GoLite Compiler Final Report

Group 12: Zoe Guan, Samuel Laferriere, Emil Rose

April 15, 2015

Contents

1 INTRODUCTION

In this project, we made a compiler which translates the GoLite subset of Go into C code. Documentation for this subset of Go can be found on the course website, `http://www.cs.mcgill.ca/~cs520/2015/`. We implemented our compiler using the Java language and its compiler compiler tool SableCC3 (`http://sablecc3.sablecc.org/`), developped by Etienne Gagnon while a Master's student at McGill University.

We did very well on the first three milestones, passing close to 100% of the tests in each case. We had a harsher time doing milestone 4 as the code generation in C proved to be much harder and different from GoLite than we initially thought it would be. We will explain in greater details in the section for Milestone 4 the constructs that gave us the most trouble.

1.1 Tools

SableCC3 was a difficult tool to use at first, but proved to be very efficient in the long run. In particular, it was convenient having all of the node files automatically generated with corresponding getter and setter functions. An advantage of using a tool that produces code written in Java is that limitations of SableCC3 could be solved using object-oriented features like inheritance, e.g. for our implementation of the GoLiteLexer discussed below. SableCC3 provides useful analysis classes, such as DepthFirstAdapter for AST traversal, which can be extended to apply customized actions to nodes; our weeder, pretty printer, type checker, and code generator were all implemented by extending DepthFirstAdapter. Other tools we used included Makefile and Bash scripts to allow rapid compilation and testing of the changes we made, and Git and Github to synchronize our project.

2 MILESTONE 1: Scanning and Parsing

2.1 Scanner

The scanner part of the project was mostly straightforward, except for GoLite's semicolon insertion rules. In GoLite, a semicolon is automatically inserted at the end of a line if the final token in the line is an identifier, a literal, a `break/continue/return` keyword, `++/--`, or a closing bracket. Since SableCC does not allow you to specify instructions for handling in the Tokens section, we implemented semicolon insertion by creating a GoLiteLexer class that extends the Lexer class generated by SableCC. GoLiteLexer overrides the filter() method to perform token replacement. When a newline/EOF token is preceded by one of the tokens mentioned above, GoLiteLexer replaces the newline/EOF token with a semicolon token. Our main function uses this extended scanner, rather than the generated scanner, to read through the file.

2.2 Grammar and AST

For the parser, we first made a rough grammar based on the GoLite specification (`http://www.cs.mcgill.ca/~cs520/2015/assignments/syntax.pdf`), and debugged it. We then went more carefully through the overall Go specification and made sure every valid phrase in GoLite could be expressed by our grammar. Our SableCC3 grammar closely follows the official EBNF specification for Go at `https://golang.org/ref/spec` (corresponding to the GoLite subset). We aimed to express as many rules as possible in the grammar to decrease the number of things to weed for, so we added some helper definitions to make it easier to enforce certain rules. For example, we enforced the rule that the left hand side of an assignment must be an lvalue by defining lvalues in our grammar. We also made a distinction between blank and non-blank identifiers in the grammar to cover most of the rules for the placement of blank identifiers, such as the fact that they can appear in declarations and on the left side of assignments (ex: `x,_:=1,2`) but cannot be used as values.

Designing the grammar proved trickier than we had thought. At times, we ran into rules that were impossible or inconvenient to express using an unambiguous grammar. We decided to relegate some of these problems to the weeding phase to make our grammar simpler while still guaranteeing correctness of the parse. We changed some definitions to be more general and then used weeding to check that the desired definition was satisfied. For example, the left hand side of a short variable declaration `idlist := explist` can only contain identifiers, but defining it that way in the grammar gave a reduce/reduce conflict because any `id` is an lvalue. Thus, instead of requiring the left hand side to be a list of identifiers, the parser allows the left hand side to be a list of lvalues, and the weeder later verifies that the lvalues are identifiers. Our lvalue definition

```
lvalue {-> exp} = {id} id
{-> New exp.id(id)}
| {selector} selector
{-> selector.exp}
| {indexing} indexing
{-> indexing.exp}
| {func_call} func_call
{-> func_call.exp}
;
```

is also more general than the desired definition because function calls are not lvalues. We chose this definition because it allows for simple definitions of selector expressions `x.f` and indexing expressions `a[i]` (`x` and `a` are usually lvalues but can also be function calls)

```
selector {-> exp} = lvalue dot id
{-> New exp.selector (lvalue.exp, id)}
;
indexing {-> exp} = lvalue l_bracket exp r_bracket
{-> New exp.indexing (lvalue.exp, id)}
;
```

and it is easy to weed out function calls mistakenly parsed as lvalues.

One parsing ambiguity dealt with in the type checking phase was the fact that type casts involving alias types and function calls with a single parameter have the same syntax and cannot be distinguished from each other during parsing. Both are parsed as function calls.

For the Abstract Syntax Tree, we used the SableCC3 feature that allows you to define the AST in a section called Abstract Syntax Tree and specify the CST to AST rules in the grammar. Most of the AST nodes fall into three categories: statements, expressions, types. There are also a few other nodes for things like struct fields and parameter lists in function declarations. For the AST productions, the part that proved most troublesome was the transformation from production rules that used the * EBNF notation instead of having specified list-unwrapping helper rules. When building the AST, it would have been useful to unwrap some of these lists (for example if the * notation is used on a list to make a list of list, and we want to store each individual list in a node by itself), but SableCC does not seem to offer this possibility. Being reluctant to modify our grammar once we finally got it working, we overcame this problem by instead introducing helper rules in the AST itself, such as the else if rule:

```
stmt = ...
| {if} if stmt?  exp [true]:stmt* else_if* [false]:stmt*;
| ...
;
elseif = stmt ?  exp [body]:stmt
;
```

We had to go back and change certain grammar and AST rules at least a few times during the other Milestones, so as to facilitate the task of other modules. For example, we originally did not allow standalone blocks of statements like

```
{
   statement 1;
   ...
   statement n;
}
```

but added them to simplify the implementation of the scoping rules during the type checking phase. Since functions, if statements and for statements all have bodies of this form, and each block of statements constitutes a scope, we realized it would be convenient to have an AST node for blocks.

One more issue we encountered was preserving line number information in the AST transformations. In SableCC3, only tokens carry line numbers, so we can only get the line number of an AST node if it contains a token. Since we wanted to have line numbers for error messages in all phases, we added some extra tokens to the AST, such as the "print" keyword in print statements, to ensure that each AST node would contain a token. We added a LineNumber class with a method that gives a line number for an AST node by finding a token among its children.

2.3 Pretty Printer

Our pretty printer extends DepthFirstAdapter. One of the tests for the parser was to check the invariant pretty(parse(pretty(parse(P)))) ≡ pretty(parse(P)) for syntactically correct programs P.

2.4 Weeder

For several cases in the parsing and type checking phases, it was more convenient to identify errors using a separate Weeder class that extends DepthFirstAdapter. The weeding phase happens in between the parsing and type checking phases.

Cases dealt with through Weeding

- Requiring the number of expressions on the RHS of an assignment statement match the number on the LHS
- Requiring "break" only occur in a loop/switch statement
- Requiring "continue" only occur in a loop
- Requiring a switch statement has just one "default"
- Requiring the list of values on the LHS of a short declaration statement are identifiers, and that the number of expressions on each side matches
- Requiring short declaration statements don't try to assign "_"
- Requiring the number of expressions on each side of a variable declaration statement matches
- Requiring for-loops `for init_stmt; exp ; post_stmt { body }` cannot have short declaration statements for their post statement
- Requiring one can't type cast to string through an alias
- Requiring that one can't type cast to string directly
- Requiring that a function that returns a value does so on every execution path
- Collecting a list of aliases for strings

3 MILESTONE 2: Type Checking

Symbol table construction and type checking happen in the same pass. They are implemented using the class SymbolTable, which again extends DepthFirstAdapter. To keep track of type information for GoLite types and functions, we made a class Type and several classes which extend Type:

IntType, Float64Type, BooleanType, RuneType, StringType, ArrayType, SliceType, StructType, AliasType, and FunctionType. These classes provide information about an expression's or a function's type. For example, an ArrayType object contains an array's element type and size and a FunctionType object contains the function name, a Type list corresponding to the function's parameters and a Type corresponding to the return type.

3.1 Symbol Table

Our symbol table implementation uses two parallel stacks of scopes, one for variable declarations and the other for type declarations. A scope is a hashmap from Strings to Types. We had started off using a single stack of scopes which held both variable declarations and type declarations, but this proved hard to work with when trying to differentiate between a type alias and a variable of that type alias. For example,

```
var num int;
var x num;
```

is not allowed because `num` is not a type. Although using separate symbol tables for type and variable declarations is perhaps not the prettiest of solutions, it made our life much easier. The main difficulty in implementing the symbol table using SableCC was the fact that the CaseANode-Name methods of the DepthFirstAdapter all return void. When trying to declare a variable of a struct type for example, we needed access to the types of the struct fields, and so a non void return would have been helpful. To circumvent this problem, we used a global variable which held the most recent visited node's Type.

Scoping Rules

- All top-level declarations belong to the outermost scope
- In a function declaration, the list of parameters (possibly empty) opens a new scope that closes at the end of the function body
- The start of a non-function-body block opens a new scope that closes at the end of the block
  - In our implementation, each of the following corresponds to a block node in the AST:
    - Function body (but a function body does not open a new scope because it is in the same scope as the list of parameters)
    - For loop body
    - If statement body
    - Case or default body in a switch statement (a case or default body is an implicit block without braces)
    - Any other block of statements
- The `init` statement in `for init; exp; post block` opens a new scope that closes at the end of the block

- The `init` statement in `if init; exp block [else (ifStmt | block)]` opens a new scope that closes at the end of last block and the `init` in `else if init; exp block` opens a new scope that closes at the end of the last block of the parent if statement of the else if
- The `init` statement in `switch init; exp { switchBody }` opens a new scope that closes at the end of the switch statement body
- A struct type declaration body opens a new scope that closes at the end of the body

A tricky case was determining when to pop which scope in an if-elseif chain containing multiple `init` statements. For example, consider the 5 scopes of this (valid) code snippet:

```
if x:=1; x!=1 {
    x:='x'
    println(x)
} else if x:="x"; x!="x" {
    x:=1.5
    println(x)
} else {
    println(x)
}
```

```
if x:=1; x!=1 {
    x:='x'
    println(x)
} else if x:="x"; x!="x" {
    x:=1.5
    println(x)
} else {
    println(x)
}
```

```
if x:=1; x!=1 {
    x:='x'
    println(x)
} else if x:="x"; x!="x" {
    x:=1.5
    println(x)
} else {
    println(x)
}
```

```
if x:=1; x!=1 {
   x:='x'
   println(x)
} else if x:="x"; x!="x" {
   x:=1.5
   println(x)
} else {
   println(x)
}

if x:=1; x!=1 {
   x:='x'
   println(x)
} else if x:="x"; x!="x" {
   x:=1.5
   println(x)
} else {
   println(x)
}
```

We kept count of the number of `init`'s in the chain and popped all of the corresponding scopes at the end of the chain. We also kept track of the block scopes throughout using the block AST node.

3.2 Type Checker

For the type checker, we took advantage of the outANodeName methods in DepthFirstAdapter. These methods are always called after the node's children are explored, so we just overrode these methods, guaranteeing that the node would have the type information required about its children. Type information was stored in a hashmap from expression nodes to Types, allowing efficient access, even for large programs with many nodes. To keep the code at a reasonable length, we heavily used helper methods for the type checker because of how repetitive many of the cases are. For example, there is a method for getting the underlying type of an alias type and one for checking if two expressions are comparable.

The typing rules are described in `http://www.cs.mcgill.ca/~cs520/2015/assignments/typechecker.pdf`. We will discuss our type checking strategy for short declaration statements, function declarations and return statements. We will also mention how the type checker deals with the fact that alias type casts are parsed as function calls.

A short declaration statement `idlist = explist` type checks if all expressions in `explist` are well-typed, there is at least one variable in `idlist` that is not declared in the current scope, and variables already declared in the current scope are assigned expressions of the same type. The type checker first type checks all expressions in `explist`. Then it iterates through `idlist` and `explist`

in parallel, updating a String list of left hand side variables encountered so far and counting the number of new variables.
For each `id-exp` pair:

If `id` is already in the String list, throw an exception because there cannot be two variables on the left with the same name.

If `id` is not yet declared in the current scope, get the Type `T` of `exp` from the type map, add the mapping `id` $\rightarrow$ `T` to the variable symbol table and increment the new variable count.

Otherwise, look up the type of `id` and the type of `exp` and check if they match.

After the final iteration, check the new variable count and throw an exception if it is zero.

For function declaration statements, we check the function type (parameter types and return type) before checking the statements in the function body because we need the function return type for type checking return statements. We check that the function parameter names are all distinct, add the function to the symbol table and update a global variable variable storing the return type of the current function. Then we type check the function body. If we encounter a return statement we check that the return expression matches the type in the global variable with the function return type. The check for return statements along every execution path in a non-void function is done in the Weeder.

Type casts with aliases and function calls with a single parameter both have the form `f(x)`, but in the type checking phase we have enough information to distinguish between them. When type checking function calls, we look up the most recent declaration of `f` and check whether it is a FunctionType. If so, we type check the expression assuming it is a function call (check that `f` takes one argument of the same type as `x`). Otherwise, we type check the expression assuming it is a type cast (check that `f` is a cartable type and that `x` can be cast to `f`).

4 MILESTONES 3 & 4: Code Generation

We generate code using a CodeGen class that extends DepthFirstAdapter. Our target language is C. We chose C so that we could avoid writing low-level code and take advantage of the optimizations provided by GCC. Also, GoLite's scoping rules are similar to C's. However, we had a challenging time emulating certain GoLite features in C and unfortunately, our compiler does not generate correct C code for all GoLite constructs.

4.1 Information Collected in Preparation for Code Generation

The code generator uses information collected during the type checking phase. Since the code generation for some constructs requires type information, the code generator obtains a copy of the type map created by the type checker. To deal with naming collisions, the type checker creates a list of the names used in a program, which is accessed by the code generator. The type checker also provides the code generator with slice sizes by keeping track of the number of `append`'s applied to each slice. Lastly, the type checker provides the code generator with information about short variable declarations.

In GoLite, the left hand side of a short variable declaration `idlist := explist` must contain at least one variable that was not previously declared in the current scope, but can also contain already declared variables. When translating the short variable declaration to C, we produce a separate statement for each variable in `idlist`: a declaration if the variable is new and an assignment if the variable is old. Since the code generator needs to know which variables in `idlist` are new and which are old, we collect this information with a hashmap in the type checking phase and give the code generator a copy of the hashmap. Each key of the hashmap is a short declaration statement node and the corresponding value is a list of the newly declared variables in the statement.

4.2 Strategy

For expressions and statements with basic types, the translation from GoLite to C was straightforward. However, for many other features, especially those involving arrays and slices, the translation was challenging due to differences between GoLite and C. We will discuss our implementation for some of the less straightforward cases and summarize what we were not able to implement.

Automatic Initialization: In GoLite, when a variable is declared but not initialized, its value is set to the zero value of its type. Thus, a GoLite variable declaration without initialization gets converted to a C variable declaration that explicitly initializes the variable to the zero value for its type:

```
var a [10]int
⟶
int a[10] = {0};
```

Top-level Declarations: One issue in translating global variable declarations to C was that C does not allow top-level declarations where the right hand side is a non-constant. For example,

```
var x = 0
var y = x
```

is allowed at the top-level in GoLite, but

```
int x = 0;
int y = x;
```

is not allowed at the top-level in C. To deal with this, when generating C code for a top-level GoLite declaration with initialization, we do the initialization in the main function instead.

Short Declaration Statements: For short declaration statements, like explained above, we generate either a C variable declaration or a C assignment depending on whether the variable is newly declared or not.

```
x := 1
x, y := 2, 3
⟶
int x = 1;
x = 2;
```

```
int y = 3;
```

Slices: To generate code for GoLite slices, which are dynamic arrays, we defined a struct containing a void pointer and ints curSize (to keep track of where to add an element when `append` is called) and maxSize (for bounds checking):

```
typedef struct {
   int curSize;
   int maxSize;
   void *data; } slice;
```

maxSize is computed during type checking by counting the number of times `append` is called on the slice. We initialize slices by getting the element type `T` from SliceType, setting curSize to 0, setting maxSize to the number computed during type checking, and setting the void pointer to point to an array of `T` of length maxSize. For appending to slices,

```
a = append(a,1)
⟶
a.curSize++; ((int*)(a.data))[a.curSize] = 1;
```

Bounds Checking: GoLite does automatic bounds checking for arrays and slices but C does not, so we added a bounds checking C function to every generated C program. The function signature is

```
int bounds_check(int index, int len);
```

and it causes the program to exit with a failure if `index<0` or `index>=len`. A GoLite indexing expression such as `a[i]` becomes `a[bounds_check(i,len)]` in C, where `len` is obtained from `a`'s Type if `a` is an array or the maxSize field of `a` if `a` is a slice.

String Concatenation: In GoLite, strings can be concatenated using the + operator, so in the generated C code, we added a helper function `concat` that performs string concatenation:

```
char* concat(char *s1, char *s2);
```

Optional `init` Statements: GoLite allows an optional short statement at the start of an if or switch statement (ex: `if init; exp block`) but C does not, so during code generation, we pull out the short statement and place it before the if or switch statement. Since the short statement starts a new scope in GoLite, we emulate this in C by adding additional braces:

```
if i:=0; i<j {
   print(j)
}
⟶
{
   int i = 0;
   if (i<j) {
      printf("%d", j);
   }
}
```

An if-elseif chain with several `init`'s becomes a chain of if statements with pulled out `init`'s. We define a new boolean before the chain and set it to `true` if and only if it enters an if-body. In each if condition, we check if the boolean is true;

```
if x==y {
} else if x:=5; x==y {
} else if x:=10; x==y {
}
⟶
if (x==y) {
bool newBool = false;
{
   int x = 5;
   {
      if (!newBool && x==y) {
         newBool = true;
      }
      {
         int x = 10;
         if (!newBool && x==y) {
            newBool = true;
         }
      }
   }
}
}
```

Array Comparisons: In GoLite, two arrays can be compared using the == operator if they have the same element type and size. However, in C, arrays cannot be compared using such a concise expression and we need to loop over all of the elements. We attempted to write a recursive C function to compare arrays:
`int eq_array(void* array1, void* array2, char* type, int* dimensions, int index);`
Here, type specifies the element type, dimensions is an array containing the dimensions of the array1 (obtained from ArrayType), and index refers to the dimension we are currently comparing. However, we had problems with the recursive call, probably because of the void pointers, and the function only works for 1D arrays.

For Milestone 3, we implemented code generation for a subset of GoLite with basic types, arrays (but not array comparisons or assigning an array to another), expressions involving basic types, declarations, and most statements involving basic types. However, during Milestone 4, we ran into a lot of problems while trying to add slices, non-basic structs, non-basic aliases, and the remaining array features. Since we came across several GoLite expressions and statements for which we could not find a comparably clean or concise C counterpart, we ended up adding quite a bit of additional C code in order to try and reproduce the behaviour of the GoLite code. However, many things do not work as expected due to issues with pointers.

GoLite's treatment of arrays is considerably different from C's in many respects.

4.3 Unimplemented Features and Known Issues

C programs generated by the compiler may fail to compile or behave in unexpected ways when the corresponding GoLite code contains slices (particularly functions that return slices; there also seems to be an issue with allocating sufficient memory during slice initialization), structs with non-basic fields, array comparisons, or statements assigning to arrays to arrays. Other constructs that were not implemented are functions that return anonymous structs and switch statements with non-basic expressions. We did not resolve naming conflicts between GoLite identifiers and C keywords/built-in functions (we have a list of names used in a program and a method for generating new names, but we did not have time to implement renaming). There is a bug with alias initialization that we overlooked when we submitted Milestone 4. As a result, if a GoLite program declares a variable of an alias type without initializing it, the generated C code will not compile.

5 TESTING

We wrote many correct and incorrect GoLite programs with which we tested our compiler at each milestone. We also used the pretty printer, symbol table dump and pretty type printer to help debug. After milestones 1 and 2, Vincent provided a collection of programs written by other teams and by him with which we also tested our compiler. Whenever we came across a failed test case, we went back and fixed the relevant part in the compiler. Our testing was not as thorough for the final milestone due to time constraints, and many of the tests we did run were not successful because of problems with slices. There was also a bug with alias initialization that we did not catch until after we had submitted our compiler. However, we did verify that the compiler generates correct C code for basic constructs and for our benchmarks. The C code runs considerably faster than the Go code when there are few indexing expressions. However, the C code runs slower when there are a lot of indexing expressions, probably because of the bounds checking function.

6 CONCLUSION

In conclusion, the scanning, parsing and type checking parts of our compiler are very functional and perform very closely to what we had intended for it to perform. Unfortunately, the code generation part of our compiler greatly lacks behind compared to the other modules. Nonetheless, thanks to the great modularity of modern compilers, from which we based our model, it would be easy to bring the code generation up to par with the other parts.

6.1 Lessons Learned from the Project

One benefit of the project, and the course overall, is that it makes you a better programmer. Even in a stripped-down language like GoLite, features that we take for granted (e.g. array bounds checking) are very difficult to make work with a compiler, especially when compiling to C. Having worked on this project, we can now look at language features, especially abstractions, and have some intuitions about what is going on underneath the hood in implementing that feature. We also have an understanding, through code generation and the peephole project, of the types of code a compiler can generate well, and the types it can't. This allows us to make educated decisions about how to write code that is efficient, especially for complex code relying on many high-level abstractions.

Another benefit of this project is that it really makes you think about what features and syntax in programming languages you value, and which ones you don't need. Go is a very interesting example of this because the creators chose to be very liberal with certain features (especially syntax, e.g optional semicolons and trailing commas), but cut out a number of popular features (e.g. generics, objects). This will allow us to better design domain-specific languages; this is important because, as we saw in the project, small details and later decisions are easy to change, but changing an early choice can cause problems through the later modules.

This project also made us appreciate how important the choice of tool is. At first, getting SableCC working and becoming comfortable with its syntax was frustrating. But being able to take advantage of all of the structure that SableCC provides paid off many times over as the project continued. When problems cropped up, e.g. not being able to generate a parser that had the semicolon behavior we wanted, Java allowed us to elegantly deal with the issue.