The GO Language
A Language Introduction and Overview

by Rich Coe
rich.coe@marquette.edu

Prof Doug Harris
Summer 2010

Table of Contents

Introduction

Go is a new experimental system programming language under active development. This document was written in July, 2010 and captures the current active design. Go is defined by the project as a fast to compile, type safe, memory safe, concurrent programming language. The current license is open source, Creative Commons Attribution 3.0 License. A key philosophy of Go is that as clock frequences are no longer increasing as fast as they used to, the number of cores per socket are increasing, but written software and development are not any faster [faq].

The Go developers believe feel that it would beneficial to develop a new concurrent garbage-collected language that will assist the developer and address the following areas[faq]:

- Decrease the compile time for large programs
- Provide a model for software construction that makes dependency analysis easy and avoids much of the overhead of C-style include files and libraries.
- A type system that has no hierarchy, so no time is spent defining the relationships between types. Although Go has static types, the language attempts to make types feel lighter weight than in typical OO languages.
- A fully garbage-collected language.
- Fundamental support for concurrent execution and communication.
- An approach for the construction of system software on multicore machines.

References

[basic-cgo]	Ostsol's Blog http://cheesesun.blogspot.com/2009/12/basic-cgo.html
[cgo]	cgo command http://golang.org/cmd/cgo/
[course1]	The Go Programming Language, Part 1, Rob Pike http://golang.org/doc/GoCourseDay1.pdf
[course2]	The Go Programming Language, Part 2, Rob Pike http://golang.org/doc/GoCourseDay2.pdf
[cox]	Russ Cox Blog: http://research.swtch.com/search/label/Go
[effgo]	Effective Go: http://golang.org/effective_go.html
[gccgo]	gccgo command http://golang.org/doc/gccgo_install.html
[home]	Go Language Home Page: http://golang.org
[ibm]	David Bacon's Publications 'recycler algorithm' http://www.research.ibm.com/people/d/dfb/papers.html

[pkg]	Package Documentation: http://golang.org/pkg
[plan9man]	Plan 9 compiler man page http://plan9.bell-labs.com/magic/man2html/1/2c
[spec]	Go Language Specification: http://golang.org/doc/go_spec.html
[talk09]	Google talk http://golang.org/doc/talks/go_talk-20091030.pdf
[tutor]	Go Tutorial: http://golang.org/doc/go_tutorial.html
[unsafe]	Package **unsafe** Documentation: http://golang.org/pkg/unsafe/

Language Basics

a. Statements

Go defines the typical statement types, assignment, if, for, and switch. Following Go's minimal style, there is no while statement.

b. Assignment Statement

Assignment follows the familiar programming paradigm: variable equals expression. The programmer can additionally declare and assign a variable with the Go paradigm: variable colon equals expression.

Assignment:

```
myvar = a *m + b;
```

Declaration and Assignment:

```
myvar := a * m + b;
```

c. for Statement

There are three forms of **for**.

(1) for initializer-expr; conditional-expr; post-expr BLOCK

```
for i = 0; i < 10; i++ {
    sum = sum + i
}
```

(2) for conditional-expr BLOCK

```
for  i < 10 {
    sum = sum + i;
    i++;
}
```

(3) for BLOCK

```
for {
```

```
    // for-ever
}
```

d. switch Statement

The switch statement is different from C's or java's switch in that it allows case expressions to be any expression to be evaluated, not just constants or integers. Each case is evaluted top to bottom until a match is found.

Switch Usage:

```
switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
}
```

e. Functions

Function declarations in Go, like other languages, defines the ability to into divide written code into named procedures. A function declaration is defined by the keyword **func**, followed by an identifier and a function signature. Unlike 'C' or java, you can have multiple return values from a function. Each of the return values can be named and referenced within the function.

```
Func foobar(inp Param, buf []byte) (n int, err os.Error) {
     nr, err = inp.Opmethod(buf)
     return
}
```

Note that unlike other languages, Go enforces a strict formmating restriction in the **func** definition that the opening curly brace '{' has to be at the end of the line.

f. Data types, variables, and constants

Variables can be explicitly declared with **var**, but as shown above in the Assignment Statement, variables can be implicit declared by with the **:=** operator. Implicit declaration works, because the type of right-hand side always matches the type of the left. Additionally, there is no implicit conversion from liked-type variables. This means if the result of a right-hand operation must always match the left-hand destination, or explicity cast the type of the result to match the destination. Declared variables with a type may have an optional initializer.

Integer numeric types: **uint8**, **uint16**, **uint32**, **uint64**, **int8**, **int16**, **int32**, **int64**

Floating-point numeric types: **float32**, **float64**

Complex types: **complex64**, **complex128**

Pre-declared numeric types: **uint**, **int**, **float**, **complex**, **uintptr**

var usage:

```
var myvar int32;
var somevar int = 5;
var a, b, c float;
```

Note that in a assignment a type of **int** is not the same as a type of **int32** even though the integer is represented by the same sized integer instance. To assign an integer of **int32** to an **int** will require an explicit type cast to the type of the result.

Type usage:

```
myvar = int32( somevar );
```

The special type **uintptr** is an unsigned integer that can hold the 'uninterpreted' bits of a pointer value. This type is used for storing pointer values and for low-level operations that violate the type system. More information can be found in Package **unsafe**. [unsafe]

A constant is declared similar to variable declaration but with the **const** keyword.

Constant usage:

```
const small_pi float = 3.14;
const birds, cats, dogs = 7, 9, 11;
```

The **iota** keyword is used within a const declartion. It make it easy to declare successive untyped integer constants. **iota** is reset to zero at each **const** declaration.

Iota usage:

```
const (
    Red = iota
    Orange
    Yellow
    Green
    Blue
    Indigo
    Violet
    numColors
)
```

g. structs

A struct is defined by a sequence of fields, each field consisting of a name and a type. The struct is declared by making it a type, and the struct can only be seen outside it's package if its typename is capitalized, otherwise the struct is private to the package it is declared in. Similarly, any field in the struct is private to the package it is declared in unless the field name is capitalized.

Public struct declaration:

```
type Buffer struct {
```

```
        Buff []byte;
        Name string;
    }
```

Private struct declaration:

```
    type buffer struct {
        buff []byte;
        name string;
    }
```

Fields of a struct are accessed with the selector operator '.'. Given a declared variable **buf** of type **Buffer** as above, access the fields of **buf** take the form **buf.Buff** or **buf.Name**.

Within a struct, anonymous fields can be declared just by sepecifying the type name. It acts as if the entire anonymous type has been inserted into the declared struct. This is a simple way to imbed or encapsulate types within a struct.

Anonymous type usage:

```
    type Anon struct {
        ax, ay int
    }
    type Outer struct {
        Anon;
        bx, by float;
    }
```

The struct *Outer* acts as if it contains four fields, `ax, ay, bx, by`. The anonymous type also can be directly accessed as a field whose name is its type. For example:

```
    out :=  Outer{  Anon{ 1, 2},  32.5,  5.3 };
    fmt.Println( out.ax, out.ay, out.bx, out.by );
    fmt.Println( out.Anon );
```

Prints:

```
    1 2 32.5 5.3
    {1 2}
```

Any named type or a pointer to to a a type may be used in an anonymouse field and appear at any location in the struct. The rules for two or more fields with the same name within the struct follow these rules: [course2]

(1) An outer name hides an inner name. This provides a method to override a field.

(2) If the same name appears twice at the same level, it is an error if the name is used by the program. No error exists if the name is not used.

No rules resolve the amibuguity, it must be fixed by the programmer.

h. Allocation with make and new

The built-in function **make** is used to create the reference types slices, maps, and channels. These will be discussed later in Key Features. **make** returns a value of the declared type, and not a pointer to the type.

The built-in function **new** is used to create a pointer to a type. **new** allocates memory for the type specified, initializes the contents to zero, and returns a value of pointer to the type.

New usage:

```
type Sometype struct { next, prev *Sometype; content int }
ps = new(Sometype);
```

i. Built-in standard library

Go has an extensive built-in package library documented at [pkg]. Some of the useful supplied packages are

- *bufio* package which implements buffered i/o
- *container* packages which implement *heap, list,* and *vector*
- *crypto* packages which implement ciphers and hashes like *aes*, *blowfish*, *md4*, *md5*, and *sha1*
- *fmt* package which implements formatted i/o
- *http* package which implements parsing HTTP requests, replies, URLs and more
- *math* package which implements basic math constants and functions
- *strings* package which implements basic string manipulation

j. Methods and Interfaces

Go has no classes. Methods can be attached to any type. The method is declared separate from the type declaration as a function with an explicit receiver. The type of a reciver on a method can be either pointer to type `(name *Type)` or just the type `(name Type)`, but the type itself cannot be a pointer type.

```
func (out *Outer) Magnitude()  float  {
     return math.Sqrt((out.bx * out.bx) + (out.by * out.by))
}
```

An anonymous field of the type inherits the methods of the anonymouse field. This allows a simple way to emulate the effects of subclassing and inheritance. [course2] For example:

```
type Point struct { x, y float }
func (p *Point) Abs() float { ... }
type NamedPoint struct {
     Point;
     name string;
}
n := &NamedPoint{Point{3, 4}, "Pythagoras"};
```

```
fmt.Println(n.Abs()); // prints 5
```

The visibility of fields and methods have these rules [course2]:

1. Go has package scope
2. Capitalization determines whether it is exported or local.
3. Structs in the same package have full access to that packages fields and methods.
4. The local type can export its fields and methods.
5. No subclassing like C++, so there is no notion of `protected.`

The definition of interfaces in Go is different from the definition in other languages. The interface type is completely abstract and it implements nothing. The interface type defines a set of properties that an implementation must provide, a set of methods implemented by some other types.

k. Go features to watch

Go has features that are different from 'C', and other languages.

The end of line is sufficient for ending a statement and the semi-colon is optional. A semi-colon is required when constructing a multi-statement line, for statement separation.

There is no pointer arithmetic or math operations on pointers.

Capitalization of a variable, function, struct, or struct member exports that definition from the package.

There is no function or operator overloading.

Strict syntax specification

Key Features

a. Arrays and Slices

Go defines arrays as a fixed allocated unit defined at compile time. If you assign an array to another array, the entire array gets copied. The size of an array is part of its type.

Array Declaration:

```
var arrname [10]int;
```

Array Usage:

```
arr1 = arr2;
```

Slices are the preferred Go array idiom. Slices are a built-in reference type. When assigning one slice to another, both slices refer to the underlying data. You can discover the current used amount of a slice with the built in function len() and the total capacity with cap().

Slices are created with **make**. The syntax to **make** takes a slice type, a length expression, and an optional capacity expression.

Slice Declaration:

```
var  slname []int;
```

Slice Usage:

```
slname = make([]int, 9, 81);
n, err = fb.Read(slname)
```

Slice Construction:

```
slv = make([]float, l, cap)        // length:l and capacity:n
mlv = make([]int32, l)             // length and capacity l
```

b. Maps

Maps are a built-in reference data type for building associated arrays, like hashes in perl or hash maps in java. The key can be of any type which the equality operator is defined.

Maps are created with **make**. The syntax to **make** takes a map type, and an optional initial length expression.

Map Declaration:

```
var commit map[string] string;
```

Map Usage:

```
commit = make(map[string] string, 10);
commit[“bz231”] = “trunk/src/file.c”
```

c. Defer Statement

The defer statement invokes an expression whose execution is deferred until the surrounding functions returns. Multiple defer operations in a function are executed last in, first out (LIFO).

The following example show how defer could be used to closing the file descriptor upon the return of the *data* function.

```
func data(name string) string {
     file := os.Open(name, os.O_RDONLY, 0);
     defer file.Close();
     data := io.ReadAll(f);
     return data;
}
```

d. Multiple return values

One of Go's unique features is that functions and methods can return multiple values. A useful application of

this feature is to return a count and an error. This idiom is used in many of Go's i/o packages. For example, the signature of `*File.Write` in package `os` is:

```
func (file *File) Write(b []byte) (n int, err Error)
```

and it returns the bytes written and a non-nil `Error` when `n != len(b)`. The current Go documentation does not describe the return values for `err`.

e. Named Result Parameters

The return results of a Go function can be given names just like input parameters. And the result parameters can be used anywhere in the function.

```
func smpl(buf []int, op flag) (value int, err Error) {
[...]
}
```

f. Goroutines

Goroutines convey the simple model of a functions running in parallel with other functions sharing the same address space. Preceeding any function call with the keyword **'go'** will invoke the function on a separate stack and have it run to completion.

g. Channels

Channels are Go's built-in communication and synchronization reference type. Channels can be either buffered or un-buffered. A receiver on the channel always blocks waiting until there is data to receive. The sender blocks on the unbuffered channel until the reciever has received the value. A sender will only block on the buffered channel until the value has been copied into the buffer. If the buffered channel is full, a sender will block until a receiver has made space by receiving a value.

Channels are created with make. The syntax to make takes a channel type, and an optional buffer size expression.

Channel Declaration:

```
uch := make(chan int);   // unbuffered channel of int
bch := make(chan int, 10); // buffered channel of int
```

Channel Usage:

```
uch := make(chan int);
go producer() {
     var i int;
     i = 0;
     for {
          i++;
          uch <- i;
     }
```

```
}()
// consumer
for {
     var cons int;
     cons <- uch;   // recv value from producer
     fmt.Printf("received val %d\n", cons);
}
```

h. Per source file init()

Each Go source file can define a **func** *init()* function that gets called once at process initialization. The *init* function is called after any variable declarations have been evaluated. An *init* function cannot be called by the code, and the code cannot create a pointer to the *init* function.

The documentation suggest that the *init* function be used to complete initializations that cannot be expressed in declarations, but also to verify or repair the correctness of the program state. [effgo#init]

Garbage Collection

The current garbage collector is a "mark-and-sweep collector. Work is underway to develop the ideas from the IBM's recycler garbage collector" [ibm] with the intent of building a efficient, low-latency concurrent collector [gotalk09]

Compiling with Go

a. Simple Go Program

```
 1  package main
 2
 3  import ( "fmt"; "os" )
 4
 5  func main() {
 6      sec, nsec, err := os.Time();
 7
 8      if nil != err {
 9          os.Exit(1);
10      }
11
12      startt := float64(sec) + float64(float(nsec) / float(1e9));
13      fmt.Printf("t:%f  sec:%d  nsec:%d  %f\n", startt, sec, nsec,
14                      float64(sec) + float64(float(nsec) / float(1e9)));
15  }
```

time-test.go

Line 1: Only one Go source file in a go program is declared as part of package main.

Line 3: Import packages for providing system functionality. In this case "fmt" for printing output, and "os" for access to the *Time()* function.

Line 5: Function *main.main* is the start of our program.

Line 6: Implicit declaration of *sec, nsec,* and *err*. Get the current time.

Line 8: Exit if an error occurred.

Line 12: Implicit declaration of *startt*: start time. Variable *sec* and constant *1e9* are *int* and have to be cast to *float* before effective division can be performed. The result is cast to *float64* to match the desired type of the result.

Line 13: Using package *fmt* to produce output using *Printf()*.

This program was written to debug a problem with computing and reporting the difference of the start and end time of a called function.

b. Compiling and running a Go program

Set the required environment variables ***GOROOT, GOOS, GOARCH, GOBIN,*** and ***PATH.***

```
export GOROOT=$HOME/src/go
export GOOS=linux
export GOARCH=amd64         ## for x86_64 platform
export GOBIN=$GOROOT/bin    ## personal preference
export PATH=$PATH:$GOBIN
```

go env variables

Run the Go compiler to compile the go program.

```
6g time-test.go
```

The Go compilers are named after the Plan 9 compiler naming scheme from which the Go compiler source originated. [plan9man] 6c is the c compiler for amd64 (x86_64) and 8c is the compiler for Intel386, therefore 6g is the Go compiler for producing amd64 output.

Run the Go linker to link the binary.

```
6l -o time-test time-test.6
```

The output produced by the Go compiler sets the file suffix to the name of the architecture, .6 in this case for amd64. The output is of the linker is specified with *-o* because the default ouptut of the linker is *6.out*.

Run the Go program.

```
myhost $ time-test
t:1279870904.512686  sec:1279870904  nsec:512686000  1279870904.512686
```

c. C language linking

Go provides the command line utility *cgo* [cgo] to let Go functions interface with C libraries, however it is not robustly documented. [basic-cgo] Another alternative is to use the gcc front end *gccgo* [gccgo]. A side effect of using 'C' libraries in either method, the developer will need to use the 'C' *free()* function to release any memory allocated by the 'C' library functions called, as any memory allocated by the 'C' runtime will not be garbage collected by the Go runtime.

Summary

Go is a new language under development. New features and enhancements are being discussed on the golang mailing list and added to the language on regular basis.

Sample Programs

a. Matrix Multiply

```
package main

import (
        "fmt"
)

func doMath() {
        //var array = [row][col]
        var intArray1 = [2][3]int{
                [3]int{0, -1, 2},
                [3]int{4, 11, 2},
        }

        var intArray2 = [3][2]int{
                [2]int{3, -1},
                [2]int{1, 2},
                [2]int{6, 1},
        }

//check that the # elements in the column of matrix 1 equals
//the number of elements in the rows of matrix 2
        if len(intArray1[0]) != len(intArray2) {
                fmt.Println("Aborting.\nMultiplication Of The Above Matrices Not Possible.")
        } else {
                fmt.Println("Matrix multiply  compute")
                var productArray = [len(intArray1)][len(intArray2[0])]int{}
                //Do Math
                //productArray = intArray1 * intArray2
                for j := 0; j < len(intArray2[0]); j++ {
                        for i := 0; i < len(intArray1); i++ {
                                s := 0
                                for k := 0; k < len(intArray2); k++ {
                                        s += intArray1[i][k] * intArray2[k][j]
                                }
                                productArray[i][j] = s
                        }
                }
                //Display Results
                var m, n int
                for m = 0; m < len(productArray); m++ {
                        for n = 0; n < len(productArray[0]); n++ {
                                fmt.Print(" ", productArray[m][n])
                        }
                        fmt.Println()
                }
        }
}

func main() {
        fmt.Printf("Matrix Math in Go\n")
        doMath()
}
```