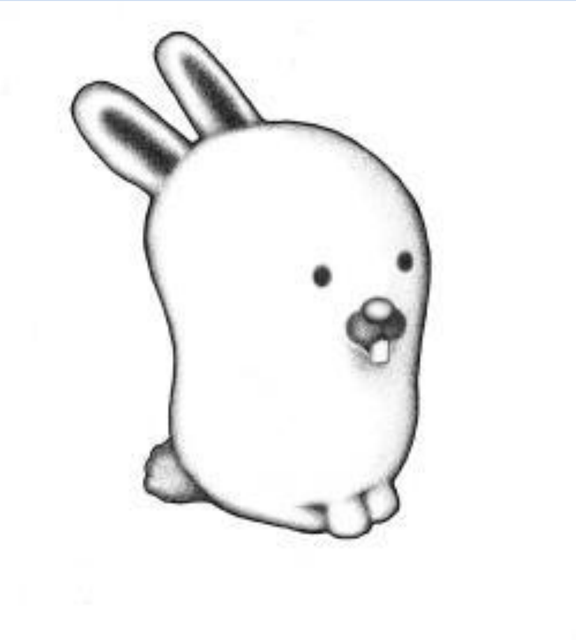
Google's Go Programming Language

Yossi Gil



What is it?

- Google's newly released Programming Language.
  - Factor: An extensible Programming Language (Slava Pestov 2008)
  - Many other “internal languages”
  - Not: Go! (a Prolog like obscure agent-based programming language).
- **Compiled**, **concurrent**, **imperative**, **structured**, **GC**, **{}**
- **By**: Ken Thompson (B,C), Rob Pike (Limbo), 2007-
  - **Other Google Purchases:** Udi Manber, Bram Moolenaar (vim), Vint Cerf, Larry Brilliant, Michael Burrows (BW), Joshua Bloch (Java), Ed Lu(astronaut)

Example

```
package main
import ("os"
        "flag”)
var nFlag = flag.Bool("n", false, `no \n`)
func main() {
    flag.Parse()
    s := "";
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 { s += " " }
        s += flag.Arg(i)
    }
    if !*nFlag { s += "\n" }
    os.Stdout.WriteString(s);
}
```

Philosophy: No Bookkeeping

- **Programming today:** bookkeeping, repetition, and clerical work.
- Dick Gabriel (IBM):

“Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication bought such noise?”

- **New Abstractions:** Good.
- **New Verbosity**: Bad.

Why?

- No new major systems language in a decade.
- But much has changed:
  - - sprawling libraries & dependency chains
  - - dominance of networking
  - - client/server focus
  - - massive clusters
  - - the rise of multi-core CPUs
- Major systems languages were not designed with all these factors in mind.

Objectives

- The efficiency of a statically-typed compiled language with the ease of programming of a dynamic language.
- Safety: type-safe and memory-safe.
- Good support for concurrency and communication.
- Efficient, latency-free garbage collection.
- High-speed compilation...

Design Principles

- **Orthogonality**: A few orthogonal features work better than a lot of overlapping ones. (e.g. no ”while” command)
- **Simple, Regular Grammar:** Few keywords, parsable without a symbol table.
- **Reduced typing.** Let the language work things out. No stuttering; don't want to see

  foo.Foo *myFoo = new foo.Foo(foo.FOO_INIT)
- **Reduce typing.** Keep the type system clear. No type hierarchy. Too clumsy to write code by constructing type hierarchies.
- **Safety**: GC and Memory (no pointer arithmetic)
- **OO?:** Yes, but the ”Google” way...

Idea: Escape from Type System Tyranny

- Const in C++ as an example
  - well-intentioned but awkward in practice
- Type Hierarchy
  - Types in large programs do not easily fall into hierarchies
- You can be safe or productive, not both

Give us good old C back, but better

Why New Language?

New libraries won’t help

Adding anything will not enable us to reduce and simplify the language

Parenthesis

- It is a ”curly braces language”, just like C, C++, Java, C# and many others.
- But, the syntax of conditionals and iteration is simplified: parenthesis are optional, curly brackets are mandatory

```
for i := 0; i < flag.NArg(); i++ {
    if i > 0 {
        s += Space
    }
    s += flag.Arg(i)
}
```

Semicolons

- No semicolons...
  - except in:
    - for i := 0; i < 10; i++ {...}
    - if v := math.Pow(x, n); v < 5 {…}
- Improve on the synthesis approach:
  - Internally, the language uses semicolons.
  - They are added automatically for you.
  - CASE tool will remove them from text.
- Necessary if you have two statements on the same line.

No Need for Type Declaration

Equivalent declerations:

```
var s string = ”Hello”;
var s = ”Hello”;
s := ”Hello”; //Initialization operator
```

Declering Constants:

```
const space = ” ”;
```

Decleration inside a for loop:

```
for i := 1; i < 100; i++
```

Declares i to be a new variable of type integer

Primitive Types

- Boolean: **`boolean`**
- Integral: **`int8`**, **int16**, **`int32, int64, int`**
  - **`int`** is 32 bits or 64 bits, but it is always distinct from **`int32`** and **`int64`**
- Unsigned: **`uint8`**, **uint16**, **`uint32, uint64, uint`**
  - **`uint`** is 32 bits or 64 bits, but it is always distinct from **`uint32`** and **`uint64`**
  - **`byte`** **is alias for** **`uint8`**
- Float**`: float32, float64, float`**
  - **`float`** is 32 bits or 64 bits, but it is always distinct from **`float32`** and **`float64`**
- Complex**`: complex32, complex64`**
  - **`complex`** is 32 bits or 64 bits, but it is always distinct from **`complex32`** and **`complex64`**
- **`uintptr`** an unsigned integer large enough to store the uninterpreted bits of a pointer value
  - Any pointer or value of type uintptr can be converted into a Pointer and vice versa.

String Type

Similar to immutable array of bytes

Partial inspection

No partial modification

Arrays and Slices

- Indices, just like C, are 0,..,Size
- Multidimensional, just like C, unlike Java.
- Size must be known at compile time
- No pointer arithmetic is allowed?
  - Why???
- Slice types: reference to a contiguous segment of an array and contains a numbered sequence of elements from that array.

Arrays and Slices- example

```
func f(a [10]int) { fmt.Println(a) }

func fp(a *[10]int) { fmt.Println(a) }

func main() {

    var ar [10] int

    f(ar)    // passes a copy of ar

    fp(&ar)  // passes a pointer to ar


    var a []int //creating a slice

    a = ar[7:9]

}
```

Struct Types and Methods

```
type Point struct { x, y float64 }

// A method on *Point
func (p *Point) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y)
}

p := &Point{ 3, 4 }
fmt.Print(p.Abs())  // will print 5
```

Interfaces

```
type Abser interface {
    Abs() float64
}
type Vertex struct {
    X, Y float64
}
func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

```
func main() {
    var a Abser
    v := Vertex{3, 4}
    a = &v // a *Vertex implements Abser
    a = v  // a Vertex, does NOT implement Abser
    fmt.Println(a.Abs())
}
```

Other Type Constructors

- Map
- Function
- Channel (for parallel programming)
- No union
- No inheritance (struct may implement interfaces)

Differences from C++

- No constructors
- No destructions (thanks to garbage collection)
- No pointer arithmetic
- Arrays are first class values
  (passed by value to functions)
- No implicit type conversion
  All conversions must be explicit
- **`nil`** "belongs" to all pointer types

White Lie Above

Constants (and literals) are untyped!

```
const b = 3
```

Gives the literal “3” a symbolic name “b”

But, “3” is untyped!

```
var a uint
const b = 3;

...
f(a + b) // untyped numeric constant “3" becomes typed as uint
```

goroutine

```
func IsReady(what string, minutes int64) {
    time.Sleep(minutes * 60*1e9)
            // Unit is nanosecs.
    fmt.Println(what, "is ready")
}
go IsReady("tea", 6)
go IsReady("coffee", 2)
fmt.Println("I'm waiting...")
```

Prints:

```
I'm waiting...   (right away)
coffee is ready  (2 minutes later)
tea is ready     (6 minutes later)
```

Channels

```
func pump(ch chan int) {
    for i := 0; ; i++ { ch <- i }
}
func suck(ch chan int) {
    for { fmt.Println(<-ch) }
}
ch1 := make(chan int)
go pump(ch1)        // pump hangs; we run
fmt.Println(<-ch1)  // prints 0
go suck(ch1)  // tons of numbers appear
```