How a Go Program Compiles down to Machine Code

hackernoon.com/how-a-go-program-compiles-down-to-machine-code-e4532dc8b8ca

Here at Stream, we use Go extensively, and it has drastically improved our productivity. We have also found that by using Go, the speed is outstanding and since we started using it, we have implemented mission-critical portions of our stack, such as our in-house storage engine powered by gRPC, Raft, and RocksDB.

Today we are going to look at the Go 1.11 compiler and how it compiles down your Go source code to an executable to gain an understanding of how the tools we use everyday work. We will also see why Go code is so fast and how the compiler helps. We will take a look at three phases of the compiler:

- The scanner, which converts the source code into a list of tokens, for use by the parser.
- The parser, which converts the tokens into an Abstract Syntax Tree to be used by code generation.
- The code generation, which converts the Abstract Syntax Tree to machine code.

*Note: The packages we are going to be using (**go/scanner**, **go/parser**, **go/token**, **go/ast**, etc.) are not used by the Go compiler, but are mainly provided for use by tools to operate on Go source code. However, the actual Go compiler has very similar semantics. It does not use these packages because the compiler was once written in C and converted to Go code, so the actual Go compiler is still reminiscent of that structure.*

Scanner

The first step of every compiler is to break up the raw source code text into tokens, which is done by the scanner (also known as lexer). Tokens can be keywords, strings, variable names, function names, etc. Every valid program “word” is represented by a token. In concrete terms for Go, this might mean we have a token “package”, “main”, “func” and so forth.

Each token is represented by its position, type, and raw text in Go. Go even allows us to execute the scanner ourselves in a Go program by using the **go/scanner** and **go/token** packages. That means we can inspect what our program looks like to the Go compiler after it has been scanned. To do so, we are going to create a simple program that prints all tokens of a Hello World program.

The program will look like this:

We will create our source code string and initialize the **scanner.Scanner** struct which will scan our source code. We call **Scan()** as many times as we can and print the token's position, type, and literal string until we reach the End of File (**EOF**) marker.

When we run the program, it will print the following:

Here we can see what the Go parser uses when it compiles a program. What we can also see is that the scanner adds semicolons where those would usually be placed in other programming languages such as C. This explains why Go does not need semicolons: they are placed intelligently by the scanner.

Parser

After the source code has been scanned, it will be passed to the parser. The parser is a phase of the compiler that converts the tokens into an Abstract Syntax Tree (AST). The AST is a structured representation of the source code. In the AST we will be able to see the program structure, such as functions and constant declarations.

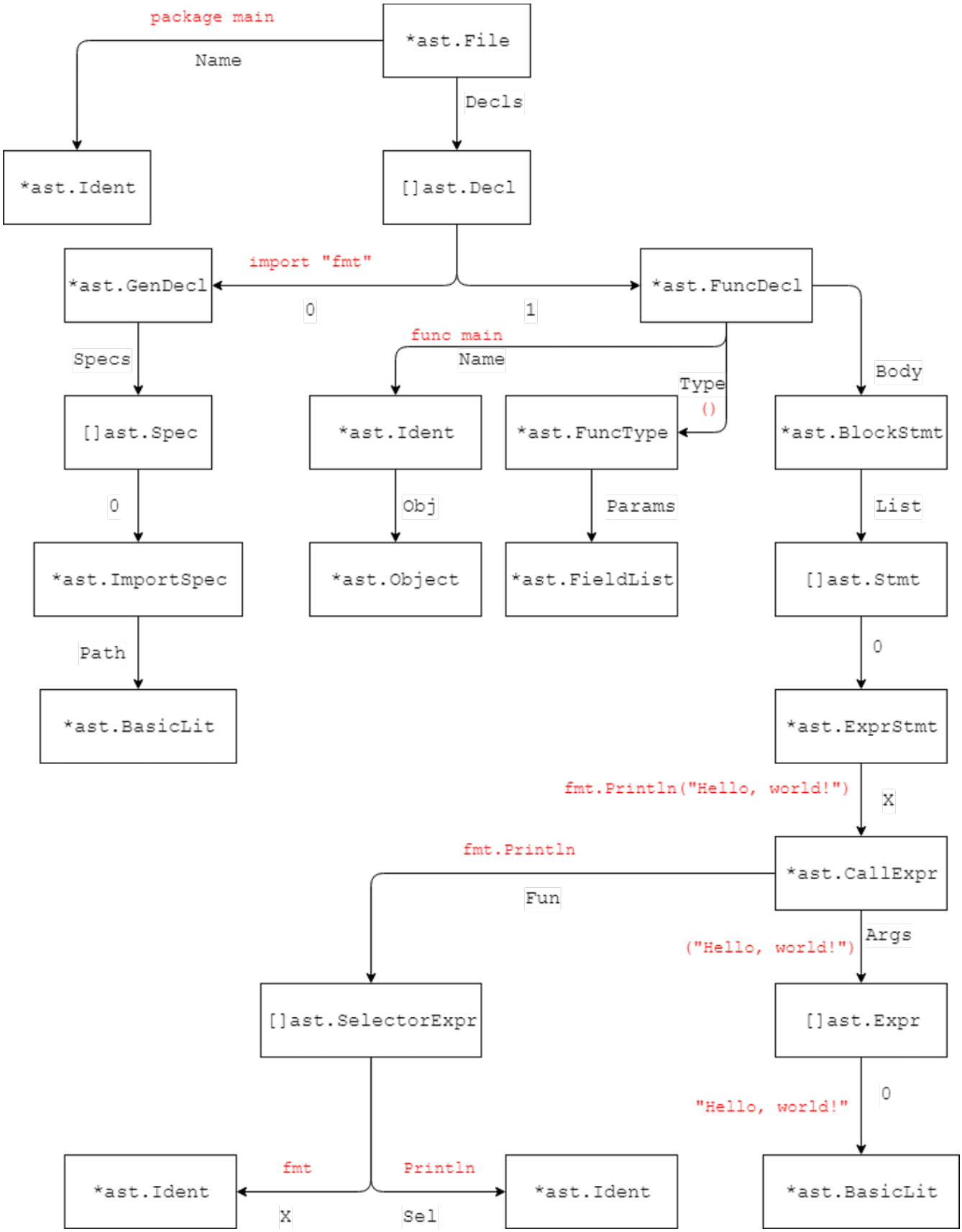
Go has again provided us with packages to parse the program and view the AST: **go/parser** and **go/ast**. We can use them like this to print the full AST:

Output:

In this output, you can see that there is quite some information about the program. In the **Decls** fields, there is a list of all declarations in the file, such as imports, constants, variables, and functions. In this case, we only

have two: our import of the **fmt** package and the main function.

To digest it further, we can look at this diagram, which is a representation of the above data, but only includes types and in red the code that corresponds to the nodes:



The main function is composed of three parts: the name, the declaration, and the body. The name is represented as an identifier with the value main. The declaration, specified by the Type field, would contain a list of parameters and return type if we had specified any. The body consists of a list of statements with all lines of our program, in this case only one.

Our single **fmt.Println** statement consists of quite a few parts in the AST. The statement is an **ExprStmt**, which represents an expression, which can, for example, be a function call, as it is here, or it can be a literal, a binary operation (for example addition and subtraction), a unary operation (for instance negating a number) and many more. Anything that can be used in a function call's arguments is an expression.

Our **ExprStmt** contains a **CallExpr**, which is our actual function call. This again includes several parts, most important of which are **Fun** and **Args**. Fun contains a reference to the function call, in this case, it is a **SelectorExpr**, because we select the **Println** identifier from the fmt package. However, in the AST it is not yet known to the compiler that **fmt** is a package, it could also be a variable in the AST.

Args contains a list of expressions which are the arguments to the function. In this case, we have passed a literal string to the function, so it is represented by a **BasicLit** with type **STRING**.

It is clear that we are able to deduce a lot from the AST. That means that we can also inspect the AST further and find for example all function calls in the file. To do so, we are going to use the **Inspect** function from the **ast** package. This function will recursively walk the tree and allow us to inspect the information from all nodes.

To extract all function calls, we are going to use the following code:

What we are doing here is looking for all nodes and whether they are of type ***ast.CallExpr**, which we just saw represented our function call. If they are, we are going to print the name of the function, which was present in the **Fun** member, using the printer package.

The output for this code will be:

fmt.Println

This is indeed the only function call in our simple program, so we indeed found all function calls.

After the AST has been constructed, all imports will be resolved using the GOPATH, or for Go 1.11 and up possibly modules. Then, types will be checked, and some preliminary optimizations are applied which make the execution of the program faster.

Code generation

After the imports have been resolved and the types have been checked, we are certain the program is valid Go code and we can start the process of converting the AST to (pseudo) machine code.

The first step in this process is to convert the AST to a lower-level representation of the program, specifically into a Static Single Assignment (SSA) form. This intermediate representation is not the final machine code, but it does represent the final machine code a lot more. SSA has a set of properties that make it easier to apply optimizations, most important of which that a variable is always defined before it is used and each variable is assigned exactly once.

After the initial version of the SSA has been generated, a number of optimization passes will be applied. These optimizations are applied to certain pieces of code that can be made simpler or faster for the processor to execute. For example, dead code, such as **if (false) { fmt.Println("test") }** can be eliminated because this will never execute. Another example of an optimization is that certain nil checks can be removed because the compiler can prove that these will never false.

Let's now look at the SSA and a few optimization passes of this simple program:

As you can see, this program has only one function and one import. It will print 2 when run. However, this sample will suffice for looking at the SSA.

Note: Only the SSA for the main function will be shown, as that is the interesting part.

To show the generated SSA, we will need to set the **GOSSAFUNC** environment variable to the function we would like to view the SSA of, in this case main. We will also need to pass the -S flag to the compiler, so it will print the code and create an HTML file. We will also compile the file for Linux 64-bit, to make sure the machine code will be equal to what you will be seeing here. So, to compile the file we will run:

$ GOSSAFUNC=main GOOS=linux GOARCH=amd64 go build -gcflags "-S" simple.go

It will print all SSA, but it will also generate a ssa.html file which is interactive so we will use that.

When you open ssa.html, a number of passes will be shown, most of which are collapsed. The start pass is the SSA that is generated from the AST; the lower pass converts the non-machine specific SSA to machine-specific SSA and genssa is the final generated machine code.

The start phase's code will look like this:

This simple program already generates quite a lot of SSA (35 lines in

total). However, a lot of it is boilerplate and quite a lot can be eliminated (the final SSA version has 28 lines and the final machine code version has 18 lines).

Each v is a new variable and can be clicked to view where it is used. The **b's** are blocks, so in this case, we have three blocks: **b1, b2**, and **b3. b1** will always be executed. **b2** and **b3** are conditional blocks, which can be seen by the **If v19 → b2 b3 (likely)** at the end of **b1**. We can click the **v19** in that line to view where **v19** is defined. We see it is defined as **IsSliceInBounds <bool> v14 v15**, and by viewing the Go compiler source code we can see that **IsSliceInBounds** checks that **0 <= arg0 <= arg1**. We can also click **v14** and **v15** to view how they are defined and we will see that **v14 = Const64 <int> [0]; Const64** is a constant 64-bit integer. **v15** is defined as the same but as **1**. So, we essentially have **0 <= 0 <= 1**, which is obviously **true**.

The compiler is also able to prove this and when we look at the **opt** phase ("machine-independent optimization"), we can see that it has rewritten **v19** as **ConstBool <bool> [true]**. This will be used in the **opt deadcode** phase, where **b3** is removed because **v19** from the conditional shown before is always true.

We are now going to take a look at another, simpler, optimization made by the Go compiler after the SSA has been converted into machine-specific SSA, so this will be machine code for the amd64 architecture. To do so, we are going to compare lower to lowered deadcode. This is the content of the lower phase:

In the HTML file, some lines are greyed out, which means they will be removed or changed in one of the next phases. For example, **v15** (**MOVQconst <int> [1]**) is greyed out. By further examining **v15** by clicking on it, we see it is used nowhere else, and **MOVQconst** is essentially the same instruction as we saw before, **Const64**, only machine-specific for **amd64**. So, we are setting **v15** to **1**. However, **v15** is used nowhere else, so it is useless (dead) code and can be eliminated.

The Go compiler applies a lot of these kinds of optimization. So, while the first generation of SSA from the AST might not be the fastest implementation, the compiler optimizes the SSA to a much faster version. Every phase in the HTML file is a phase where speed-ups can potentially happen.

If you are interested to learn more about SSA in the Go compiler, please check out the Go compiler's SSA source. Here, all operations, as well as optimizations, are defined.

Conclusion

Go is a very productive and performant language, supported by its compiler and its optimization. To learn more about the Go compiler, the source code has a great README.

If you would like to learn more about why Stream uses Go and in particular why we moved from Python to Go, please check out our blog post on switching to Go.