Programming Paradigms CSI2120 – Winter 2018

Jochen Lang
EECS, University of Ottawa
Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

System Programming: Go

- Design of the Language
- Keywords and Types
- Variables and Functions
- Structured types

Introduction to Go

- **Started as a part-time project of three programming and OS veterans at Google**
  - Robert Griesemer (Java HotSpot Virtual Machine)
  - Rob Pike (Part of Unix team at Bell Labs)
  - Ken Thompson (Part of Unix team at Bell Labs, One of the Inventors of Unix, C and Plan 9)
- **Timeline**
  - November 10, 2009 officially announced for Linux and Mac with a BSD license
  - Windows port announced by November 22
  - First public release on January 8, 2010
- **Premier source of information is golang.org**

Go Programming Language

- **Go is designed to be the "C of the 21st century"**
- **Go unifies conflicting language features**
  - Efficiency, speed and safety of strongly typed languages (like C/C++)
  - Fast compilation and building (like Java/C#)
  - Ease of programming of a dynamically typed language (as Python)
- **Goal is to "make programming fun again"**
  - Type-safe
  - Memory-safe
    - Pointers but no pointer arithmetic

Paradigms in Go

- **Imperative Language**
- **Not quite object-oriented**
  - No classes and inheritance
  - But with interfaces and forms of polymorphism
- **Fundamental aspect of a functional programming**
- **Support for parallel and concurrent programming**
  - Systems programming
  - High-performance computing

Missing Concepts

- **No function or operator overloading**
  - simpler design
- **No implicit type conversions**
  - avoid bugs and confusion
- **No dynamic code loading or dynamic libraries**
  - avoid complex dependency on installed environment
- **No generics or templates**
- **No exceptions**
  - but alternative recover after panic
- **No classes and type inheritance**
- **No assertions**
- **No immutable variables**

Programming Go

- **Go compiler is available as binary and source for different platforms**
- **Two options**
  - gc-Go compiler
    - Go compilers and linkers are written in C
    - Named g and l with a corresponding number
  - gccgo
    - Go is an officially supported language since 4.6.0 in the gcc compilers
- **gc-Go**
  - FreeBSD 8 or later on amd64, 386, arm
  - Linux 2.6.23 or later with glibc on amd64, 386, arm
  - Mac OS X 10.6 or later on amd64, 386
    - Use of cgo: use gcc that comes with Xcode
  - Windows XP or later on amd64, 386 (installer available)
    - Use of cgo: use mingw gcc; cygwin or msys is not needed

IDEs for Go

- **LiteIDE**
  - Open source cross platform IDE
  - configurable builds (projects)
  - debugging
  - available for windows, linux, MacOS and OpenBSD.
  - https://sourceforge.net/projects/liteide/
- **Eclipse with goclipse plugin**
  - http://goclipse.github.io/
- **IntelliJ IDEA Plugin**
  - https://plugins.jetbrains.com/plugin/5047-go-language-golang-org-support-plugin
- **There is always emacs or vim**

Go Programming Environment

- **Code is generally very portable**
- **Go is set up to be easy to cross compile**
- **Go compiler uses 4 environment variables to control the compile**
  - `$GOROOT` root of the go tree installation
  - `$GOARCH` processor architecture of the target machine
  - `$GOOS` target machine OS
  - `$GOBIN` location of compiler/linker binaries

Go Programming Environment

- **Go expect your code to adhere to a particular directory structure**
  - Wokspace will have three directories
  - Executables go in ($MY_WORKSPACE)/bin/
  - Packages (libraries) go in ($MY_WORKSPACE)/pkg/
  - Source code goes at some level under ($MY_WORKSPACE)/src/

Elements of Go

- **Code is structured in packages**
  - The package main is the default package
    - all go code belongs to a package (similar to C++ namespaces)
  - Go uses Unicode (UTF-8) everywhere
  - Identifiers are case sensitive, begin with a letter or _ followed by 0 or more letters or Unicode digits
- **25 keywords (that's all!)**
- **36 predefined identifiers**

Keywords and Predefined Identifiers

- **Keywords**
  - `break default func interface select case defer go map struct chan else goto package switch const fallthrough if range type continue for import return var`
- **Predefined identifiers (functions and types)**
  - Constants
    - `false, true, iota`
  - Functions
    - `append cap close complex copy delete imag len make new panic` *`print println`* `real recover`
  - Types
    - `bool byte complex64 complex128 error float32 float64 int int8 int16 int32 rune int64 string uint uint8 uint16 uint32 uint64 uintptr`

Hello World in Go

```
package main // default package

import "fmt" // import of a package

func main() {
     fmt.Println("Hello World")
}
```

- **Notes**
  - Comments use // for one-line comments
  - Section can be commented out with /* */
  - No special character to end a line (semicolon is required to put several statements in one line)
  - Exported functions from a package are always capitalized
  - Entry point main has no arguments and no return

Style is Defined with the Language

- **Most languages have no particular style defined**
  - often different styles develop in different communities
  - often companies have a style guide
  - makes code harder to reuse amongst different styles
  - example: C/C++
- **Java has some recommendation**
  - not always followed
  - different style guides
- **Go defines the universal go style**
  - It makes disregarding some preferred formatting a compile error
  - It provides a formatter: `go fmt`

Defining and Using Variables and Functions in Go

```
package main

import "fmt"

const pi= 3.1416
var x int = 5 // global variable

func main() {
        var ( // grouping or “factoring the keyword”
                a float64 = 8.8
                b float64
        )
        b= foo(a)
        fmt.Printf("Result: %f", b)
}

func foo(z float64) float64 {
        u := 3.3 // intializing declaration
        return u*z
}
```

Variables

- **Variables can be defined in two different ways**
  - defined and default initialized
    - `var x int` makes x a default (0) initialized integer
  - initializing definition
    - `x := 1` makes x an integer and initializes it from the assignment
  - sometimes we want to combine it
    - `var x int16 = 5` makes x an initialized int16
- **Can factor (group) variable definitions**
  - most often used with globals

```
var (
      a float64 = 8.8
      b int
)
```

Functions

- **Types of arguments and returns are at the end**
- **Multiple arguments and multiple returns**
- **General syntax**

```
func functionName( parameter_list )
   (return_value_list) {
    function-body
    return [return_values]
}
```

- **where**
  - `parameter_list` is of the form `(param1 type1, param2 type2, …)`
  - `return_value_list` is of the form `(ret1 type1, ret2 type2, …)` or is unnamed `(type1, type2, …)`
  - `[return_values]` either have to be specified if unnamed or can be associated by name

Example: Function with Multiple Returns

```
func main() {

   var (
        s int
        d int
    )
   s, d = plusminus(7,9)
   fmt.Printf("Result= %d and %d", s , d)
}


func plusminus(a int, b int) (sum int, difference int) {

   sum = a+b
   difference = a-b

   return
}
```

Functions with an Error Code

- Errors are signalled with error codes

```
func bmi(height float64, weight float64) (float64,
   bool) {

   if height > 0.0 {
       return weight / (height*height), true
   } else {
       return 0.0, false
   }
}
```

- Test the returned error code

```
if value, ok := bmi(1.50, 55); ok {

   fmt.Printf("BMI: %f\n", value)
}
```

Lambda Functions – Closure

- **Lambda functions are like variables**
  - Example: Callback

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

func Distance(p1 Point, p2 Point) (distance float64){
   distance = math.Sqrt(math.Pow(p1.x - p2.x, 2.0) +
       math.Pow(p1.y - p2.y, 2.0))
   return
}

func calc(p1 Point, p2 Point,
       d func(Point, Point)(float64))(float64) {
   return d(p1,p2)
}
```

Anonymous Lambda Functions

- **Lambda function are anonymous and can either be called directly or assigned to a variable, passed to a function**

```
func main() {

   a := Point{2.,4.}
   b := Point{5.,9.}

   dist := calc(a,b,Distance)

   fmt.Printf("result= %f\n", dist)

   dist = calc(a,b,
     func(p Point, q Point)float64{
        return math.Abs(p.x-q.x)+math.Abs(p.y-q.y)
    })

   fmt.Printf("result= %f\n", dist)
}
```

Variables and Type Categories

- **Three categories of data types**
  - elementary or primitive
    - int, float, bool, string
  - structured or composite
    - struct, array, slice, map, channel
  - interfaces
    - only describe the behavior of a type

  - Variables are initialized to their zero type by default
  - Structures have a default value of nil

Pointers and Structures

```
type Point struct {
   x int
   y int
}

func main() {

   pt := Point{8, 1}
   complement(&pt)

   fmt.Printf("Result= %d and %d\n", pt.x , pt.y)
}

func complement(p *Point) {
   p.x, p.y = -p.y, -p.x
}
```

- **Note:**
  - A dereference operator is here not required
    - no pointer arithmetic

Arrays in Go

```
package main
import "fmt"

func mean(tab [5]int) (meanVal float64) {
        // for index, value := range collection
        for _, value := range tab {
                meanVal+= (float64)(value)
        }
        meanVal /= (float64)(len(tab))
        return
}

func main() {
        var table = [5]int{3, 4, 8, 9, 2}
        m := mean(table) // pass by value
        fmt.Printf("result= %f\n", m)
}
```

Slices in Go

- **A slice is a reference to a contiguous region in memory**
  - it refers to contiguous elements in an array
    - as such it “shares” the elements
- **Slices are commonly used in Go instead of copying arrays**
- **A slice has a given dimension and capacity**
  - the capacity is determined by the underlying array
    - a slice can not be bigger than the underlying array
- **Creating a slice**
  - `var slice []int = array[start:end]`
  - `slice := make([]int, 10, 100)`
- **Familiar from python**

Example: Slices

```
func mean(tab []int) (meanVal float64){
   // for index, value := range collection
   for _, value := range tab {
       meanVal+= (float64)(value)
   }
   meanVal /= (float64)(len(tab))
   return
}

func main() {

   var table = [5]int{3, 4, 8, 9, 2}

   m := mean(table[:]) // all elements
   fmt.Printf("result= %f\n", m)
   m = mean(table[2:]) // elements 2 to the end
   fmt.Printf("result= %f\n", m)
   m = mean(table[1:3]) // 2 elements from 1 up to 3
   fmt.Printf("result= %f\n", m)
}
```

Summary

- **Design of the Language**
  - Multi-paradigm
  - Targeted at system programming
  - Efficient to code but also efficient to run
- **Keywords and Types**
- **Variables and Functions**
  - main, multiple returns, error codes
- **Structured types**
  - structures, arrays, slices