Alex Brisbois, Jessica Forrett

CSCI 324

April 25th, 2017

The Language of Go

Evolution of the language (Antecedents, language paradigm):

Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the whiteboard on September 21, 2007. Within a few days the goals had settled into a plan to do something and a fair idea of what it would be. Design continued part-time in parallel with unrelated work. By January 2008, Ken had started work on a compiler with which to explore ideas; it generated C code as its output. By mid-year the language had become a full-time project and had settled enough to attempt a production compiler. In May 2008, Ian Taylor independently started on a GCC (the widely used GNU compiler) front end for Go using the draft specification. Russ Cox joined in late 2008 and helped move the language and libraries from prototype to reality. Go became a public open source project on November 10, 2009.

Go was born out of frustration with existing languages and environments for systems programming. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java. Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multi-core computing. Finally, working with Go is intended to be *fast*: it should take at most a few seconds to build a large executable on a single computer. To meet these goals required addressing a number of linguistic issues: an expressive but lightweight type system; concurrency and garbage collection; rigid dependency specification; and so on. These cannot be addressed well by libraries or tools; a new language was called for.

Go is mostly in the C family (basic syntax), with significant input from the Pascal/Modula/Oberon family (declarations, packages), plus some ideas from languages inspired by Tony Hoare's CSP, such as Newsqueak and Limbo (concurrency).

Go attempts to reduce the amount of typing in both senses of the word. Throughout its design, the designers of Go have tried to reduce clutter and complexity. There are no forward declarations and no header files; everything is declared exactly once. Initialization is expressive, automatic, and easy to use. Syntax is clean and light on keywords. Stuttering (foo.Foo\* myFoo = new(foo.Foo)) is reduced by simple type derivation using the := declare-and-initialize construct. And perhaps most radically, there is no type hierarchy: types just *are*, they don't have to announce their relationships. These simplifications allow Go to be expressive yet comprehensible without sacrificing, well, sophistication.

Another important principle is to keep the concepts orthogonal. Methods can be implemented for any type; structures represent data while interfaces represent abstraction; and so on.

Elements of the Language (reserved words, primitive data types, structured types, unique elements):

The reserved words in Go are mostly par for a programming language: break, default, func, interface, select, case,map, struct, else, goto, package, switch, const,if, range, type, continue, for, import, return, and var. There is no “while”, as the for-loop in Go also serves as the while-loop. In addition to those words, there is “defer” which invokes a function whose execution is deferred to the moment the surrounding function returns. The word “go” itself is used for the execution of a function call as an independent concurrent thread of control, or *goroutine*, within the same address space. Also provided in Go is “chan”, which refers to a channel that provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type. The last known keyword is “fallthrough”, which refers to a "fallthrough" statement that transfers control to the first statement of the next case clause in an expression "switch" statement.

Some of the primitive data types in Go are again typical for a programming language: string, int, and bool. There are 10 additional subtypes for int: int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr. In addition, there are data types called byte and rune, which serves as aliases for uint8 and int32 respectively, with rune representing a Unicode code point. There are two each of float and complex data types: float32, float64, complex64, complex128. In addition to those data types, structured types can be made if needed for a program.

Syntax (Basic Structures, how the language handles functions and procedures):

Towards the beginning of a Go program, the import statements are similar to other languages, albeit slightly different in that the import statement is closed in parentheses as opposed to brackets. How you setup a structured type and a function is again largely the same compared to other languages. But, there is one crucial difference in syntax between Go and other languages: the data type follows the variable name, such as type List struct, or var word string. It may not be a huge change in the grand scheme of the program, and it does not take long to get used to declaring variables in that matter. Still though, it is a noticeable change if the programmer and reader are used to declaring a data type prior to the variable name in other languages. Another noticeable aspect of writing in Go is that when declaring a function, the data type of the return value comes after the input variables, such as func dictionary\_holder(word string, alphabet [26]string) bool. In addition, arrays are declared such that the bracketed value of array elements appears prior to the data type, such as [26]string. Also, assigning variables within a function is done with a colon and an equals sign together (:=), as opposed to just the equals sign for assignment statements in other functions. Otherwise, the only other noticeable aspect of syntax in Go is that semicolons are not really used, which does help in making it easier to write as there is no pressure in forgetting a single semicolon, as it would be par for the course with other languages..

Evaluation of Language’s Writability, Readability, and Reliability (use criteria from Ch. 1 of Sebesta):

Go as a programming language is easy to read, however it is mildly challenging to interpret at times. The readability is especially difficult when the programmer decides to make a structure out of a primitive data type, as this unusual meaning would cause confusion between the author and the reader. Another potential problem with readability is the assignment operator :=, which will declare the variable on the left hand side to be the same type as the value on the right hand side. Although this can be useful, it can potentially make it difficult for someone reading the program to figure out what type the newly declared variable is. Still, Go is a complex enough language that it allows for fewer needed statements. In terms of orthogonality, the primitive data types and constructs present in Go are relatively small, and every possible combination of them is legal and meaningful. The few primitive data types that serve as aliases (byte, rune) help make the language easier to understand. There are adequate facilities for defining data types and data structures. The syntax is slightly different than what most people are used to, such as how the for-loop also serves as the while-loop. Nevertheless, the syntax is easy enough to write and derive meaning from. Computations done in Go are relatively convenient to type out. It does a wonderful job with type checking, as well as pointing out other errors through exception handling. Overall, Go is a wonderful programming language to try out.

Since Go is a Free Open-Source programming language, then there was no monetary cost towards downloading it. Instead, the cost of training programmers in Go would be low, as there is not that many changes from the standard writing in C++. The cost of writing programs in Go would also be low, as they would be pretty close in purpose to the particular application specified. The cost of compiling and executing programs would again be low, considering how relatively easy it was to compile and execute, respectively. Another low cost would be for the language implementation system, since Go’s compiler/interpreter system is free. The cost of poor reliability would be low as well, as we did not experience any significant failures with the software. For the last consideration, the cost of maintaining programs would be low, as Go has been very readable.

Strengths:

Go is pretty easy to read and write, there is a fair balance between complexity and flexibility within the language, and the special features such as the goroutines and channels and writing methods for primitive data types allow for more efficient programming possibilities than other languages.

Weaknesses:

The only main issue with Go is getting used to the writing, as there are some differences between this and other languages in how it declares and assigns variables, alongside other minor details. Regardless, the programmer will be able to ease into the language after a while.

Overview of Programs Implemented (What language features each highlight, how it made it easy/hard to implement):

Aside from the “Hello, World” and Spell Checking programs that we were already mandated to do, we worked on a program called Planetarium. The basic idea was to have an interactive program that allows you to search for a planet within our solar system, and find out information regarding its current distance from the sun (taking into account the perihelion and aphelion distances), as well as retrieving the values of orbital period for all of the planets. We took some time searching for the formulas for calculating the distances and orbital periods of the main planets in our solar system, including the dwarf planet Pluto. In addition, for retrieving the orbital periods that we found online, we wanted to pull multiple values at once to make the process of fetching values more efficient. That is where we used a combination of channels and goroutines to fetch the orbital periods for the nine planets. However, when we initially tried to retrieve all nine orbital periods at once, it got said values but in a different order than what we expected when going through the order in which the planets were listed in the planet\_list array. To compensate for this but still retrieve multiple values, we set a buffer for the channel to 3, and retrieved 2 or 3 values at a time. That way, we could learn what order the values are being retrieved, which was generally getting the biggest value first and then going for smaller values. Another small problem we had was the program finishing before all of the goroutines were finished. We solved this relatively easily by adding channels with boolean values, and only printing out the last message to the user once all of the channels came back true, signalling that each goroutine had finished. The biggest challenge we faced was that Go had no graphical interface, which is why we had to improvise on how to make the program interactive, and why we added ascii art into our creative program to improve how it looked.