GoLite Compiler - Final Report

COMP 520, Compiler Design
McGill University

Anton Gladyr

May 1, 2020

1 Introduction

The goal of the Compiler Design course was to master modern techniques and tools for building a fully functional compiler. To consolidate gained knowledge in practice, throughout the whole course, we were designing GoLite compiler - a complete compiler for a significant, non-trivial subset of Go language.

Go is a modern, statically typed programming language designed by Google employees. Go provides memory safety, garbage collection, structural typing, and concurrency, which make "easy-to-write" language for a programmer. From the list of Go features, GoLite supports only Go instructions and optional semi-colons. The set of features in GoLite language is small due to the time limitations of the course, but the implementation can be extended to support other features.

This report covers major design decisions, as well as the methods used at the implementation stage.

2 Compiler Functionality

GoLite compiler is a command line tool which supports 7 modes:

- *scan*: Takes *.go source file as input and transforms a string of characters (the code) into a stream of tokens. If the input successfully split into tokens, compiler exits with status code 0 and outputs "OK". Otherwise compiler prints an appropriate error message.

- *tokens*: Takes *.go source file as input and outputs generated tokens by scanner. Used for testing purposes.

- *parse*: Takes *.go source file as input and processes tokens produced by the scanner. Parser builds an abstract syntax tree (AST) using a context-free grammar.

- *pretty*: Takes *.go source file as input and outputs the source code by traversing the AST. Mainly used for testing purposes.

- *symbol*: Takes *.go source file as input and outputs the symbol table representing the declarations that are added in each scope as well as their type. If during scoping an error occurs, compiler generates an error message and exits with status code 1.

- *typecheck*: Takes *.go source file as input. If the input program code is type correct, compiler exits with status code 0 and outputs "OK". Otherwise compiler prints an appropriate error message.

- *codegen*: Takes *.go source file as input and translates the code into C language code. Saves the result in __golite_target_code.c file. If no errors occurred, compiler exits with status code 0 and outputs "OK".

3 Language and Tool Choices

GNU Bison and Flex are used together for implementing scanner and parser, as well as for building the AST. The structure of the AST and the remaining phases of the pipeline are implemented using C++11. C++ is a high-level language, which supports object-oriented paradigm and provides a wide variety of libraries and built-in features. This allows to apply OOP techniques and write clean code in general. The target language of the compiler is C language. It is challenging to simulate Go language constructions using C, however it provides high performance on the resulting executable programs.

4 Major Design Decisions

GoLite compiler is modular, consisting of multiple phases which form the compilation pipeline. Each individual phase takes input from the previous phase and outputs data for the next phase.

4.1 Scanner and Parser

The first phase of GoLite compiler is the scanning phase. Scanner transforms a string of characters (source program) into tokens. Each token is defined by a regular expression. Provided implementation supports only the ASCII charset, so the input (source code) must be in ASCII encoding. The Scanner defines regular expressions for recognizing the next language constructions:

- *keywords*: Scanner recognizes all the Go keywords. It stores all reserved keywords of Go language, so we prevent users from using them as identifiers. Types are not reserved words.
- *operators*: Recognizes all the Go operators.
- *comments*: Scanner recognize both block comment and line comments.
- *literals*: Scanner recognizes integer and floating-point numbers as well as interpreted and raw strings.
- *semicolons*: scanner inserts semicolon tokens in the token streams according to the rules of Go specification.

The scanner regular expressions are implemented on the basis of the rules defined in the Mini GoLite Syntax Specification by Vincent Foley.[1]

The second phase of the compiler is the parsing phase (syntactic analysis). It takes a string of tokens generated by the scanner as input, and builds a parse tree using a context-free grammar (CFG). In Go, a program is divided up in three parts: a package declaration (exactly 1), a list of import statements, and a list of top-level declarations. GoLite does not support packages, so we do not have to implement the import statements. However, we keep the package declaration; this allows GoLite programs to be used with a Go compiler. GoLite supports only five basic types: int, float64, bool, rune, string. The basic types are not reserved words.

Parser supports the next syntax structures:

- *variable declaration*
- *type declaration*
- *function declaration*
- *arrays*
- *empty statements*
- *block statements*
- *expression statements*
- *assignment statements*
- *declaration statements*
- *increment and decrement statements statements*
- *print and println statements*
- *return statements*
- *if statements*
- *switch statements*
- *for statements*
- *break and continue statements*
- *parenthesized expressions*
- *identifiers*
- *literals*
- *unary operations*
- *binary operations*
- *function calls*
- *array indexing*
- *built-in len and cap functions*
- *type casting*

The parser CFGs are formed on the basis of the rules defined in the Mini GoLite Syntax Specification by Vincent Foley.[1]

4.2 Abstract Syntax Tree

An abstract syntax tree (AST), is a tree form that represents a more abstract grammar. In an AST only important terminals are kept and intermediate non-terminals used for parsing are removed. Designing a solid AST nodes is important for later phases of the compiler as they will extensively use the AST. The set of AST nodes should represent all distinct programming language constructs; and be minimal (avoiding excess intermediate nodes). A concise AST will have 1 node type for each type of programming language construct. In the GoLite implementation, nodes of the AST are implemented as C++ classes:

- *Node class*: is an abstract class, which is the base structure for the all nodes in the tree. So, all nodes in the AST are inherited from this class.
- *Program*: is an abstract class, which inherits from the *Node* class. It is the parent node of the AST, and contains Go package name and all top-level declarations.
- *Declaration*: is an abstract class, which inherits from the *Node* class. It is the base structure for the all *declaration* nodes. So, all *declaration* nodes are inherited from this class.
- *Statement*: is an abstract class, which inherits from the *Node* class. It is the base structure for the all *statement* nodes. So, all *statement* nodes are inherited from this class.
- *Expression*: is an abstract class, which inherits from the *Node* class. It is the base structure for the all *expression* nodes. So, all *expression* nodes are inherited from this class.

Such an architecture allows us to upcast and downcast nodes of the AST to the parent and child nodes. Thus, this simplifies the process of a node identification as well as allows to merge identical blocks of code.

To manage the relatively large space of potential phase and node interactions, modern software engineering principles dictate that the code for a phase should be written in a single class, and not distributed among the various node types.[5] The **visitor pattern** enables phase- and node-specific code to be invoked cleanly, while aggregating the functionality of a phase in a single class. So, by using visitor design pattern, we can separate an algorithm from an object structure on which it operates. It allows us to add new operations to existing object structures without modifying the structures. Thus, provided implementation follows visitor pattern principles and keeps each traversal phase as a separate class.

4.3 Weeder

Sometimes, programming languages require more power, such as when the language is not context-free or an LALR grammar too big and complicated. To solve this issue we can change our CFG and accept a slightly larger language. In a separate phase (weeder) we can then weed out the bad parse trees. In GoLite there are 8 cases which should be weeded out:

- *break statement*: Allowed only within loops and switch statements.
- *continue statement*: Allowed only within loops.
- *default statement*: Switch statement should have only 1 optional default block.
- *expression statement*: Must be a function call.
- *assignment statement*: Number of expressions on the left-hand side must be equal to the number of expressions on the right-hand side.
- *post statement*: In a 3-part loop statement, the post statement cannot be a short declaration (not supported).
- *blank identifier*: Ignores assignments of blank identifiers (not supported).
- *short declaration list*: not supported.

4.4 Symbol Table

The next 2 phases of the compiler pipeline are building of symbol table and type checking. Since we do not need to support mutually recursive declarations in GoLite, both phases happen at the same time in SymbolTableBuilder class. The scopes and type system for GoLite are nearly the same as for Go, except the restricted language subset. The typing and semantics rules for all GoLite constructs are defined in *GoLite Type Checking Specification* by Vincent Foley.[2] Therefore, all the rules in the compiler are implemented in accordance with the specification.

Symbol table is used to describe and analyze declarations and uses of identifiers. The structure is implemented as a cactus stack of hash tables. Implemented symbol table provides 3 primary functions:

- Mapping identifiers to declarations/other information.

- Linking uses with their respective symbols.
- Scoping and unscoping.

In GoLite, the **scoping rules** implemented in the way that statements and expressions may reference symbols of either current or parent scope. However, we cannot reference symbols defined in sibling/child scopes. The symbol table contains both the table of symbols, as well as a pointer to the parent scope.

The symbol structure contains the next fields:

- *name*: the name of the identifier (type, variable, function, etc.).
- *category*: specifies the kind of the identifeir. There are 4 categories: type, constant, variable, function.
- *type*: type name of the identifier.
- *baseType*: resolved base type of the identifier.
- *symbType*: a reference to the symbol of the type.
- *tmpCounterNum*: an integer counter. Used for creating temporary variables during the code generation phase.
- *next*: a pointer to the next symbol in the list (current scope).
- *node*: a link to the AST declaration (weaving)

4.4.1 Declarations

Declarations define new identifiers in the symbol table. GoLite makes a simplification that identifiers must be declared before they are used, so we avoid recursion calls. If an identifier is already declared in the current scope, the compiler generates an error. There is only exception for the special function *init* which may be declared multiple times, since it does not introduce a binding. If an identifier is already declared in the parent scope, the new mapping will shadow the previous mapping.

At the top-level scope identifiers *init* and *main* must be used only for function declarations with no parameters and return type. *main* function must be declared only once.

4.4.2 Types

In provided implementation, a symbol table stores pre-declared (base) types at the top-level scope. GoLite supports 5 base types: *int, float64, bool, rune, string*. Top-level scope also stores boolean identifiers. If a given type of a declaration is not declared in the symbol table, an error is raised.

During type checking/type resolving, each expression saves type name string as a TypeDescriptor object, so it simplifies the type checking process of declarations and statements. TypeDescriptor objects are used during code generator phase as well.

4.5 Code Generator

GoLite is a translator. This approach (separate "frontend") makes it possible to simplify the porting of the language to different platforms (JVM, CLR, Web, etc.). This implementation uses C as an intermediate language. The compilation is divided into 2 stages. The first stage converts the AST into intermediate code (code-generation) and the other is to compile intermediate code into a set of target system instructions. At the code generation level, GoLite determines which instructions should be used (instruction selection) and in what order to place these instructions (instruction planning). The placement in the registers occurs in the second stage (compilation of generated instructions).

This section shows the main problems that arise during translating Go source code into C language instructions and possible solutions.

4.5.1 Execution

A go program may consist one main function (entry point) and zero-or-more init functions. The init functions must be invoked before the main function in lexical order. To implement this behavior in C language, all init function names must be changed by adding a numerical specifier to each function name, so C program can recognize each init function. C invokes all init functions in the order in which they are declared. Go-main function must be invoked after all init functions.

During program execution, Go is pass-by-value and return-by-value. So when we pass or return an array in C, we should keep in mind that in C arrays are passed as a pointer to the first element. That allows the called function to modify the contents of the array. To implement the simulation of passing by value for arrays, we can make a copy of the array (memcpy) before passing it to a function.

During execution, Go preserves left-to-right evaluation order. It can be trivially reproduced in C.

4.5.2 Declarations

There are 3 kinds of declarations in Go language:

- *Function declarations*
- *Variable declarations*
- *Type declarations*

Declarations can cause naming conflicts with C keywords. Renaming all identifiers with a unique prefix/-suffix trivially solves the problem.

Function declarations are nearly the same across all programming languages, so in C it has the almost the same structure. However, as discussed in *4.5.1 Execution* section, there are some challenges with passing and returning arrays.

In Go language there are 4 special cases of defining variables:

- *Implicit initialization* - a variable implicitly initialized to 0
- *Multiple declarations* - assign multiple values simultaneously
- *Shadowing of true and false constants*

C has different scoping rules than Go. To simulate the behavior of Go language, we should use temporary variables with unique specifiers/ids.

Since type declarations in Go does not make any difference in C in terms of base types, we do not have to translate *type* instructions.

4.6 Assignments

An assignment statement copies the value of the expression to the variable. The tricky part of implementing Go assignments in C, is that Go supports multiple assignments:

```
a, b = b, a
```

where the instruction swaps the values of a and b. To solve this issue in C, we can use temporary variables to store old values of all right-hand side expressions.

```
int tmp0 = b;
int tmp1 = a;
a = tmp0;
b = tmp1;
```

4.7 Statements

In GoLite mini if statements consists of:

- *Condition expression*
- *True branch*
- *Zero-or-more else-if branches*
- *Optional else branch*

Because of the simplified version of if statements in GoLite mini, *if* condition can be easily translated to C. However, there are issues in the full version, namely with optional init statements.

There are 3 kinds of loop statements

- *Infinite loop*
- *While loop*
- *3-part loop*

The first two kinds of loop statement can be easily translated into C language by using *while()* and *while(true)* instructions. However, it is very hard to implement 3-part loop because of its multiple init and post statements. To simulate this behavior in C, we can use *while()* loop with goto(jmp).

Switch statements in Go simulates if/else-if behavior. In GoLite mini switch statements contains:

- *Optional switch expression*
- *Zero-or more cases*
- *List of one-or-more non-constant expressions*
- *Body*
- *Optional break*
- *Optional default case*

To implement switches in C, we should use try/-catch and loops. As it proposed in *GoLite Tutorial*, the implementation in C will look similar to the next *while* loop: [4]

```
while (true) {
    int tmp__0 = foo();
    if (tmp__0 == a || tmp__0 == bar())
    if (b > c) {
        break;
        }
    } else {
        // Default branch
    }
    break;
}
```

5 Conclusion

This report discussed the main principles for implementing a fully-functional compiler. Namely, the report shows the flow of building GoLite compiler. It justifies language and tool choices for building the compiler as well as explains each module in the compilation pipeline.

References

[1] Foley, V.: *Mini GoLite Syntax Specification* [https://www.cs.mcgill.ca/ cs520/2020/project_mini/ Milestone1_Specifications_Mini.pdf]. Montreal, Canada: McGill University, School of Computer Science.

[2] Foley, V.: *GoLite Type Checking Specification* [https://www.cs.mcgill.ca/ cs520/2020/project_mini/ Milestone2_Specifications_Mini.pdf]. Montreal, Canada: McGill University, School of Computer Science.

[3] Foley-Bourgon, V.: *GoLite Project* [https://www.cs.mcgill.ca/ cs520/2020/slides/10-golite.pdf]. Montreal, Canada: McGill University, School of Computer Science.

[4] Krolik, A.: *GoLite Tutorial* [https://www.cs.mcgill.ca/ cs520/2020/slides/10-golite-tutorial.pdf]. Montreal, Canada: McGill University, School of Computer Science.

[5] Charles N. Fischer, Richard Joseph LeBlanc, Ronald Kaplan Cytron (2010). Crafting a Compiler. Pearson Education, Inc.: Addison-Wesley.