Go!

Ryan Habibi and Ariel Webster

March 10, 2014

1 Introduction

The Go language was developed by Google in 2009 and primarily designed by Robert Griesemer, Rob Pike, and Ken Thompson. Go has strong ties to the C family of programming languages and because of this is easy to learn for many computer scientists with that background. We don't believe that Go has the staying power to become a permanent and widely used language for a few reasons. Mainly, although Go offers some excellent native concurrency support it fails to find a niche where it excels beyond industry standard languages. While Go is extremely powerful, even including the features to be a fully function language, it is still in it's infancy and fails to offer the robust experience of more mature languages. Even with these low expectations of adoption Go quickly became a favorite of ours and in the following report we will present the interesting points, limitations, and surprising advantages encountered while learning about this light weight programming language.

2 Go Syntax

2.1 Variables

Go offers strong and inferred variables. Variables can be declared in two ways: "var x = 2" and "x := 2." The former can create variables of strong or inferred type while the latter is only used to declare and initialize inferred type. Constants, arrays, and maps must be strongly typed this syntax can be applied to primitive types, which is why strongly typed primitives exist in the language.

After initialization the = is used to assign new values to a variable just as it would be in Java or C with == used for logical comparison. When converting types, as opposed to the syntax in C or Java where the type is put in parentheses before the variable x to be converted (Type)x, the variable is put in parentheses e.g. Type(x).

In general variables work similarly to other languages, especially C because Go also supports pointers. Go pointers use identical syntax to C, the & is used for referencing the address of a pointer and the * is used for referencing the content.

2.2 Control Statements

Go only has one loop structure, the *for* loop. This loop has several different uses and structures. First there is the generic *for* loop, which looks much like a *for* loop in C or Java, though without the parentheses. Similarly, Go allows programmers to iterate over a range of items in a list simulating a *foreach* loop with use of the *range* keyword. There is also a *for* loop, which functions as a *while* loop where only the condition is present in the *for* statement. Finally, there is a *forever* loop, which is just a *for* loop with no exit condition. For examples of all of these loops see EX_2_Loops.

If statements also have very similar syntax to C or Java, again without the parenthesis. Like the Go *for* loop, the *if* statement can also start with a variable initialization statement to be executed before the condition is checked. Variables declared in the initialization statement are still in scope within a subsequent else block.

Unlike in C and Java where conditional and loop statements with a single line of code to be executed can forgoe the braces in Go the braces, and their possition, are mantetory. The reason for this was described in the Go documentation as follows, "Mandatory braces encourage writing simple if statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a return or break." (http:\\golang.org\doc\effective_go.html#if)

For example code see EX_3_Conditional.

2.3 Complex Types

There are a variety of complex data types in Go. Arrays and maps are structured similarly to their counterparts in C and Java. There is a strict and unintuitive syntax for array declaration where the type of the array is on the right side of the = instead of the left, opposite usage for primitive variable declaration. These complex types should be declared using the *make* command, which is analogous to C *malloc*. Make includes parameters to define how much memory is to be used at initialization and how much to reserve for later. It is also worth noting that you can make key value pair maps, where keys are named making the mapping equivalent to a SQL style table.

Slices are a native type within Go that are a pointer to an array and a length which can be less than or equal to the length of that array. Slices also have their own built in append() and copy() functions, which in theory make them a dynamically sized array type, but in practice, because of the weak typing, this is difficult to implement,

as type mismatch errors are easily introduced to the code through the slices's infered type from the base array. In practice slices work as sub arrays and are useful for solving subset style problems. (See EX_4_ComplexTypes)

2.4 Functions

Functions are structured differently than in C or Java, with the parameters listed before the return type. More interestingly, functions in Go are able to return more than one value. A common convention in Go is to always include an error value return, which can be checked to insure correct execution.

Functions can also be closures, where the function can be declared and scoped inside another function and saved in a variable. This, along with recursion, define Go as a functional programming language.

2.5 Structs and Interfaces

The method receiver is as close as Go gets to Object Oriented Programming (OOP) style objects. A method receiver is an extra argument within a function declaration, added between the *func* keyword and the method name. The receiver is a single pointer to any primitive or struct available in scope, this allows the method to be called using "MyStruct.ThisMethod()". In addition to tying the method to the type this also adds a reference to the specific instance used to call the method allowing access to other object attributes and behaviors. Using this we can add methods to a struct to create more complete objects, as seen in traditional OOP languages.

An interface is a type defined by its methods. Meaning, any struct that defines the methods of an interface is said to satisfy to implement that interface. "The idea behind go interfaces is duck typing. Which simply translates into: If you look like a duck and quack like a duck then you are a duck. Meaning that if your object implements all duck's features then there should be no problem using it as a duck." (http:\\stackoverflow.com\questions\7042640\usage-of-interface-in-go)

2.6 Concurrency

Goroutines are light weight functins executing in parallel created by putting the keyword "go" before a function call and managed by Go at runtime. Channels allow for data communication between goroutines. The channel requires both goroutines to be ready to send or receive respectively, a feature which allows goroutines to synchronize without conditional variables or locks. The channel operator to both send and receive data is "<-", with the arrow pointing in the direction the data is flowing.

Channels are created with a *make* the same way a complex type is. In this case *make* has two parameters, one required and the other optional. There must by a type,

such as *int*, that the channel will be recieving. The second, optional, argument is a maximum buffer length. When a sender has completed all possible data transmission it can close a channel, at which time no more data may be sent to it from any sender. A sender/receiver can also check whether a channel has been closed. Closing a channel is not necessary, there will be no memory leaks if it is not done. Closing is, however, a method of indicating to the receiver that there will be no more information sent. There is no way to reopen a channel, to reestablish communication another channel must be created.

The *select* statement is one of our favorite features of Go. The *select* is structured much like a switch statement without a value to switch on. The *select* checks each case block on which there is a channel, if any of them is ready to either send or receive information it executes the corresponding code. If more than one is ready it chooses at random. If none are ready the goroutine will wait or execute the default code if it exists.

3 Thoughts on Go

The variety of variable declaration statements, which can be used interchangably, is a detriment in the ease of adoption of Go. Unlike strongly or weakly typed languages where there is a single syntax Go offers rigid flexibility i.e. there exist many ways to write the declaration statement but only one correct way in each case dependant on usage of strong or weak typing. For example, in Java an integer is declared "int x = 5; , while an integer array is declared "int [] x = {5, 5};". Whereas in Go the side of the = the chosen type is on changes. In Go an integer can be declared "var x int = 5", an array declared using the same syntax, "var x [2] int", cannot be initialized in the same line. To declare and initialize the array the syntax must be "x:= [2]int{5,5}". This inconsistency is a weakness as the programmer must remember each case. True, an experienced Go programmer would not have trouble remembering the varying syntax rules, but this inconsistency does not smooth the path to adoption. (Demonstrated in EX_4)

Another undesirable aspect of Go syntax is that all variables once declared must be used, if only printed, or the code will throw an error. While we can see how this might be useful for space efficiency it can be annoying when writing code, whether the programmer is rearranging code and ensuring it will still run before adding a variable implementation back in or planning later variables that are not in use yet. As an interesting side note, there is a special syntax for for loops where, *int i* is declared at the beginning of the loop which is to replace *int i* with an "_" to signify the statement is not needed. We mostly encountered this error while debugging code, which forced us to either include extra code or clutter the output space. This same error is thrown in other cases as well such as unused package references and library includes.

Use of braces in Go is very similar to that found in Java or C, that is they are

required for scope of methods and functions as well as other control structures. One difference is the requirement that the brace be on a certain line depending on the context. For example, when used to open a function the brace must be on the same line as the function declaration or errors are generated as if the statements in the function are out of scope. Similarly, when using an if...else if...else the $elseif$ and $else$ statements must be on the same line as the closing brace from the previous component of the statement.

Go provides an interesting mechanism for custom error throwing and debugging in the form of the $defer$, *panic*, and *recover* functions. The functions work individually but when used together can create a comprehensive error handling flow. The $defer$ function is the most flexible and offers an interesting control structure.

$Defer$ can be used to push the execution of a function to after the end of the execution of the current scope. This can be useful to chain functions together while not cluttering key points of the code such as return statements.

The *panic* function allows a custom error to be thrown at any time. When debugging this is helpful because it can exit code at appropriate times with descriptive messages (provided to the function as a parameter).

The *recover* function can be used to prevent code from exiting when an error or panic is thrown, but isn't very useful alone since the error exits the current scope without executing the recovery call. This is why combining *recover* and $defer$ can be very useful. A deferred recovery will execute at the end of the current scope before the error is passed up to the calling function thereby preventing the program from exiting. When also used with *panic* this testing suite can be leveraged for code of any complexity and is easy to learn.