On the design and implementation of a GoLite compiler

Wei Gao Omar Gonzalez Deepanjan Roy

We present the design and implementation a compiler that compiles GoLite to Java bytecode. GoLite is a proper subset of the Go programming language defined by Vincent Foley-Bourgon for a compilers course at McGill University. Its syntax and typing rules can be found at `http://www.cs.mcgill.ca/~cs520/2015/`, and its operational semantics is assumed to be identical to Go.

Our compiler supports all the features of GoLite. We first present a brief overview of the architecture and comment on our choice of implementation tools, and then dive deeper into the more fine-grained design details of each component.

1 Architecture of the compiler

Our compiler follows the textbook modular compiler architecture. It is made up of several independent components:

Scanner/Lexer: The compiler begins its execution by feeding the input stream through a lexer. The lexer verifies all the input characters are valid, and breaks up the input into logical tokens while throwing away comments and whitespaces. The output of the scanner is a list of tokens.

Parser: The output of the scanner is fed to an LR(1) parser, which verifies that the input conforms to the GoLite syntax, and then converts the flat token stream into an abstract syntax tree representing the logical structure of a Go program. The output of the parser is an untyped abstract syntax tree.

Weeder: Some of the nuances of the GoLite syntax is too tedious to capture with the LR(1) grammar. The weeder operates on the AST and handles these corner cases.

Typechecker: Go (and by extension GoLite) is a statically typed language. The typechecker operates on the AST to verify that all typing rules are obeyed, and annotates the AST it received from the parser with rich type information. The output of the typechecker is a type-annotated abstract syntax tree, linked with a symbol table.

Code Generator: The type annotated GoLite AST is then passed on to the Code Generator, which generates an Jasmin abstract syntax tree for each of the classes that needs to be produced. The output of the code generator is a list of jasmin abstract syntax trees.

Code Emitter: The code emitter takes in the Jasmin Abstract Syntax trees, and outputs a valid jasmin file for each of them.

Assembler/Linker: At the very last step, the produced jasmin files are compiled to java bytecode using the jasmin assembler. Several hand coded java classes are used to provide a runtime environment for the generated bytecode; these classes are copied over from a static directory to accompany the generated class files.

Apart from these main components, the compiler includes a **PrettyPrinter** that converts a Go Abstract Syntax tree to a go file, which was of invaluable assistance during development.

2 Implementation tools

It is said that OCaml is "basically a DSL for writing compilers" [1]. This statement, along our sense of adventure, motivated us to pick OCaml as the main implementation language. It was a fantastic ride, and we believe the functional style of programming made reasoning about code significantly easier. We can't claim to be very disciplined in our coding style, but the OCaml compiler is very reluctant to compile buggy code; almost all the bugs and errors we found stemmed from logical errors or improper understanding of a particular compiler concept rather than OCaml misinterpreting our intentions.[2]

The choice of language naturally motivated the related compiler tools: The scanner is implemented using ocamllex, an OCaml port of the classic flex lexer. The parser is implemented using Menhir, a parser generator which is quite similar to Bison, except that it can work with general LR(1) grammars instead of being constrained to LALR(1) grammars. The implementation of the weeder, typechecker, code generator, and code emitter is done in pure OCaml. Compilation of Jasmin files to bytecode is done with the Jasmin assembler, and there are some Java classes that are compiled using the Java compiler and then eventually copied over to the proper directory using a small shell script.

Some comments on the JVM will be presented in the code generation section.

3 Scanner

Ocamllex is a standard regular expression based lexical analyzer generator, and the implementation of the lexer was quite straightforward. There are three interesting aspects of the scanner that are worth mentioning:

3.1 Handling strings and runes

We needed to support escape characters in strings. Since Ocamllex supports multiple parser entry points, this was done by defining a completely separate parser function for the string with its own set of regular expressions that matched the particular escape sequences and replaced them with the corresponding character code.

3.2 Handling multi-line comments

Handling multi-line comments using a regular expression is surprisingly difficult.
(`/* .* \*/`) does not work because it is too greedy and matches the opening of the first comment to the closing of the last comment. Ocamllex (as well as flex) has no non-greedy

[1]Citation needed

[2]There is one memorable case where we *were* thrown off by a quirk of ocaml: `!=` is not the right inequality operator because it compares the memory locations of two expressions. Rather the correct inequality operator is `<>`. We were handrolling our custom `compare` function for complex types before we realized this issue.

matching, so we tried to come up with all sorts of different contraptions - for example modifying the inner part of the regular expression to match everything except a star (but then it does not accept comments like `/* look at star * */`), or trying something along the lines of `/* (.* \*)* /` (this has the same greedy problem.) Omar was in the process of drawing out a DFA, converting it to NFA, and converting that NFA to a regexp when we realized the newline character inside comments will require special handling because it needs to update the line number counter, thus the regular expression approach will never be satisfactory. We instead implemented multiline comments in a similar way we handle strings - with a separate parser entry point with its own set of regular expressions.

3.3 Semicolon insertions

Go designers, we are sure with nefarious intentions, put in a lexing rule that dictates automatic insertion of a semicolon into the token stream if certain conditions are met. This brought much misery to our lives, but eventually it was handled quite neatly by wrapping the lexing function in another function that keeps track of the last token returned, and adding an action to the newline character that optionally returns a semicolon depending on the last token value.

4 Parser

Implementing the parser made us realize once again that an unambiguous CFG is not the same thing as an conflict-free LR(1) grammar. The first time we compiled our grammar, we had 52 conflicts. It was not a pretty sight. For example, the following completely reasonable looking minimal grammar results in a shift/reduce conflict:

```
%left TADD TSUB
%left TMUL TDIV
binary_exp -> exp op exp
exp -> ... | binary_exp
op -> TADD | TSUB | TMUL | TDIV
```

This only works if op is inlined.

Fortunately all the nuances of LR(1) parsers were eventually understood, and all the conflicts resolved. We mention here some of the design decision we made regarding the parser that made the grammar clear and concise.

- Int, float literals are stored as a string in the parsing phase, and we hoped that we would be able to get away with never converting hexadecimal literals to OCaml ints.[3]

- Operator precedence is implemented by defining five levels of left associative rules, and then two more levels of non associative rules to bind unary operators and parentheses with proper strength.

[3]We didn't - you cannot pass a hexadecimal instruction to `ldc` in jasmin.

- `if ... elseif ... else` statements are converted into `if ... else ( if ... else )` AST nodes, since it does not change the semantics and makes our grammar simpler.
- Determining what is a valid lvalue for assignment statement proved to be quite difficult to do using only the grammar. We accepted arbitrary expressions on the left hand side of assignments, but caught invalid cases in the weeder.
- Type cast of custom types cannot be determined at this phase. For example:

  ```
  type length int; // length is now an alias for int
  y := length(x) // Parser is context free - so it cannot decide whether
      length(x) is a function call or a type cast.
  ```

  We therefore record type cast using custom types as function calls in the AST, and handle function calls with caution forever after in our compiler.[4]

5 Weeder

The primary job of the weeder is to weed out cases from the AST that was too difficult to catch during parsing. Our weeder handles five specific cases:

- Using BlankID as a value:

  ```
    x := _ // not valid
  ```

- Using wrong expressions as left values:

  ```
    1 = 2 // No you cant do that.
    foo() = 1 // Neither can you do this
  ```

- Using a short variable declaration in the for loops post statement:

  ```
  for (i := 1 ; i < 10; y:= 3) {  } //  youre  not allowed to declare a new
      variable in the third clause.
  ```

- Having more than one default case in a switch statement
- Having break statements and continues outside of loops

Apart from this five syntax errors, the weeder also checks that a function of non-void return type has a return statement on every execution path. Of course, since checking this fact perfectly using static analysis is impossible and is equivalent to solving the halting problem, our analysis provides a conservative approximation.

[4]We considered "changing" the node type from a function call to type cast in the typechecker phase, but data being immutable in OCaml, it turned to be very difficult. We accepted this minor inconvenience of putting in a check in whenever we processes a function call expression and moved on with life.

```
// The weeder is not convinced that this has return statement
// on every possible execution path
func foo() int {
  x := true
  if (x) {
    return 1;
  }
}
```

6 Type checker

The typechecker recursively visits the entire abstract syntax tree, and does two main things: builds a symbol table containing important information about all the identifiers used in the program, and verifies that all the expressions in the program are well typed. It does these two things concurrently:

```
  func foo() { // symtable empty
    var x int; // symtable contains x -> int
    x = x + y // type error: symtable does not contain y yet
    var y int;
  }
```

The symbol table is implemented as a tree of scopes. Each scope has a reference to its parent, and contains a hashtable of all the identifiers declared in the scope.

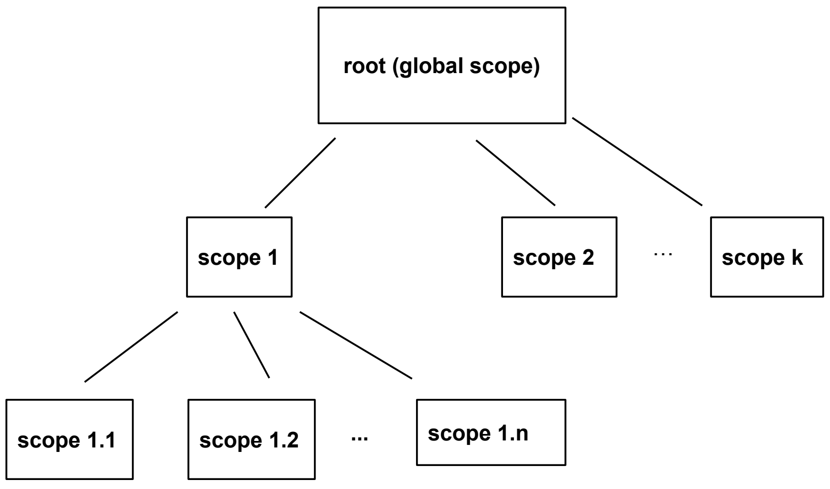

Figure 1: Structure of the symbol table

The hashtable links each symbol to an entry that holds its type and the line number it was declared in, as well as a backward link to the scope. Opening a scope involves adding a new Scope node to this tree, while closing a scope only does some cleanup work (for example printing the symbol table). Note that closing a scope does not destroy a node. Once built, the symbol table is persistent. In addition, the symbol table also records a unique integer for symbol table entry, and this variable number becomes a variables identity in the code generation phase where we have to flatten all the scopes into one jvm method scope. The line number information is handy for printing useful error messages like this:

```
var x int
// do something
var x string // Error: Previous declaration of x at line 1
```

The typechecker also annotates the AST as it typechecks it - it stores the expression type on every expression node, and it binds each identifier in the AST to its symbol table entry.

Once the symbol table structure was put in place, the implementation of the type checker was a straightforward translation of the specifications to code.

7 Code Generation

Code generation was one of the most complicated parts of the project. Let us first say a few nice words about the Java bytecode:

- The java bytecode, perhaps quite deceptively, is statically typed. Types and static typechecking is a wonderful language feature for high level languages, but at the bytecode level it is quite annoying. Our ArrayLists stored values of type object, and consequently every time we retrieved something from the list, we had to put in an explicit `checkcast` instruction to ensure proper types.
- Everything is passed by reference, and comparing two objects checks for equality of their memory locations. Checking equality of contents requires implementing a custom `equals` function.
- There is no way to do low level operations like memcopy, and there is no general way to deep copy a general object.

We could say more about JVM, but we risk turning this report into a rant rivaling the length of the jvm specifications; let us return to discussing code generation.

To run on the jvm, each file must be a class. Let us look at a very simple go file, and the corresponding bytecode our compiler generates:

```
// gofile
package main

var x int = 4
type size int

func main () {
  var y int = 3
  print(x)
  print(y)
}
```

```
; Jasmin file generated by our compiler
.source dev/nice_example.go
.class public GeneratedBytecode
.super java/lang/Object

.field public static x_2 I

.method public <init>()V
  aload_0
  invokespecial java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  .limit locals 25
  .limit stack 25
  ldc 3
  istore 0
  getstatic java/lang/System/out Ljava/io/PrintStream;
  getstatic GeneratedBytecode/x_2 I
  invokevirtual java/io/PrintStream/print(I)V
  getstatic java/lang/System/out Ljava/io/PrintStream;
  iload 0
  invokevirtual java/io/PrintStream/print(I)V
  return
.end method

.method static <clinit>()V
  .limit locals 25
  .limit stack 25
  ldc 4
  putstatic GeneratedBytecode/x_2 I
  return
.end method
```

We draw attention to the few high level code generation principle that you can follow along with this example:

- All top level variable declarations are converted to public static fields in the generated class. The initialization instructions go into the clinit method.
- All local variables are mapped to slots in the locals array. The names of these local variables disappear.
- The type declarations disappear in the compiled class file. All custom types are resolved to basic types.
- Each go function corresponds to a public static method in the class file.

When converting GoLite expressions and statements to jasmin statements, we maintain two invariants:

- After the evaluation of every expression, stack height should increase by either 2 (is the expression is of type float64) or 1 (for all other cases).

- After evaluation of every statement, stack height should remain the same.

Keeping these invariants in mind, code generation proceeds in an obvious recursive manner. We present some of the interesting issues we ran into:

7.1 Generating unique labels

Many of statements (e.g. for loops) requires the use of labels. To generate correct code, it is very important to keep these labels unique. We use a few global counters to generate fresh integers for creating unique labels for different types of statements.

7.2 Managing variables

When we generate code for a method, we first go through all the statements to create a mapping of local variables to local array indices. This mapping maps the the unique *variable number* of a variable (as mentioned in the typechecker section) to a local array index, and thus evades any issues involving the variable with same name being defined in two different scopes.

Loading and storing variables can be quite complex. For example, Generating code for `y = x` means we first load the value of x, and and store the value to the location of y. What load and store instruction to use depends both on the variable type and exactly where it is defined - in this case x could be loaded with either an iload / dload /a load instruction, or might have to be retrieved from the class level variables using getstatic.

7.3 Switch Statement

Since Go supports case expressions of arbitrary types and Java only supports those of primitive and String types, we cannot use tableswitch and lookupswitch instructions. We used regular label jumping to achieve the same result.

7.4 Return Statement

All methods in java bytecode requires a return, but go functions of void return type are allowed to not have any return statement inside. Since having multiple return statements is not an problem, we proactively generate a return instruction at the end of every void function.

7.5 Expression Statement

This involves evaluating the expression (which increases the stack height by one or two) and then popping off those values off the stack to maintain the invariant that statements do not change the stack height.

7.6 Boolean expressions

JVM handles booleans through integers - there is no true or false at the bytecode level. Consequently evaluating the expression `3 < 5` requires more instructions that what one would normally expect:

```
//compiled from docs/report_examples/boolean.go
  ldc 3
  ldc 5
  if_icmplt True_0
False_0:
  iconst_0
  goto EndBoolExp_0
True_0:
  iconst_1
EndBoolExp_0:
```

7.7 Structs

Many C like languages have the struct feature, which is tight packing of a group of data in memory. However, JVM provides no primitive that covers such functionality. To mimic the behavior of structs, we had two options: either handle structs through a general Object to Object HashMap, or define a new class for each struct. The hashmap approach might have been simpler, but it required that we always wrap our ints and doubles in the Integer and Double object types, which imposes a performance penalty. We decided to implement structs by defining classes for them.

We have a seperate compiler phase at the very beginning of code generation that goes through the whole Go AST and collects all the structs ever introduced. It than produces a class for each of those structs, and also a mapping from struct nodes to struct class names so that code generator can properly generate code involving structs. We tried to be economical about how many struct classes we generate, so if two structs were identical it mapped them to the same struct class.

```
var s1 struct { x int; y int;}
var s2 struct { x int; y int;}
var s3 struct { x string; y float64; }
```

produces two struct classes:

```
.source autogenerated_struct_class
.class public StructClass_0
.super GoStructAbstract

.field public x I
.field public y I

; ... init method ...
```

```
.source autogenerated_struct_class
.class public StructClass_1
.super GoStructAbstract

.field public x Ljava/lang/String;
.field public y D

; ... init method ...
```

Creating new structs then becomes creating new instances of these struct classes, and select expression (`struct.field`) and assignments to struct fields becomes an appropriate getfield/putfield instruction.

7.8 Arrays and slices

JVM does have support for multidimensional arrays, but the instructions are quite complex and does not play very well with the recursive nature of array definitions in go where you can have arrays of arbitrary nesting with arbitrarily typed elements. For the sake of simplicity, we decided to implement arrays in terms of ArrayLists, knowing fully well that this poses a performance penalty and that we have to wrap ints and doubles into Integers and Doubles. However, we also implemented slices using ArrayLists, and using the same JVM data structure for both of them saved us a significant amount of work.

7.9 Troublesome assignments

Consider the following program fragment:

```
x := 5;
y := 2;
x,y = 6, x; // y gets the old value of x, not the new value
println(x) // 6
println(y) // 5
```

Thus for multiple variable assignments, we have to evaluate all the expressions first, and then store them into variables. We did not want to introduce new local variables, therefore we took the strategy of evaluating all the expressions first and putting them on the stack, and then storing all the values from the stack in reverse order. To be more concrete:

`x, y = 6, x` first pushes `6` on the stack, then pushes the value of x (5) of the stack. At this point the stack looks like `| 6, 5 | <-- (top)`. We then store the first value of the stack (5) to y, and then store the next value (6) to x.

7.10 (Almost) everything is passed by value

Another issue we encountered was that in go everything except slices is immutable. So we have the following:

```
type point struct{ x int; y int; }
var a [2]point;
var b = a;
b[0].x = 5
println(b[0].x) // prints 5
println(a[0].x) // prints 0. a remains unchanged.
```

We use JVM reference types to implement both structs and arrays, and there is no straightforward way to deep clone a reference type in JVM. We solved this issue by providing a custom clone method to all the mutable reference types we used, and cloning arrays and slices before assigning them to other variables or passing them as function arguments. This clone method must also recursively clone any member variables.

To implement this clone method, we extended the Java ArrayList class to create a custom GoLiteList class, which contained the custom clone method. Structs and Arrays are actually implement using this GoLiteList class instead of directly using ArrayLists. For structs, we made each struct class inherit from a custom GoStructAbstract class. GoStructAbstract defines a clone method that uses Java reflection to access all the fields and recursively clones them. Since objects of Integer, Double, and String classes are immutable, they do not need to be cloned and the recursive calls eventually hit a base case.

7.11 Initializing values

Go assigns default values to all declared variables

```
var a [2]int;
println(a[0]) // 0
println(a[1]) // 0
```

In GoLite we also make sure all declared variables are getting the same initial values as in go. For basic types like ints and strings, this is quite trivial. Declaring an array actually introduces a loop and initializes each array element and pushes them onto the array one by one. Struct initiazation instructions reside in the constructor of the struct classes, therefore simply invoking `<init>` initializes struct variables.

8 Code emitter

There is not much to say regarding the code emitter - it takes in a bytecode ast, determines where to print what and prints them.

9 Linker and Assembler

The GoLiteList and GoStructAbstract are coded in Java. They are precompiled using javac and and copied to the directory of the source file to provite a runtime environment for our compiled code. We also have a small handcoded jasmin file that provided a function to convert boolean types to the string true and false, which is used for printing values of boolean types.

10 Testing code generation

We were fortunate to have a source of ground truth for our compiler - the Go compiler itself. We had a small testing system where we would only write a go file (*any* go file), and the test runner script would compile and run it first with the Go compiler, and compile and run it with our compiler, and then take the diff of the two outputs and pass if the two tests were identical. We could not cover as many test cases as possible due to time constrains, but this testing system was very confidence boosting.

11 Decisions regarding vague specifications

Some aspects of the specification were (perharps deliberately) vague. We took the liberty to choose our implementation to strategy in these cases.

- Our `println` prints all the arguments on a separate line, and `print` prints all its arguments contiguously, i.e. with any extra space between them.
- We use 32 bit ints since runes and booleans would have to use ints anyway and it is nice to model a more things using only a single JVM type.

12 Known issues with the compiler

- There is a typechecker bug regarding custom type of functions:

```
package main

type point int

func boo(a point) {
  return
}

func main() {
  type point int; // We now have a new point
  var a point;
  a = 34;
  boo(a) // a is of type point#s but boo expect type point#1
}
```

  This should be caught in the typechecker, but the datastructure we use to model go types does not capture the difference between the two point types.

- We overzealously clone reference type objects. Slices are passed by reference, and if an array contains a slice, passing the array should only copy the reference to the slice, not the actual content. We did not have time to fix this:

```
package main

func main() {
  var a [2][]int
  var s []int
  s = append(s, 1)
  a[0] = s
  b := a
  a[0][0] = 5
  println(b[0][0]) // Our compiler prints 1, but Go compiler prints 5.
}
```

- We attempted to compute the exact maximum stack height for a method (See `stackHeight.ml` for an partial attend.) but we didn't have enough time. Since the implementation of the code generator was changing all the time as we found more and more bugs, any implementation of the stack height calculator would also be fragile. For now every method is given a max stack height of 25 and local limits of 25.

13 What we would do differently if we started over

- We would add a type called GoVoid that will be the return type of void function. Things would be much simpler if we did that.
- We model types of variables and newly introduced types using the same datatype - resulting in a lot of empty pattern match cases.

14 Conclusion

We had tremendous amount of fun building this compiler. Thanks for the great project!