A Brief Overview of a Not-So-Simple GoLite Compiler

Deng, Shao Kun
260638592

Hai Tong Yang
260453583

Dong Kyu (Jason) Kim
260531174

April 16, 2016

1 Introduction

Nowadays, compilers are mainstream for all programming languages as they allow programming to be accessible to everyone, without the need of understanding the complex architecture that execute the machine code. In fact, very few people now possess the knowledge of write proper and structured machine code. From Fortran to Javascript, there exist a wide variety of languages that abstract low-level code to different level and it becomes increasingly important to understand how programming languages are constructed in order to improvement our programming skills. In this project, we design a subset of Go language by Google and build a complete compiler for it.

1.1 Motivation

The GoLite language was chosen because of its universality over OnceTime language. GoLite is a general-purpose language and is therefore turing complete. The target language of choice is C++. C++ is the industry standard for for its remarkable mix of high performance and abstraction. It is a mature language that has performant compilers, and will consequently improve the performance of our generated code. Other target languages were also considered. C++ was chosen over low-level code such as assembly because of our lack of knowledge in those languages and because of the high risk of buggy code due to their complexity. Higher level languages such as Python were rejected because of their lack of performance. C was a specially strong contender of target language. Howevere, C++ simply offered more flexibility with its Standard Template Library that offers high performance data structure and algorithms.

The toolkit of choice was SableCC3 because of it's simplicity and because of our familiarity with Java over C.

The rest of the report is organized as follows: in section 2, the structure of the compiler is depicted and explained; in section 3, the front end of the compiler, which includes the scanner, the parser and the weeder will be described; in section 4, the type system will be discussed along with the symbol table generation; the code generation scheme is described in section 5, examples of the compilation procedure along with their explanation will be provided in section 6 and the report concludes in section 7.

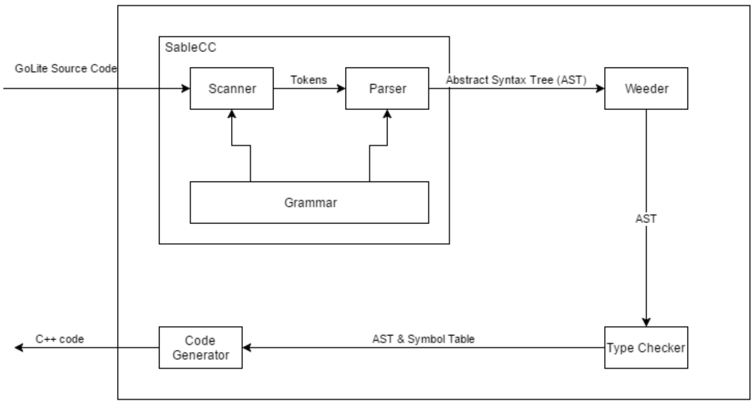

Figure 1: Overview of the system as a block diagram.

2 Compiler Structure

2.1 Overview

The overview of the entire system is depicted in Figure 1. The source code is fed into the scanner/parser, which is a program generated by SableCC from a given grammar, itself written in a language defined by SableCC. The scanner scans through the input source code as a string and generates a list of tokens which is then passed to the parser. The parser will then generate an abstract syntax Tree (AST) which will be used for subsequent operations. The AST is then fed into the weeder which will scan the AST for any semantic errors that are impossible or very tedious to verify with a context-free grammar. If the input program passes the weeder, the type checking is then performed on the AST by the type checker. Finally, if the input program is valid, the code generator will be called with the AST to generate the target code, which in this case is C++.

3 Front-End Structure

3.1 Scanner

The very first model of the system is the scanner. The scanner takes the Source code as a string and will generate a list of tokens, defined by a set of regular expressions. The tokens are first defined in a grammar file. SableCC then takes the grammar file and generates a scanner program in Java. Because the GoLite language allow the programmer to omit semi colons, we need to insert them manually in the scanner. Since the scanner is generate by SableCC, it needs to be overwritten by SableCC every time the grammar is modified, and it is therefore unwise to change the generated grammar directly. Instead, we extends the generated scanner class (called Lexer) and add supplement the semicolon insertion functionality without interfere with the generated scanner. This customized scanner is then used instead of the generate scanner. This process is depicted in the upper portion of Figure 2.

3.2 Parser

Based on a grammar definition file written using context free grammar, SableCC is able to generate a parser in Java. The parser will take the tokens produced by the scanner and generate an abstract syntax tree (AST) using the AST definition in the grammar definition file. This process if depicted in the lower portion of Figure 2.

3.3 Weeder

In our project, we made use of a weeder to not only verify constraints that cannot be expressed in context free grammar, but also for constraints that either are hard to enforce using the grammar or will make the grammar unnecessarily complex.

3.3.1 ***break* & *continue* keywords**

Break and *continue* are two very important keywords that are used in special locations to control the flow of the program. They should only appear in very specific code blocks, such as inside a *for* loop. Although it is possible to implement this restriction as part of the grammar, it results in very messy and hard to understand.

3.3.2 ***default* keyword**

The enforcement of at most one *default* keyword in a switch statement is done in the weeder.

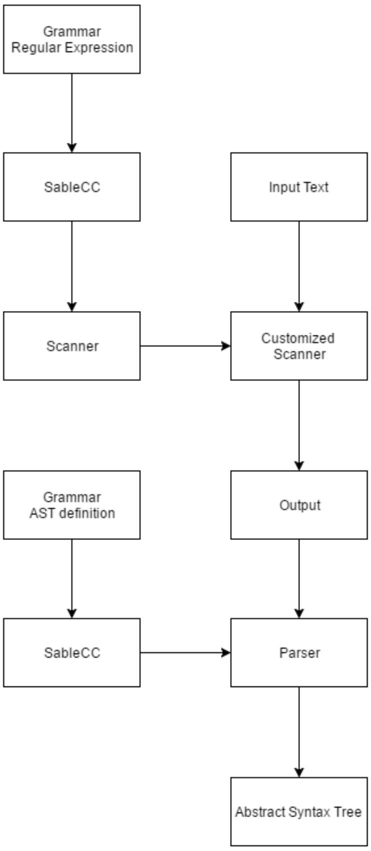

Figure 2: Structure of scanner and parser, both written in SableCC.

3.3.3 Assignment of multiple variables

When assigning multiple variables in one statement, we need to make sure that the number of L-values is the same as the number of R-values. This constraint, however, is context-sensitive and is therefore not included in our grammar definition file.

3.3.4 Rvalue assignment

In the grammar, an assignment is defined as two expressions separated by an equality sign. However, not all expressions are assignable and we need to make sure that the expression on the left-hand-side is a Lvalue.

3.3.5 Short declaration

For short declaration, we only allow the expression to be declared to be a list of identifiers.

3.3.6 Increment/decrement operator

Since the increment operation ++ and the decrement operator -- are treated separately in the grammar, we need confirm that their operands are Lvalue.

3.3.7 Basic type cast

We have *string* as a basic type, but by GoLite language definition, *string* type cannot be used for type casting. Therefore, the weeder needs to ensure that basic type casts are performed with *non-string* types. Same rule applies for aliases of *string* type.

3.3.8 *return* statement

In order to avoid runtime errors, a function that has a return type that is not *void* should return a value on all possible execution paths. This is context-sensitive and can only be verified in the weeder.

3.3.9 *for* loops

The post statement in a *for* loop definition cannot be a short declaration.

4 Type system

4.1 Types

4.1.1 Basic types

There are five basic types available in GoLite:

- *int* for integers. This is implemented as C++ *int* type.
- *float64* for 64-bit floating point numbers. This is implemented as C++ *double* type.
- *bool* for boolean type. This is implemented as C++ *bool* type.
- *rune* for characters. A *rune* type variable holds the ASCII number of the character. This is implemented as C++ *char* type.
- *string* for strings. This is implemented as *std::string* type from Standard Template Library.

4.1.2 Aggregate types

There are three aggregate types in GoLite:

- Arrays are fixed size ordered collections of variables. They are implemented as *std::array* types from Standard Template Library.
- Slices are variable length ordered collections of variables. They are implemented as pointers to *std::vector* types from Standard Template Library.
- Structures are collections of named variables. They are implemented as C++ *struct* types.

C++ natively support arrays implemented as pointers to preset heap locations. However, in GoLite, array assignments are deep copies and they are passed by value in function calls, which is not accomplished if we simply assign the pointers. By implementing arrays as *std::array* objects, the assignment rule will match that of GoLite since C++ assign objects by value. Conversely, slice assignments are shallow copies and only the reference is passed through function calls. Therefore, we have decided to use a pointer to C++ *std::vector* to represent the slices.
GoLite also allows the program to alias any type and use the alias as if it were the underlying type. This feature is implemented in C++ using the *typedef* statement.

4.2 Type rules[1]

4.2.1 Assignment rule

Two type are always assignment compatible if they are exactly the same type, that includes all child-types for aggregate types.

4.2.2 Casting rule

Types *int*, *float* and *rune* can freely cast among themselves and *bool* and *string* cannot be casted to or from other types.
GoLite does not support casting of non-basic types

4.3 Symbol table

To ease the implementation of the symbol table and type checking process. An abstract *Type* class has been created along with a set of possible types subclasses, including *AliasType*, *FunctionType* and *VoidType*. Those types abstracts the details of each class, which is especially handy for aggregate type where we need to keep track of child-types.
The *Type* class has the following methods:

- *is(Type t)* recursively checks if two types are exactly the same.
- *abstract assign(Type t)* checks if the t is assignable to the current type.
- *print()* returns the corresponding C++ code snippet for that type.

[1] See Appendix for complete listing of type rules

- *toString()* returns a abstract representation of the type to help visualize the type.

5 Code generation

5.1 Code generation pattern

Due to the similarities between GoLite and C++, the code generation scheme was fairly simple and straight-forward. All control structures present in GoLite can be used in C++, albeit with small modifications, which will be discussed in the next subsection. Variable and function manipulation schemes are also almost identical to their C++ counterpart.

5.2 Design decisions

The different in control structures between GoLite and C++ is that GoLite allows an initial statement along with every *if* statement, *for* statement and *switch* statement. That initial statement can declare variables that shadow existing variables in the same scope as the control structure but themselves can be shadowed by variable declared within the block of the control statement. This feature is not supported natively by C++. To implement it, we have envelopped every *if* block, *for* block and *switch* block by another scope that contains the initial statement and the control block itself. This way, the initial statement does not interfere with the outside scope (and can shadow variables from outside scope properly) and let variables declared inside the control block shadow any variable declared in the initial statement.

6 Example

6.1 Source code to abstract syntax tree

Consider the following code snippet written in GoLite:

```
package main
var x
func foo() int {
    return x
}
```

Given our grammar for the GoLite Language, it will generate the abstract syntax tree in Figure 3.

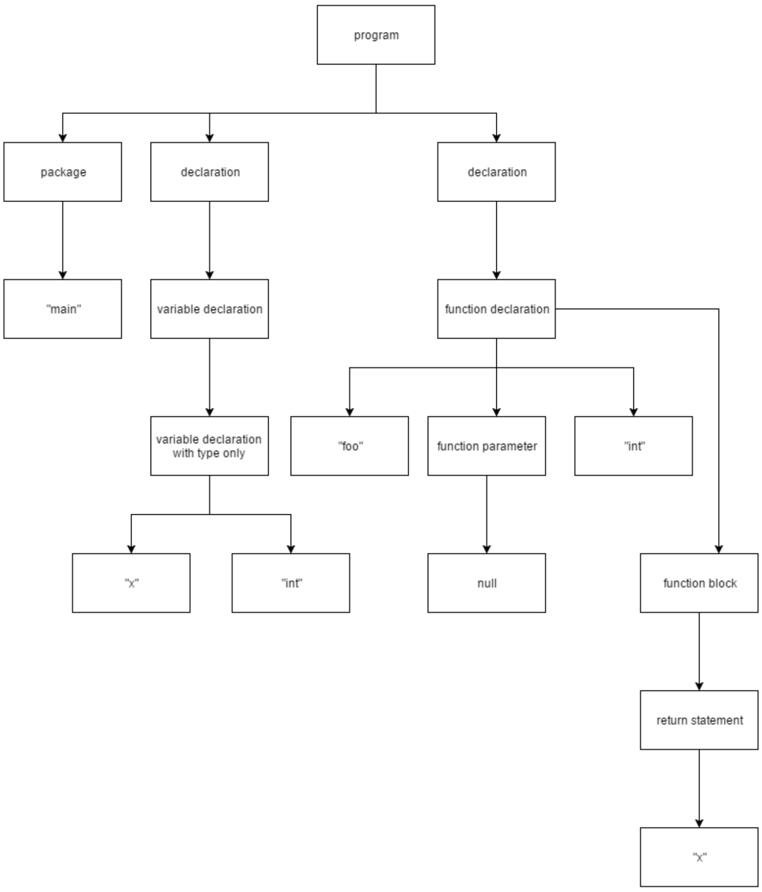

Figure 3: Abstract syntax tree for Example 6.1

6.2 Type checking

Consider the following code snippet written in GoLite:

```
var x int
var y float64
if (x == y) {
    //...
}
```

To type check the argument of the if statement properly. The type checker first has to verify the type of both sides of the binary operator ==. It finds that x can be casted to *float64* and the == operator is applicable on *float64* types. Given that information, the type checker then verifies the type of the expression returned by the operator ==, which is *bool*. Indeed, *if* statement type checks if the argument is of type *bool*.

6.3 Symbol table

Consider the following code snippet written in GoLite:

```
var x int
func foo() int {
    var x float64
    //A
}
func goo() int {
    //...
}
```

Assuming not forward declaration of global declarations, at point A, the symbol table would have the definition of *foo* as

```
 () int
```

in the outer scope and the definition of x as

```
float64
```

in the inner scope. Notice how the declaration of x in *foo* shadows the variable x in the outer scope.

7 Conclusion

In conclusion, building a GoLite compiler was not only a great opportunity to learn the important aspects of a compiler, but it also allowed us to familiarize with the process of designing a language and helped us understand why languages are built like they are. The skills we learned not only help us become better compiler programmer, but they also help us become better programmer in general by understanding the underlying constructs of programming languages.